

## **INFORMATION TO USERS**

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
800-521-0600

**UMI<sup>®</sup>**



Representations and Algorithms for Large Stochastic Planning Problems

by

Kee-Eung Kim

B. S., Korea Advanced Institute of Science and Technology, 1995

Sc. M., Brown University, 1998

A dissertation submitted in partial fulfillment of the  
requirements for the Degree of Doctor of Philosophy  
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2001

UMI Number: 3006748

Copyright 2001 by  
Kim, Kee-Eung

All rights reserved.

UMI<sup>®</sup>

---

UMI Microform 3006748

Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

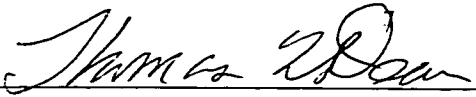
---

Bell & Howell Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

© Copyright 2001 by Kee-Eung Kim

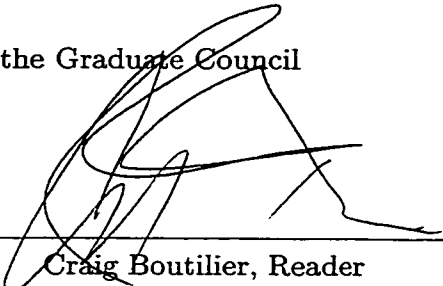
This dissertation by Kee-Eung Kim is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date 05/15/01


  
Thomas L. Dean, Director

Recommended to the Graduate Council

Date May 9, 2001

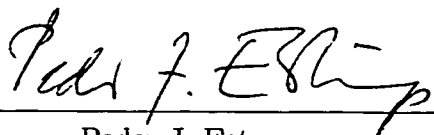
  
Craig Boutilier, Reader  
University of Toronto

Date 4 May 01

  
Leslie Pack Kaelbling, Reader  
Massachusetts Institute of Technology

Approved by the Graduate Council

Date 5/16/01

  
Peder J. Estrup  
Dean of the Graduate School and Research

# Vita

- Name** Kee-Eung Kim
- Born** June 5, 1974 in Seoul, Korea
- Education** *Brown University*, Providence, RI, U.S.A.  
Ph.D. in Computer Science, May 2001.
- Brown University*, Providence, RI, U.S.A.  
Sc.M. in Computer Science, May 1998.
- Korea Advanced Institute of Science and Technology*, Taejon, Korea  
B.S. in Computer Science, August 1995.
- Honors** *Brown University*, Dissertation Fellowship, 2000.
- Korea Advanced Institute of Science and Technology*, Undergraduate Fellowship, 1992-1995.
- No-Yup Foundation*, Undergraduate Fellowship for Academic Excellence, 1994-1995.

# Acknowledgements

There are so many people without whom I could never come to this point. This thesis is dedicated to all of those who either directly or indirectly guided me through my life of learning.

I must start with my advisor, professor Thomas Dean, who has been my *mentor* throughout my studies at Brown. The word mentor in the oriental culture means a lot. A pupil would not step on his mentor's shadow. Although, as a mentor, he has supported me with endless encouragement and helped me understand the important topics of the research, he did not allow me to show him a proper respect as a pupil. I hope I can make it up some day in the future. I am greatly indebted to him about everything, and I can never thank him enough.

I would like to thank my thesis committee members professor Craig Boutilier and professor Leslie Pack Kaelbling for their invaluable advice during my graduate studies and agreeing to be on my committee. Their suggestions and comments especially at my thesis proposal have tremendously helped shaping the thesis.

All the professors at Brown have made my studies very exciting. In particular, professor Philip Klein have helped me formulate some of the theoretical ideas in this thesis. Professor Shriram Krishnamurthi and his wife, professor Kathi Fisler, have also taught me about decision diagrams and temporal logic.

Bob Givan, Milos Hauskrecht and Nicolas Meuleau, who were post-doctorial research fellows at Brown and now professors, taught me what a research is in a down-to-earth fashion. I enjoyed the wholehearted discussions with them, and although we are geographically apart, I hope we can maintain the collaboration.

My fellow AI graduate students, Luis Ortiz and Leonid Peshkin have been good friends through my time at Brown. Besides the friendship, it has been a pure pleasure to have discussion and debate on the current research issues. My senior graduate students, Tony Cassandra, Jose Castanõs, Vaso Chatzi, Sonia Leach, Shieu-Hong Lin, Hagit Shatkay and



Bill Smart have influenced and helped me a lot during my early years at Brown. Joel Young has kindly proof-read this thesis in the midst of a busy semester. My current officemate, Manos Renieris, has also helped me with his supreme technique in L<sup>A</sup>T<sub>E</sub>X and the profound knowledge in software engineering.

I should also acknowledge all my mentors at KAIST, especially professor Gil Chang Kim, professor Myoungho Kim, professor Jin Hyung Kim and professor Kwang Hyung Lee, who has given me encouragement over e-mails and hearted welcome every time I paid a visit to KAIST.

Additional thanks to the Niners, a gang of Seoul science high school and KAIST alumni. There was nothing better than sharing everyday life with folks who have been friends for 11 years. My college roommate Seonghwan Cho at MIT often made visits to Providence and we counseled each other over drinks.

My family has been the foremost supporter throughout my life, including my stay at Brown. I thank them for their nurture, motivation, support, love, and everything. Lastly but most importantly, my grandfather, Yoon-Haeng Kim, in memory, who first introduced to the Korean bar society the idea of using AI for legal reasoning. Without all the science toys he picked up for me during my childhood, I might be seeking my career in law at this moment as he did.

# Contents

<b>List of Tables</b>	<b>ix</b>
<b>List of Figures</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Contribution . . . . .	4
1.3 Outline of the Dissertation . . . . .	5
<b>2 Mathematical Preliminaries</b>	<b>7</b>
2.1 Markov Chain Theory . . . . .	7
2.2 Monte Carlo Estimation Techniques . . . . .	10
2.3 Markov Decision Processes (MDPs) . . . . .	11
2.4 Stable Value Function Approximation (VFA) . . . . .	20
2.5 Stochastic Bisimulation Equivalence in MDPs . . . . .	22
<b>3 Factored MDPs (FMDPs)</b>	<b>27</b>
3.1 Representation and Complexity . . . . .	27
3.2 History Dependent Policies . . . . .	34
<b>4 FMDP Planning</b>	<b>40</b>
4.1 Homogeneous Aggregation Algorithms . . . . .	41
4.1.1 Experiments . . . . .	54
4.1.2 Related Work . . . . .	60
4.2 Non-homogeneous Aggregation Algorithms . . . . .	61
4.2.1 Experiments . . . . .	69
4.2.2 Related Work . . . . .	77
4.3 Decomposition Algorithms . . . . .	78

4.3.1	Experiments . . . . .	84
4.3.2	Related Work . . . . .	88
4.4	Model-Based Reinforcement Learning Techniques . . . . .	89
4.4.1	Experiments . . . . .	93
4.4.2	Related Work . . . . .	96
<b>5</b>	<b>Structured Linear Algebra</b>	<b>98</b>
5.1	Exploiting the Regularities in Matrices and Vectors . . . . .	99
5.2	Representing Matrices and Vectors . . . . .	102
5.3	Elementary Structured Operators . . . . .	104
5.4	Approximating Vectors with Large Representation Size . . . . .	112
5.4.1	Error Analysis from Numerical Techniques . . . . .	113
5.4.2	Averagers and Stable Approximations . . . . .	115
5.5	Applications and Analyses . . . . .	116
5.5.1	Solving Systems of Linear Equations . . . . .	116
5.5.2	Calculating Eigensystems . . . . .	126
5.5.3	Singular Value Decomposition . . . . .	129
5.6	Related Work . . . . .	134
<b>6</b>	<b>Conclusions</b>	<b>135</b>
6.1	Future Work and Open Problems . . . . .	137
	<b>Bibliography</b>	<b>139</b>

# List of Tables

4.1	Results from the FA-FMDP minimization algorithm using decision trees. . .	57
4.2	Results from the FA-FMDP minimization algorithm using ADDs. . . . .	59
4.3	Summary of results after 100 iterations on EXPON and FACTORY. . . . .	75
4.4	Comparison of the non-homogeneous partitioning algorithm and the APRICODD algorithm on EXPON domain. The number inside the parentheses is the pruning parameter determined by “sliding-tolerance” pruning. “Perf” is the ratio between the performance of the optimal policy and that of the approximate policy assuming uniform starting distribution on states. We also present the number of nodes and leaves (terminal nodes) in the ADD representation of the approximate value function from the APRICODD. . .	77
4.5	Comparison of the non-homogeneous partitioning algorithm and the APRICODD algorithm on the FACTORY domain. . . . .	77
5.1	Experimental results with exact operations. . . . .	123
5.2	Experimental results with approximate operations. “Rel err” denotes the relative error incurred by approximating the vector. The threshold $\theta$ used for tree pruning algorithm heuristic was set to 1.0 throughout the experiments on the Richardson method and 5.0 on the extrapolation method. . . . .	123
5.3	Experimental results with approximate operations with classical pruning method. $\theta$ is the threshold used for pruning. “Rel err” denotes the relative error incurred by approximating the vector. $\epsilon$ and $\gamma$ were set to 0.1 and 0.8, respectively. . . . .	125
5.4	Experimental results with approximate operations with pruning using elimination variables. “Rel err” denotes the relative error incurred by approximating the vector. $\epsilon$ and $\gamma$ were set 0.1 and 0.8, respectively. . . . .	126
5.5	Experimental results with ADDs on the FACTORY domain. . . . .	126

5.6	Results of the structured power method on the Coffee Robot domain. For the transition probability matrices corresponding to actions <i>stay</i> , <i>go1</i> , and <i>go2</i> , we calculated the dominant eigenvalue and eigenvector for each action. For the various pruning parameter $\theta$ , we ran the power method for 20 iterations and show the running time $T$ in seconds and the number of terminal nodes. Although the matrices for the actions <i>stay</i> and <i>go2</i> were different, the calculated eigenvalues and eigenvectors were the same. . . . .	129
5.7	Results of the structured power method on the FACTORY domain using ADDs. For the transition probability matrices corresponding to actions <i>polisha</i> , <i>polishb</i> , and <i>glue</i> , we calculated the dominant eigenvalue and eigenvector for each action. For the various pruning parameter $\theta$ , we ran the power method for 20 iterations and show the running time $T$ in seconds and the number of nodes in the ADDs. . . . .	129
5.8	Results of the structured SVD on the Coffee Robot domain. For the transition probability matrices corresponding to actions <i>stay</i> , <i>go1</i> , and <i>go2</i> , we calculated the first two dominant singular values, $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$ , and their associated vectors, $U_1$ and $U_2$ . $ U_1 $ and $ U_2 $ are the numbers of terminal nodes in $U_1$ and $U_2$ , respectively. For varying degree of the pruning parameter $\theta$ , we run power method for 20 iterations for the matrix $A^T A$ and show the running time $T$ in seconds. . . . .	133
5.9	Results of the structured SVD on the FACTORY domain. For the transition probability matrices corresponding to actions <i>polisha</i> , <i>polishb</i> , and <i>glue</i> , we calculated the first two dominant singular values, $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$ , and their associated vectors, $U_1$ and $U_2$ . $ U_1 $ and $ U_2 $ are the numbers of nodes in $U_1$ and $U_2$ , respectively. We run power method for 20 iterations for the matrix $A^T A$ and show the running time $T$ in seconds. . . . .	133

# List of Figures

1.1	A planning domain with completely independent processes. The domain is composed of three independent shipping tasks from factories to retailers. Each transport has only one designated factory and retailer, and no two transports share the same factory or retailer. . . . .	2
1.2	A planning domain with shared transports. Each transport serves a small number of factories and retailers. We cannot decompose the domain with three independent tasks, but we can still represent compactly. . . . .	3
2.1	A finite state space Markov chain . . . . .	8
2.2	MDP policies. $\pi$ represents an infinite sequence of decision rules for executing an action at each step, so that $\pi \equiv [d_0, \dots, d_t, \dots]$ . . . . .	13
2.3	The discount rate $\gamma$ can be seen as the probability of living another time step. Once the agent reaches the “dead” state, it stays there and no reward is given throughout the future. . . . .	14
2.4	Value iteration algorithm. . . . .	16
2.5	Policy iteration algorithm. . . . .	18
2.6	An MDP subject to minimization. The reward function is given as $R(s_1) = R(s_2) = 0$ and $R(s_3) = R(s_4) = 1$ . . . . .	25
2.7	MDP minimization algorithm . . . . .	26
3.1	Graphical representation of an FMDP describing a toy robot domain. . . .	28

3.2	Comparing a decision tree policy (left) with a looping plan (right). In a decision tree policy, the action is found by following down the path from the root to the leaf determined by the current state. In a looping plan, the action is determined by following the outgoing edge determined by the current state. We can convert a decision tree policy to a looping plan by preparing node for each terminal node of the decision tree and labeling incoming edges with the path formula of the terminal node. Hence, the conversion can be done without the explosion in representation size, but not vice versa. The two examples in the figure are not equivalent. . . . .	31
3.3	Constructing the FMDP from a circuit. . . . .	32
3.4	A task determining majority represented as a FMDP. (a) answer generation phase : In this example, $Y$ is sampled to be $k$ at time $t$ . (b) problem instance generation phase : Then, exactly $k$ fluents among $X_1, \dots, X_n$ are set to 1 by the time $t + 2n$ . In this example, $X_1$ is sampled to be 0 and $X_n$ to be 1. . .	35
3.5	A task determining majority represented as a FMDP (continued from Figure 3.4). (c) answer guess phase: In this example, at time $t + 2n$ , the guess “yes” is fed into the FMDP. (d) answer verification phase: The guess answer is checked and the reward of 1.0 will be given at time $t + 3n$ if and only if $-n/2 < -n + k \leq n/2$ . . . . .	36
3.6	CPTs for fluents $X_i$ (top) and $Y$ (bottom). . . . .	38
4.1	Partition of the state space induced by CPTs . . . . .	45
4.2	Graphical representation of a FA-FMDP. . . . .	46
4.3	The first version of FA-FMDP minimization algorithm. . . . .	47
4.4	The second version of FA-FMDP minimization algorithm. . . . .	48
4.5	BACKUP( $P$ ) in FA-FMDP minimization using decision trees. We construct intermediate partitions storing the probabilities for each terminal node in $P$ , then take the intersection to calculate $I_P$ . . . . .	50
4.6	An example of a decision tree and an ADD. . . . .	52
4.7	A bad case of existential abstraction. When we sum over action fluent $A_1$ , two blocks merge since the additions of the block numbers are the same. . .	53
4.8	The algorithm for projecting ADD representation of a homogeneous partition onto the state space. . . . .	53
4.9	The algorithm for the BACKUP operator on ADD representation of a partition. . . . .	54

4.10	State and action fluents for the ROBOT- $k$ domain. This is an example where there are 4 rooms ( $k = 4$ ). The room numbers are encoded as bit vectors. . . . .	55
4.11	An example of JOBSHOP- $k$ - $l$ domain with 4 machines and 6 tasks. The solid arrows between tasks specify dependencies as pre-requisites. The dashed arrows between machines and tasks specify allocations. . . . .	56
4.12	The algorithm for projecting the ADD representation of a homogeneous partition onto the action space. . . . .	60
4.13	An FMDP representing a 3-bit counter. Note that CPTs are still of size polynomial in the number of fluents since they are skewed trees. There is only one action in the FMDP. The size of the coarsest homogeneous partition is $2^3 = 8$ . We can generalize this example to an $n$ -bit counter. See Boutilier <i>et al.</i> [16] for a similar example with $n$ actions. . . . .	61
4.14	Chopping the block $X_4 \wedge X_5$ with respect to $X_1, X_2$ and $X_3$ . Note that the blocks resulting from the CHOP operator are not stable in general. . . . .	63
4.15	Value functions for an MDP constructed from non-homogeneous partition. The figure illustrates an example where the MDP has two states and the partition has one block. The dotted line is the set of representable approximate value functions. $d_1 \leq 2(1 + \frac{\gamma}{1-\gamma}\epsilon)$ and $d_2 \leq \frac{\ LV_P^* - V_P^*\ _\infty}{1-\gamma}$ . . . . .	64
4.16	The algorithm for finding a non-homogeneous partition for an approximately optimal policy . . . . .	66
4.17	The algorithm for calculating $T_{P'}$ . . . . .	68
4.18	Calculating $T_{P'}(B, a, C_i)$ with decision trees. . . . .	69
4.19	COFFEE domain (6 variables, 64 states). The non-homogeneous partitioning algorithm found the optimal policy after 6 splits, totaling 10 blocks in the partition. The optimal value is 8.11727. . . . .	70
4.20	LINEAR domain (14 variables, 16,384 states). The optimal policy of the aggregate MDP from the reward partition yields the optimal value, <i>i.e.</i> , the optimal policy was obtained without any split. The optimal value is 257.356. . . . .	70
4.21	EXPON domain (12 variables, 4096 states). After 100 splits (113 blocks), the non-homogeneous partitioning algorithm yields an approximately optimal policy. The optimal value is $2.4 \times 10^{16}$ . . . . .	71
4.22	FACTORY domain (17 variables, 131,072 states). After 100 splits (126 blocks), the non-homogeneous partitioning algorithm yields an approximately optimal policy. The optimal value is 28.4. . . . .	71



4.23	Distance plots between two value functions of successive aggregate MDPs ( $\ V_{P^*} - V_P^*\ $ in Figure 4.16). From left to right: COFFEE, EXPON, FACTORY. . . . .	71
4.24	Munos and Moore's heuristic on COFFEE domain (6 variables, 64 states). The optimal value is 8.11727. . . . .	74
4.25	Munos and Moore's heuristic on LINEAR domain (14 variables, 16,384 states). The optimal value is 257.356. . . . .	74
4.26	Munos and Moore's heuristic on EXPON domain (12 variables, 4096 states). The optimal value is $2.4 \times 10^{16}$ . . . . .	74
4.27	Munos and Moore's heuristic on FACTORY domain (17 variables, 131,072 states). The optimal value is 28.4. . . . .	75
4.28	Distance plots between two value functions of successive aggregate MDPs ( $\ V_{P^*} - V_P^*\ $ in Figure 4.16). From left to right: COFFEE, EXPON, FACTORY. . . . .	75
4.29	Random partitioning on COFFEE (upper-left), LINEAR (upper-right), EXPON (lower-left) and FACTORY (lower-right). . . . .	76
4.30	The MTD algorithm in the on-line phase. . . . .	82
4.31	The running times of MTD. Each plot is an average running time of 30 randomly generated problems with size parameter $s$ , so that the problem involves $100s$ sites, $1000s$ packages, and $10s$ airplanes. Note that the running times on problems with airplane constraints are larger and more varied than those without airplane constraints. . . . .	85
4.32	Performances of the policies on a randomly generated 5-site problem without airplane constraints. . . . .	86
4.33	Performances of the policies on a randomly generated 7-site problem with airplane constraints. . . . .	86
4.34	Performances of the policies on a randomly generated 200-site problem. . . . .	87
4.35	Performances of the policies on a randomly generated 200-site problem with 20 airplanes. . . . .	87
4.36	Performance on the modified shuttle problem. . . . .	94
4.37	Performance on the Coffee Robot domain. . . . .	95
4.38	Performance on the Majority domain with 5 bits. . . . .	96
5.1	A vector represented as a decision tree. . . . .	102
5.2	Transition matrix for a robot action. . . . .	104

5.3	Adding two vectors represented as decision trees. . . . .	106
5.4	Building intermediate decision tree for taking inner product of two vectors represented as decision trees. . . . .	107
5.5	Intermediate steps in multiplying the transpose of a matrix with a vector. .	108
5.6	Vector resulting from Figure 5.5. . . . .	108
5.7	Modification of CPT for $W$ by introducing the index variable at the next time step. . . . .	109
5.8	Multiplication of the transition probability matrix with a vector (step 1). .	110
5.9	Multiplication of the transition probability matrix with a vector (step 2). .	110
5.10	Left multiplication of the transition probability matrix with a vector (step 3).	111
5.11	An example illustrating the pruning process and the resulting averager matrix. The example follows the domain given in Figure 3.1. The four diagonal blocks are of size $8 \times 8$ and all of their components are $1/8$ . . . . .	115
5.12	The plot of $\ V^\pi - V^{(n)}\ $ of the Richardson and the extrapolation method at each iteration. The graph showing the slower convergence (upper curve) is the plot from the Richardson method. . . . .	125

# Chapter 1

## Introduction

### 1.1 Motivation

Planning has been one of the core areas of research in AI since the beginning of the field. A typical scenario involves an intelligent agent and a representation of the environment so that the agent has to achieve specified goals or maintain a certain level of performance. At each discrete point in time, the agent observes the state of the environment and makes a decision about which action to execute based on a rule. Such a rule in decision making is called a *plan*. Examples of planning problems include factory maintenance, military and industrial logistics, construction planning and many others (Wilkins [109]). Depending on the domain, these planning problems belong to one of the two categories: deterministic or stochastic. The deterministic planning problems deal with domains in which the effect of every action on the environment is deterministic. The main focus of this thesis is on stochastic planning problems — problems with domains in which the effects of actions are stochastic. Although there has been a lot of progress in developing practical algorithms for deterministic planning problems (Blum and Furst [12], and Kautz and Selman [63]), there has been little success with stochastic planning problems.

One of the most common frameworks for modeling stochastic planning problems is the Markov Decision Process (MDP) (Puterman [93]). Assuming that the dynamics of the environment is Markovian, the domain can be described in terms of the transition probability for each state-action-state triple and the reward function for each state-action pair. The transition probability distribution specifies the probabilistic effect of executing each action on the state at the next time step, and the reward function gives the immediate payoff of executing each action in each state. If the problem is small enough so that we can

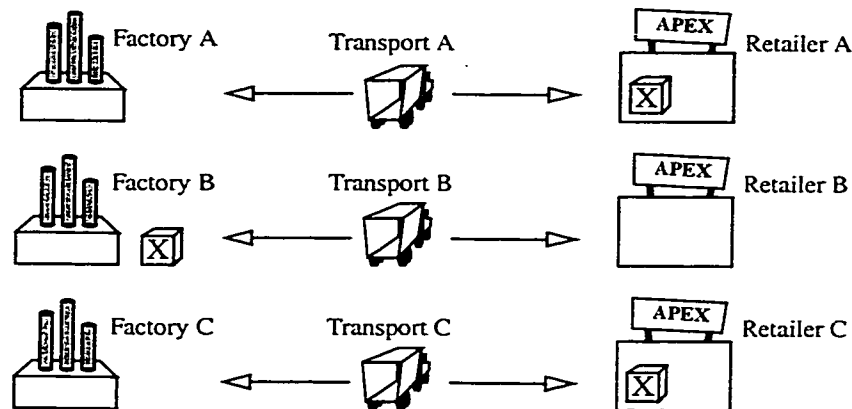


Figure 1.1: A planning domain with completely independent processes. The domain is composed of three independent shipping tasks from factories to retailers. Each transport has only one designated factory and retailer, and no two transports share the same factory or retailer.

efficiently enumerate all the states of the domain, we can cast the transition probabilities and the reward function in terms of matrices, and such a representation enables us to use standard MDP algorithms that solve the problem in polynomial time in the number of states and actions.

To cast a stochastic planning problem as an MDP, we have to find an appropriate representation that allows us to describe it in reasonable size. As is often the case with real world problems, this condition is very difficult to meet. Since the state space should include all the states in the environment, its typical size ranges from thousands to millions. Hence, over the years, researchers have used *factored* representations for MDPs to describe large stochastic planning problems. A common approach to a factored representation is to use a *fluent* to symbolize a logical entity or a particular aspect in the domain. For example, in a logistics problem shown in Figure 1.1, we describe the domain in terms of nine fluents, which are three factories, three transports and three retailers, and how the transports change the states of factories and retailers. Note that if we plainly encoded this domain as an MDP, the number of states is exponential in the number of factories, transports and retailers.

There are two main advantages to representing MDPs in a factored form.

First, the representation itself can reveal a good deal of information about various *regularities* that exist in the domain. Back to our previous example, Figure 1.1 shows a special case where the domain is composed of three completely independent processes. The transports between different pairs of factories and retailers do not interfere with one another. It follows that we can decompose the domain into three separate subproblems, each of

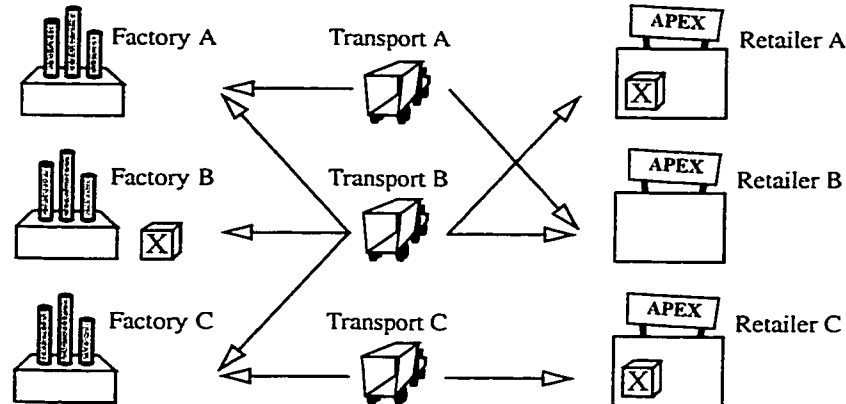


Figure 1.2: A planning domain with shared transports. Each transport serves a small number of factories and retailers. We cannot decompose the domain with three independent tasks, but we can still represent compactly.

which can be solved independently. The problem becomes complicated when it is no longer completely decomposable, as in Figure 1.2. Although the domain is not completely decomposable, we may be able to identify nice properties such as “Transport A and transport B behave the same under certain settings of factories and retailers” or “It never happens that transport A and transport B ship items from factory A at the same time”. These are just a few examples of different types of *regularity* that can provide us with a computational advantage when we design the algorithms for solving the planning problems. In the later sections of the thesis, we will explore various types of regularity in the domain and the algorithms for identifying and exploiting specific types of regularity.

A second advantage to representing MDPs in a factored form is that, in some cases, we can adapt classical algorithms that were originally developed for MDPs. These classical algorithms include exact algorithms that provide an optimal solution and approximate algorithms that find an approximately optimal answer for MDPs. However, in general, nontrivial issues have to be settled to apply these algorithms, and using non-traditional techniques seems more appealing in some cases. Some parts of this thesis will demonstrate the effectiveness and difficulty in adapting the classical algorithms.

Existing factored representations include Probabilistic STRIPS Operators (PSOs) (Kushmerick *et al.* [72], Boutilier *et al.* [14]) which extend the original STRIPS operators (Fikes *et al.* [40]) used in deterministic planning problems so that they can compactly express the probabilistic effects of actions. PSOs describe the effect of each action in terms of fluents that are mentioned earlier in this section. Each of these fluents captures a particular aspect of the domain, and the probabilistic effect on fluents is encoded as a compact

list of changes in a small set of fluents and the probabilities of such changes. As another example of widely used factored representations, Factored MDPs (FMDPs) encode the state space of MDPs also in terms of fluents, and further assume that the effects on fluents are independent given all the values of fluents in the previous time step. An assignment to each and every fluent uniquely defines a state in the MDP when the state space is explicitly enumerated. Thus, the transition probability between each state of the environment is obtained by multiplying the transition probability of each fluent.

There have been other compact representations for stochastic planning problems as well. In continuous domains, where the state space is not discrete, it is a common practice to use parameterized functions for describing the transition probability distribution and the reward function. In general, algorithms for continuous state space stochastic planning problems exploit the continuity of the change in the dynamics of the environment when the state space is discretized. FMDPs lack this property, hence we cannot directly adapt such algorithms. However, they are relevant in the sense that they all pertain to the MDP framework and the techniques for MDPs with continuous state spaces have provided insights for developing techniques for MDPs with discrete state spaces (Chapter 6 of Dean and Wellman [37], Hernandez-Lerma [51], Moore and Atkeson [84], Munos and Moore [86]).

Another type of representation for encoding the environment is related to model-free reinforcement learning (Kaelbling *et al.* [61]). Model-free reinforcement learning assumes the environment is given as a black box that takes as its input the action executed by the agent and the current state of the environment, and generates as its output the next state of the environment. The algorithm uses only past experiences gathered from the black box — the explicit description of the problem is hidden from the reinforcement learning algorithm and the black box acts as a simulator of the environment (Singh and Bertsekas [96], Mahadevan *et al.* [78], Kearns *et al.* [64]). Since we can construct the simulator from the description of an FMDP, we can adapt reinforcement learning algorithms to work on the FMDP.

## 1.2 Contribution

The main contribution of this thesis is answering (at least in part) the following question:

*What types of regularity found in FMDPs provide us with computational leverage and how do we identify these types of regularity and take advantage of them in designing new algorithms?*

The FMDP assumes that the interactions among fluents are local in the sense that these interactions are limited to consecutive time steps. The local interaction assumption enables us to describe the domain as a collection of concise specifications on how individual fluents change in the next time step. This is the most apparent regularity in the FMDP representation. Once again, Figure 1.2 shows a domain that is not completely decomposable but yet fluents, which are factories, transports and retailers, interact locally. Meanwhile, the types of regularity that lead to efficient algorithms can be quite different. The local interactions among fluents can affect the whole set of fluents in the long term. Since the objective of the algorithms is to maximize the long-term payoff, we have to find the types of regularity that are preserved throughout the infinite (or some finite) horizon. Note that the example statements we made in the previous section hold throughout the infinite horizon. These sorts of regularity are hidden in the sense that it requires a significant amount of computational resource to identify, often comparable to that needed for solving the problem itself. However, the revealed regularity can shed light on better understanding of the domain, and thus it will expedite the development of problem-specific algorithms with high performance that no general purpose algorithm can provide. For each technique we present, we will define what types of regularity of the domain the algorithm is exploiting.

Boutilier *et al.* [14] comprehensively survey the published work and ongoing research on finding the useful types of regularity in FMDPs. Some parts of the work presented in this thesis are included in the survey. There are also non-traditional techniques to the AI planning community, namely reinforcement learning techniques, which have emerged as promising for solving large planning problems including FMDPs (Kearns *et al.* [64], Kim *et al.* [68]).

### 1.3 Outline of the Dissertation

The rest of the thesis is organized as follows:

We will start out in Chapter 2 with mathematical preliminaries including the definition of MDPs and related traditional MDP algorithms. When solving MDPs, efficient algorithms using dynamic programming, such as value iteration (Bellman [7]) and policy iteration (Howard [56]), have been widely used for decades. Most of the current FMDP algorithms borrow their basic ideas from these algorithms. Hence, it is crucial that we review the traditional techniques to understand the ideas behind the new techniques we develop and those originated by others.

We discuss FMDPs in Chapter 3, particularly their representation and computational

complexity. We will prove that the problem of solving FMDPs belongs to the complexity class PSPACE-complete, and this fact forms the motivation for our approximation and local search techniques in the later sections.

In Chapter 4, we describe our techniques for solving FMDPs. We point out that we can concentrate on a few properties of the problem that makes solving FMDPs hard. In fact, the algorithms we present in the chapter are designed to overcome those obstacles by identifying and exploiting certain types of regularity in the domain. In some domains, we may be able to aggregate the states of FMDPs that behave the same and solve the problem with significantly reduced state space. In other domains, we may be able to decompose the problem into a small number of independent subproblems and produce the solution by efficiently combining the sub-solutions. The difficulty comes from the fact that these structural regularities are hard to identify. There is currently no efficient algorithm for determining if the aggregation technique will lead to a small state space or the decomposition technique will produce a solution sufficiently close to the true solution. As such, it is hard to identify which type of regularity is present and which solution technique to apply. The techniques described in Chapter 4 will provide us with insights so that we can deploy appropriate methods for domains with certain types of regularities.

In Chapter 5, we step back from the FMDP techniques and establish a framework for handling vectors and matrices of very large dimension. This will in turn provide us with new techniques for solving FMDPs. The motivation for this work comes from the fact that some of the FMDP algorithms can be seen as extensions of classical numerical algorithms to very high dimension, once we have an appropriate implementation of linear algebra operators such as matrix-vector product, vector addition, *etc.* We also explain a way to approximate the results of these linear algebra operators when exact calculation results in an explosion in the size of the representation for vectors.

Finally, we present conclusions and open problems in Chapter 6.



## Chapter 2

# Mathematical Preliminaries

This section reviews the basic mathematical background needed to understand and appreciate the results presented in this thesis. The material regarding Markov chain theory in Section 2.1 presents useful properties of stochastic dynamical systems. Section 2.2 summarizes basic ideas concerning Monte Carlo estimation. Markov decision processes (Section 2.3) provide the classical framework for modeling stochastic planning problems. We also review recent work on approximating value functions (Section 2.4), which are useful when the stochastic planning problem is large. Section 2.5 describes Markov decision processes as finite state automata and explains how to apply the notion of model minimization to Markov decision Processes.

### 2.1 Markov Chain Theory

The Markov chain, or the Markov process, is one of the standard models for stochastic processes. It is one of the most well studied mathematical tools. The precise definition is as follows:

**Definition 2.1.1 (Markov Chain)** *A Markov chain  $M = \{S, T\}$  is defined as*

- *A Borel space  $S$  is the set of states,*
- *$T$  is the set of transition probabilities:*

$$T(s_t, s_{t+1}) = P(s_{t+1}|s_t)$$

*where  $s_t$  and  $s_{t+1}$  denote the state at time  $t$  and  $t + 1$ , respectively.*

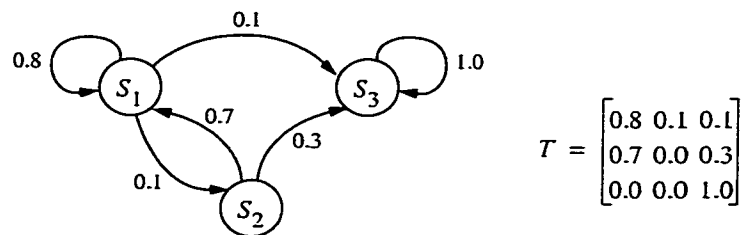


Figure 2.1: A finite state space Markov chain

For a Markov chain  $M = \{S, T\}$  with a finite state space (*i.e.*, the set  $S$  is finite), we have an equivalent definition using a directed graph:

**Definition 2.1.2 (Finite State Space Markov Chain)** A finite state space Markov chain  $M = \{S, T\}$  is a graph  $G = \{V, E, W\}$  where

- $V$  is the set of vertices that represent the state space,
- $E$  is the set of edges,
- $W$  is the set of weights on edges such that

$$\exists e : v_1 \rightarrow v_2 \text{ then } w(e) = T(v_1, v_2).$$

Figure 2.1 shows an example of a Markov chain with finite state space. Note that for each vertex the sum of weights on outgoing edges is 1. The weights correspond to non-zero entries in the transition probability matrix  $T$  in the same figure where the row is the origination vertex and the column is the destination vertex.

Markov chain theory provides useful definitions of properties of Markov chains. As we will see in the next section, the underlying dynamics of a MDP is essentially a Markov chain. Since some of the MDP algorithms assume certain properties of the chain structure of the domain, we present some relevant results from Markov chain theory. The material we show here can also be found in Puterman [93] and Motwani and Ragahavan [85].

Let  $T^k(s, s')$  denote the probability of visiting state  $s'$  at time  $k$  starting from state  $s$ . State  $s$  is called *recurrent* if and only if

$$\sum_{k=0}^{\infty} T^k(s, s) = \infty$$

and *transient* if and only if

$$\sum_{k=0}^{\infty} T^k(s, s) < \infty.$$

Among recurrent states, if state  $s$  has a finite expected first return time as defined by

$$\sum_{k=0}^{\infty} kF^{(k)}(s, s) < \infty$$

where

$$F^{(k)}(s, s') \equiv P(s_0 = s \text{ and } s_k = s' \text{ and for } 1 \leq t \leq k-1, s_t \neq s').$$

state  $s$  is *positive recurrent* otherwise it is *null recurrent*. Subset  $C \subseteq S$  is said to be *closed* if no state outside of  $C$  is accessible from any state in  $C$ ,

$$\forall s \in C, \forall s' \in (S - C), T^k(s, s') = 0 \text{ for any } k.$$

A closed set  $C$  is *irreducible* if no proper subset of  $C$  is closed. Note that every recurrent state is a member of some irreducible set.

The state space of a Markov chain can be partitioned into a number of disjoint closed irreducible sets  $C_1, \dots, C_m$  and the set of transient states  $E$  so that  $S = C_1 \cup \dots \cup C_m \cup E$ .

By relabeling the states in a Markov chain, we can transform any transition probability matrix  $T$  to an equivalent transition probability matrix  $T'$  given as

$$T' = \begin{bmatrix} T_1 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & T_2 & 0 & \cdot & \cdot & 0 \\ \cdot & & \cdot & & & 0 \\ \cdot & & & & & \cdot \\ 0 & & & T_m & & 0 \\ Q_1 & Q_2 & \cdot & \cdot & Q_m & Q_{m+1} \end{bmatrix},$$

where the block  $T_i$  is the transition probability matrix between states in  $C_i$ ,  $Q_i$  from  $E$  to states in  $C_i$ , and  $Q_{m+1}$  between states in  $E$ . We call  $T'$  the *canonical form* of  $T$ . Partitioning the state space of a Markov chain to irreducible sets of recurrent states and the set of transient states can be done using the Fox-Landi algorithm (Fox and Landi [42]).

Depending on the number of closed irreducible sets, a (finite state space) Markov chain is called *unichain* if it has only one such set and *multichain* otherwise.

The *periodicity* of state  $s$  is defined to be the greatest common divisor of all  $k$  for which  $T^k(s, s) > 0$ . State  $s$  is called *aperiodic* if the periodicity is 1. Note that all the states in the same closed irreducible set have the same period. A Markov chain is called *aperiodic* if every state is aperiodic.

An *ergodic* state is one that is aperiodic and positive recurrent. An *ergodic* Markov chain is one in which all states are ergodic. Note that any unichain, aperiodic Markov

chain is ergodic. Standard Markov chain theory shows that any ergodic chain has a unique *stationary distribution*  $p_\infty$  such that

$$p_\infty = p_\infty T.$$

In other words, if the probability distribution on states at time  $t$  reaches the stationary distribution, the distribution remains the same at time  $t + 1$ .

Another important property of a Markov chain is its *mixing time*. It is the rate of convergence of a Markov chain to the stationary distribution. Formally, the mixing time  $\tau(\epsilon)$  is defined as

$$\tau(\epsilon) \equiv \min\{k : |T^{k'}(s, s') - p_\infty(s')| \leq \epsilon \text{ for all } k' \geq k \text{ and } s \in S\},$$

which means the earliest time when the error between the probability distribution on states and the stationary distribution is within  $\epsilon$ . A detailed discussion on estimating the mixing time can be found in Jerrum and Sinclair [59].

## 2.2 Monte Carlo Estimation Techniques

Suppose that we want to calculate an approximate solution for the conditional expectation of function  $f$  given as

$$E[f(\vec{Z})|\vec{y}] \equiv \sum_{\vec{z} \in \Omega_{\vec{z}}} f(\vec{z})p(\vec{z}|\vec{y}), \quad (2.1)$$

where  $\vec{Z} \equiv [Z_1, \dots, Z_n]$  is a vector of random variables with its sample space  $\Omega_{\vec{z}} \equiv \prod_i \Omega_{Z_i}$ . The exact calculation of the summation is often intractable for a vector of random variables of a high dimension. A simple approach for obtaining an approximate solution is to select  $N$  random points uniformly in  $\vec{Z}$ , evaluate  $f(\vec{z})$  and  $p(\vec{z}|\vec{y})$  at the sampled points, and then take the sample mean:

$$\hat{E}[f(\vec{Z})|\vec{y}] = \frac{1}{N} \sum_{i=1}^N f(\vec{z}_i)p(\vec{z}_i|\vec{y}), \quad \vec{z}_i \sim \text{uniform}(\Omega_{\vec{z}}) \quad (2.2)$$

where  $\vec{z}_i$  denote the sampled points from the uniform distribution on the sample space  $\Omega_{\vec{z}}$ . The *law of large numbers* states that as  $N \rightarrow \infty$ ,  $|\hat{E}[f(\vec{Z})|\vec{y}] - E[f(\vec{Z})|\vec{y}]| \rightarrow 0$ . Equation 2.2 is the simplest form of Monte Carlo estimation methods for estimation.

Another way of estimating Equation 2.1 is to sample  $\vec{z}$  from the probability distribution  $p(\vec{z}|\vec{y})$ . The estimate is obtained by

$$\hat{E}[f(\vec{Z})|\vec{y}] = \frac{1}{N} \sum_{i=1}^N f(\vec{z}_i), \quad \vec{z}_i \sim p(\vec{z}_i|\vec{y}). \quad (2.3)$$

Note that the above equation assumes that it is easy to sample from the distribution  $p(\vec{z}|\vec{y})$ . We still have the guarantee that Equation 2.3 converges to the exact solution as the number of samples approaches infinity.

We can make an important observation from Equation 2.2 and Equation 2.3 that we can choose any sampling distribution and an appropriately weighted mean of the samples serves as the estimator. *Importance sampling* provides a general framework for choosing the sampling distribution. Let  $g$  be a chosen sampling distribution, then we rewrite Equation 2.1 as

$$\sum_{\vec{z} \in \Omega_{\vec{z}}} f(\vec{z})p(\vec{z}|\vec{y}) = \sum_{\vec{z} \in \Omega_{\vec{z}}} \left( \frac{f(\vec{z})p(\vec{z}|\vec{y})}{g(\vec{z})} \right) g(\vec{z}),$$

which means that the new estimator is given as

$$\widehat{E}[f(\vec{Z})|\vec{y}] = \frac{1}{N} \sum_{i=1}^N \frac{f(\vec{z}_i)p(\vec{z}_i|\vec{y})}{g(\vec{z}_i)}, \quad \vec{z}_i \sim g(\vec{z}_i).$$

Without loss of generality, importance sampling can be applied to any expectation, summation or integration. Note that  $g$  can be any probability distribution with the condition that  $\forall \vec{z} \in \Omega_{\vec{z}}$ , if  $p(\vec{z}|\vec{y}) \neq 0$  then  $g(\vec{z}) \neq 0$ . The importance sampling is used in a number of AI algorithms, such as likelihood weighting for Bayesian Networks in which it is infeasible to sample from the posterior probability distribution  $p$  (Shachter and Peot [95], Pradhan and Dagum [91]).

## 2.3 Markov Decision Processes (MDPs)

For decades, the MDP has been an effective mathematical framework for modeling stochastic planning problems. The MDP framework assumes that all the information in the environment *directly* relevant to decision making is captured in the state space — the agent directly observes the state of the environment. Furthermore, the dynamics of the environment is assumed Markovian — the agent makes a series of decisions at discrete time steps as the environment evolves, and the state of the environment at the next time step is independent of the past history given that the agent knows the state at the current time step. Although there are variants of MDPs that enable us to formulate continuous-time decision making, we concentrate on the discrete case since it is more applicable to modeling classical planning problems.

There are other models for stochastic planning problems as well. For example, the Partially Observable Markov Decision Process (POMDP) (Kaelbling *et al.* [60]) assumes that

the decision-making agent does not have direct access to the state. Instead, the agent has access to *observable states* which are produced from the current state. In POMDPs, the state is called *hidden* in the sense that the agent makes noisy or partial observations of the state. The reward and the transitions are still dependent on the state. POMDPs have been used in robot navigation tasks where the sensor returns erroneous data (Cassandra *et al.* [26]), and in medical decision making where the patient shows various symptoms depending on the disease (Hauskrecht [50]). Solving POMDPs is known to be hard, namely undecidable (Madani *et al.* [77]) and PSPACE-complete even in restricted settings (Papadimitriou and Tsitsiklis [88]), thus it is often intractable to solve problems even with small domains. Although a number of algorithms for POMDPs have been proposed (Cassandra *et al.* [27], Hansen [48], Meuleau *et al.* [82]), we focus on MDPs rather than POMDPs since there exist MDP algorithms that run in polynomial time in the size of state space and action space, and classical stochastic planning problems can be seen as MDPs with huge state spaces.

In this section, we provide a precise definition for MDPs and related terms, along with description of the standard algorithms for solving MDPs. The following is the formal definition of MDPs:

**Definition 2.3.1 (MDP)** *An MDP  $M = \{S, A, T, I, R\}$  is defined as*

- *$S$  is the finite set of states.*
- *$A$  is the finite set of actions.*
- *$T$  is the set of transition probabilities:*

$$T(s_t, a, s_{t+1}) = P(s_{t+1}|s_t, a)$$

*where  $s_t$  and  $s_{t+1}$  denote the state of the environment at time  $t$  and  $t+1$ , respectively.*

- *$I$  is the set of initial probabilities:*

$$I(s_0) = P(s_0)$$

*where  $s_0$  denote the state of the environment at time 0.*

- *$R : S \rightarrow \mathfrak{R}$  is the reward function. We can also define the reward to be determined by both state and action ( $R : S \times A \rightarrow \mathfrak{R}$ ) without significant modification.*

The definition of sets  $S$  and  $A$  is subtle. In the most general case, they are Borel subsets of complete separable metric spaces, but in this thesis, we assume that they are finite discrete sets.

	Deterministic	Probabilistic
History dependent	$\Pi^{\text{HD}} \equiv \{\pi d_t : S^* \rightarrow A\}$	$\Pi^{\text{HP}} \equiv \{\pi d_t : S^* \times A \rightarrow [0, 1]\}$
History independent	$\Pi^{\text{MD}} \equiv \{\pi d_t : S \rightarrow A\}$	$\Pi^{\text{MP}} \equiv \{\pi d_t : S \times A \rightarrow [0, 1]\}$

Figure 2.2: MDP policies.  $\pi$  represents an infinite sequence of decision rules for executing an action at each step, so that  $\pi \equiv [d_0, \dots, d_t, \dots]$ .

Traditionally, solving an MDP refers to finding an *optimal policy*. In general, a policy can be regarded as an infinite sequence of decision rules for executing an action. We define four categories of policies depending on whether they are deterministic or probabilistic, and whether history-dependent or history-independent. The four categories are summarized in Figure 2.2. History-dependent policies take the whole series of states observed from time step 0 through the current time step  $t$  as the basis for choosing the action. History-independent policies, also called Markov policies, only take the state observed at the current time step as the basis for choosing the action. Deterministic policies choose only one action for each possible state. Probabilistic policies, also called mixed policies, choose the action drawn from a probability distribution defined for each possible state. Note that history-dependent probabilistic policies ( $\Pi^{\text{HP}}$ ) are the most general class of all four categories.

There are a few models for measuring the quality of a policy. This quality is defined in terms of the expected future rewards. The models can be classified into the finite-horizon model or the infinite-horizon model depending on the life span of the decision-making agent. In the *finite-horizon model*, the quality is determined by the expected rewards for the next  $H$  time steps,

$$V_H^\pi(s) = E \left[ \sum_{t=0}^H \gamma^t R(s_t) | s_0 = s, \pi \right]$$

and we ignore the rewards after the  $H$ -th time step. This model has a natural explanation when the agent has a life of  $H$  time steps. We call  $V_H^\pi(s)$  a  $H$ -step value function of policy  $\pi$ .

In this thesis, the quality of a policy is determined by the *infinite-horizon* value function under the expected total discounted reward,

$$V^\pi(s) \equiv E \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s, \pi \right],$$

where the action is selected according to policy  $\pi$  and the reward is depreciated exponentially by discount factor  $\gamma \in [0, 1)$ . The most straightforward interpretation of  $\gamma$  is as an interest

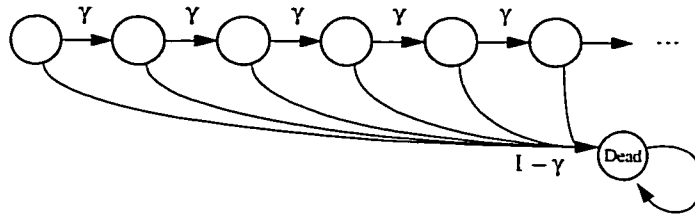


Figure 2.3: The discount rate  $\gamma$  can be seen as the probability of living another time step. Once the agent reaches the “dead” state, it stays there and no reward is given throughout the future.

rate, but it can also be seen as a probability of living another time step as shown in Figure 2.3. It also serves as a mathematical trick to bound the infinite sum. Other value functions under different criteria exist, such as the value function under average reward. A detailed introduction to various value functions can be found in Puterman [93]. These value functions are used to compare policies under different criteria.

Throughout the thesis, we assume infinite-horizon under expected total discounted reward model for measuring the quality of a policy.

Another useful definition related to the value function is the  $Q$ -function,

$$Q^\pi(s, a) \equiv E \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, [a, \pi] \right],$$

which is the expected total discounted reward starting from state  $s$  while we execute action  $a$  in the first time step and follow the policy  $\pi$  afterwards. In the equation above,  $[a, \pi]$  denotes such policy since a policy is represented as a vector of decision rules at time  $t$ . We define an optimal policy  $\pi^*$  as

$$\pi^*(s) \equiv \arg \max_{\pi} V^\pi(s) \text{ for each } s \in S,$$

and the optimal value function  $V^*$  as the value function of the optimal policy  $\pi^*$ , that is,  $V^* \equiv V^{\pi^*}$ . Likewise, the optimal  $Q$ -function is defined as

$$Q^*(s, a) \equiv Q^{\pi^*}(s, a) \text{ for each } s \in S.$$

Generally speaking, in the search for an optimal policy, we should consider every policy in the space of history-dependent probabilistic policies ( $\Pi^{\text{HP}}$ ) as a candidate answer. It is infeasible to do so since these policies take an arbitrarily long series of states as input. However, using the fact that the state dynamics is Markovian and that the reward only depends on the state at the current time step, we can prove that an optimal policy can be found by restricting our search in the space of history-independent probabilistic policies



( $\Pi^{\text{MP}}$ ) (Theorem 5.5.1 of Puterman [93]). Note that although these policies are history-independent, they can be non-stationary, *i.e.*, have different rules for choosing the action at different time steps. If the policy is non-stationary, we have to represent the decision rules  $\pi_t$  at every time step  $t$ , which is again infeasible to do so for arbitrarily long time steps. Fortunately, we can show that there exists a stationary policy which is optimal (Theorem 6.2.7 of Puterman [93]). In other words, there exists an optimal policy  $\pi : S \times A \rightarrow [0, 1]$  such that same  $\pi$  is applied at every time step. The value iteration algorithm we show in this section uses this property. In fact, probabilistic stationary policies have the following properties:

**Definition 2.3.2 (Value Functions and Q-Functions (Howard [56]))** *The value function of policy  $\pi \in \Pi^{\text{MP}}$  satisfies the following equation:*

$$V^\pi(s) = \sum_a \pi(s, a) [R(s, a) + \gamma \sum_{s'} T(s, a, s') V^\pi(s')] \text{ for each } s \in S.$$

*The Q-function of policy  $\pi$  satisfies the following equation:*

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') V^\pi(s') \text{ for each } s \in S, a \in A.$$

We can go further in exploiting the properties of MDPs and prove that we only have to search in the space of *deterministic stationary* policies. In other words, there exists an optimal policy  $\pi : S \rightarrow A$  such that same decision rule  $\pi$  is applied to states at every time step. The value iteration algorithm we show uses this property (Equation 2.4).

The value function can be seen as the basis for a useful heuristic for finding the optimal policy. If we can obtain optimal value function  $V^*$ , the optimal policy  $\pi^*$  is found by the following set of formulas for each  $s \in S$ :

$$\pi^*(s, a) = \begin{cases} 1 & \text{for } a = \arg \max_{a'} [R(s, a') + \gamma \sum_{s'} T(s, a', s') V^*(s')]. \\ 0 & \text{otherwise.} \end{cases}$$

If there is more than one action that satisfies the max in the above equation, we pick one action for each state so that only one action gets all the probability weight.

There are a number of algorithms for calculating the optimal value function. The first is the *value iteration algorithm* (also called successive approximation [93]) shown in Figure 2.4. In summary, the algorithm iteratively improves the policy by using the value function from the previous iteration. Note that step 2 should have been calculating the

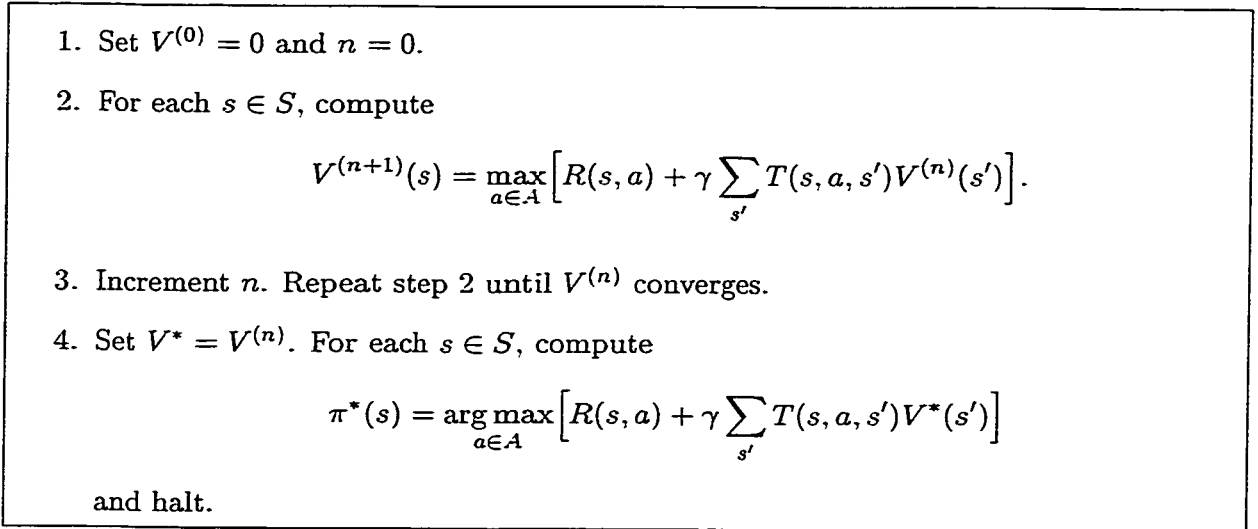


Figure 2.4: Value iteration algorithm.

max over probabilistic policies but the algorithm utilizes the property that for each  $s \in S$ , the inequality

$$\begin{aligned} \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') V^{(n)}(s')] \\ \geq \sum_a \pi(s, a) [R(s, a) + \gamma \sum_{s'} T(s, a, s') V^{(n)}(s')] \end{aligned} \quad (2.4)$$

holds for any  $\pi$  since it is a probability distribution on actions. Thus, we can find the optimal policy in the space of deterministic policies.  $V^{(n)}$  is guaranteed to converge, and the deterministic stationary policy  $\pi^*$  produced by the value iteration algorithm is provably optimal.

The convergence of  $V^{(n)}$  is due to the following *contraction property*. A mapping  $L : \mathfrak{R}^n \rightarrow \mathfrak{R}^n$  is a *contraction mapping* (Browder [24]) if there exists a vector norm  $\| \cdot \|$  and a constant  $0 \leq \alpha < 1$  such that for any  $V, U \in \mathfrak{R}^n$ ,  $\|LV - LU\| \leq \alpha \|V - U\|$ . It is easy to show that step 2 of the value iteration algorithm that maps  $V^{(n)}$  to  $V^{(n+1)}$  is a contraction mapping: Since the value functions are  $|S|$ -dimensional vectors, let

$$LV(s) \equiv \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s')] \quad (2.5)$$

and

$$a_V \equiv \arg \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s')],$$

and let

$$LU(s) \equiv \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') U(s')]$$

and

$$a_U \equiv \arg \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') U(s')].$$

We can prove that  $L$  is a contraction mapping in the infinity norm

$$\|V\|_\infty \equiv \max_s |V(s)|$$

from the fact that when  $LV(s) \geq LU(s)$ ,

$$\begin{aligned} |LV(s) - LU(s)| &= |R(s, a_V) + \gamma \sum_{s'} T(s, a_V, s') V(s') \\ &\quad - R(s, a_U) - \gamma \sum_{s'} T(s, a_U, s') U(s')| \\ &\leq |R(s, a_V) + \gamma \sum_{s'} T(s, a_V, s') V(s') \\ &\quad - R(s, a_V) - \gamma \sum_{s'} T(s, a_V, s') U(s')| \\ &\leq \gamma \max_s |V(s) - U(s)| = \gamma \|V - U\|_\infty. \end{aligned}$$

We can derive the same inequality when  $LV(s) \leq LU(s)$  and thus,

$$\|LV - LU\|_\infty = \max_s |LV(s) - LU(s)| \leq \gamma \|V - U\|_\infty.$$

The second algorithm for calculating the optimal value function is the *policy iteration algorithm*, which is shown in Figure 2.5. The best way to explain this algorithm is to compare it to the value iteration algorithm. Let us assume that we record the policy corresponding to the value function computed in step 2 of the value iteration algorithm at every iteration,

$$\pi_{VI}^{(n+1)}(s) = \arg \max_{a \in \mathcal{A}} [R(s, a) + \gamma \sum_{s'} T(s, a, s') U^{(n)}(s')],$$

where we use  $U^{(n)}$  for  $V^{(n)}$  to avoid confusion with that of the policy iteration algorithm. Note the similarity of the above equation to the policy improvement step in the policy iteration algorithm. The only difference is that in the value iteration algorithm, instead of trying to improve the current policy from its value function  $V^{(n)}$  as in the policy iteration algorithm, we improve the current policy from a crude estimate —  $U^{(n)}$  is not exactly the

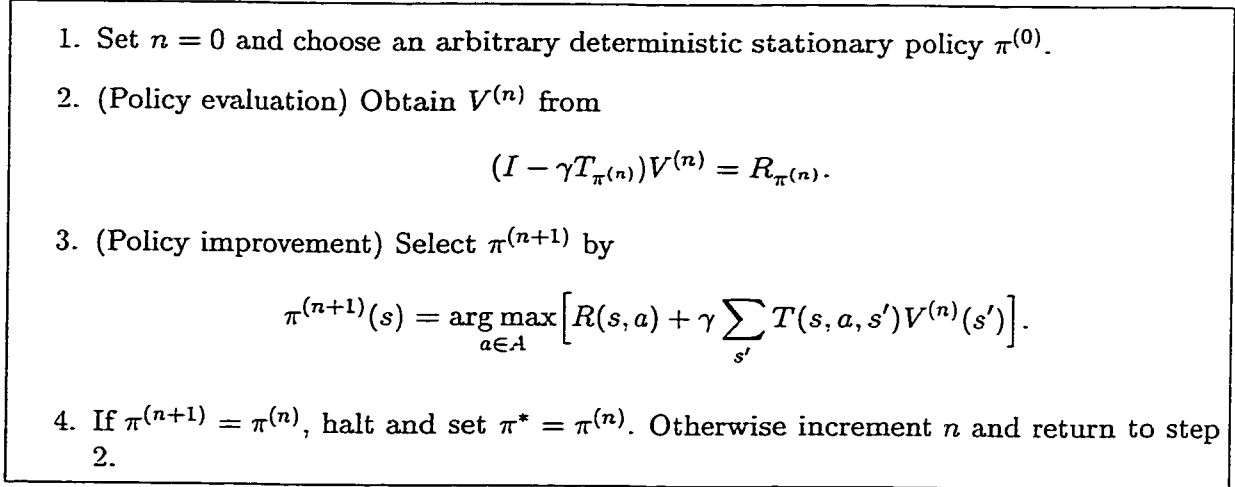


Figure 2.5: Policy iteration algorithm.

value of following policy  $\pi_{VI}^{(n)}$ . It might seem that the policy iteration algorithm converges faster than the value iteration algorithm, but the policy evaluation step is quite expensive, since solving a system of linear equations takes at least  $\Omega(|S|^2)$  operations. Thus, it is not widely used except in special cases, such as approximating value functions with a complex function. Bertsekas and Tsitsiklis [10] provide a detailed discussion with case studies on approximating value functions.

Littman [74] shows that the sequence of  $V^{(n)}$  from the policy iteration algorithm converges at least as fast as that from the value iteration algorithm. Hence, if we can evaluate the policy at almost no cost, the policy iteration algorithm is at least as fast as the value iteration algorithm.

The policy evaluation step is also worthy of special mention. Among a number of methods for solving a system of linear equations, Richardson method (Kincaid and Cheney [69]) provides the simplest form among iterative methods:

$$V^{(i+1)} = R_{\pi^{(n)}} + \gamma T_{\pi^{(n)}} V^{(i)}. \quad (2.6)$$

The convergence analysis of the above iteration is important in the sense that there are a number of methods derived from the Richardson method, such as TD( $\lambda$ ) (Sutton and Barto [103], Bertsekas and Tsitsiklis [10]). It is easy to prove that the equation

$$L_{\pi} V(s) \equiv \sum_a \pi(s, a) \left[ R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s') \right] \quad (2.7)$$

derived from Equation 2.6 is a contraction mapping in the infinity norm. There are other useful norms in which  $L_{\pi}$  is a contraction mapping, and we give two other norms that are

important in the following sections. The first is weighted infinity norm,  $\|\cdot\|_\infty^w$ , which is defined as

$$\|V\|_\infty^w \equiv \max_s V(s)/w(s)$$

for positive weight  $w(s)$  for each state  $s$ . The reader can verify that  $L_\pi$  in Equation 2.7 is a contraction mapping in  $\|\cdot\|_\infty^{V^\pi}$ . The second is the weighted Euclidean norm,  $\|\cdot\|_2^w$ , defined as

$$\|V\|_2^w \equiv \sqrt{\sum_s w(s)V(s)^2}.$$

$L$  in Equation 2.7 is also a contraction mapping in  $\|\cdot\|_2^{P^\pi}$ , where  $P^\pi$  is the stationary distribution following the policy  $\pi$ . The weighted Euclidean norm is useful in the sense that it is a continuous measure in  $\mathfrak{R}^{|S|}$ . Typical value function approximation strategies use gradient descent methods to approximate the target function to the real value function. An analogy can be found in training artificial neural networks for classification tasks, in which we use continuous sigmoid functions as output functions rather than discontinuous step functions.

*Linear programming* (Hillier and Lieberman [52]) is another method for solving MDPs. Every MDP can be converted to a linear program although the converse is an open problem (Littman *et al.* [75]). There are efficient algorithms for solving linear programs in polynomial time (Karmarkar [62], Padberg [87]). The *primal* problem for calculating the optimal value function  $V^*$  is to minimize

$$\sum_s V^*(s)I(s)$$

subject to

$$V^*(s) \geq R(s, a) + \gamma \sum_{s'} T(s, a, s')V^*(s') \text{ for each } s \in S, a \in A.$$

Thus, we have  $|S|$  variables and  $|S| \times |A|$  constraints. Constraints which have equality when the optimal solution is plugged in are called *binding constraints*. If there are a lot of constraints with few binding constraints, LP algorithms take a longer time to identify the binding constraints. As we have seen, MDPs have many more constraints than variables, and as such people consider the *dual* problem which gives a better running time in general. The dual problem of the MDP is to maximize

$$\sum_{s, a} R(s, a)x(s, a)$$

subject to

$$\sum_a x(s, a) - \gamma \sum_{s', a} T(s', a, s) x(s', a) = I(s) \text{ for each } s \in S.$$

Note that we now have  $|S| \times |A|$  variables and  $|S|$  constraints. Another advantage of solving the dual problem is that the variable  $x(s, a)$  yields useful information about the optimal policy. Assume that we have an arbitrary probabilistic history-independent policy  $\pi \in \Pi^{\text{MP}}$  and define

$$x_\pi(s, a) \equiv \sum_{s'} I(s') \sum_t \gamma^t P_\pi(S_t = s, A_t = a | S_0 = s')$$

where  $S_t$  and  $A_t$  are random variables denoting the state and action at time  $t$ .  $x_\pi(s, a)$  is the sum of discounted probabilities of the system being in state  $s$  and executing action  $a$  while following policy  $\pi$ . For any feasible solution  $x(s, a)$  to the dual problem, the probabilistic history-independent policy that corresponds to the solution is found by

$$\pi_x(s, a) = \frac{x(s, a)}{\sum_a x(s, a)}.$$

## 2.4 Stable Value Function Approximation (VFA)

In this section, we survey VFA algorithms that are primarily developed for MDPs with large state spaces. The basic idea is to search for a class of functions that approximate the value function with reasonable quality and has tractable representation size. Although in principle we can deploy any approximation function, for example, artificial neural networks, the focus in VFA research has been on finding the approximation algorithm and the class of functions so that they guarantee the bounded error with respect to the true value function. Such an algorithm is called a *stable VFA algorithm*.

It is known that using any non-linear function with a naive MDP algorithm such as the value iteration algorithm can produce arbitrarily bad results (Baird [4], Gordon [47], Tsitsiklis and Van Roy [105]). Gordon [47] examines several classes of approximation functions and focuses on using approximations in the value iteration algorithm. A key result is the definition of *averagers*, a class of approximation functions that are *stable* in the sense that they satisfy the following requirements: Let  $V^{(n)}$  denote the value function calculated from the previous iteration (Figure 2.4). Also assume that the approximation is used with the value iteration algorithm in a way that, at each iteration, we first approximate the value function from the previous iteration as  $\hat{V}^{(n)}$  and calculate the value function at the next

iteration by

$$V^{(n+1)}(s) = \max_{a \in A} \left[ R(s, a) + \gamma \sum_{s'} T(s, a, s') \hat{V}^{(n)}(s') \right]. \quad (2.8)$$

The approximation is stable if the  $V^{(n)}$  converges and if we can bound the distance between the convergence point and the true optimal value function. More formally, an averager is defined as follows:

**Definition 2.4.1 (Averager)** *An approximation function is an averager if, given the target vector  $U$ , the approximation function generates  $\hat{U}$  defined as*

$$\hat{U}(s) \equiv \beta_s c_s + \sum_{s'} \beta_{s,s'} U(s') \quad (2.9)$$

where  $c_s$  is a constant and  $\beta_s$  and  $\beta_{s,s'}$  is a non-negative constant such that  $\forall s, \beta_s + \sum_{s'} \beta_{s,s'} = 1$ .

Let mapping  $A$  denote an averager so that the above equation becomes  $\hat{U} = AU$ . Using an averager with the value iteration algorithm can be viewed as the composite mapping  $L \circ A$ , where  $L$  represents the mapping defined as the each iteration of the value iteration algorithm. As an example, if we use averager  $A$  for obtaining approximation  $\hat{V}^{(n)}$  from  $V^{(n)}$  so that  $\hat{V}^{(n)} = AV^{(n)}$ , Equation 2.8 becomes

$$\begin{aligned} V^{(n+1)}(s) &= \max_{a \in A} \left[ R(s, a) + \gamma \sum_{s'} T(s, a, s') \hat{V}^{(n)}(s') \right] \\ &= \max_{a \in A} \left[ R(s, a) + \gamma \sum_{s'} T(s, a, s') (AV^{(n)})(s') \right] \\ &= ((L \circ A)V^{(n)})(s). \end{aligned}$$

Thus,  $V^{(n+1)} = (L \circ A)V^{(n)}$ . It is easy to show that  $L \circ A$  is a contraction mapping in the infinity norm. Thus, using the averager with value iteration algorithm converges to a fixed point. Furthermore, we can establish the error bound between the fixed point and the optimal value function from the following theorem:

**Theorem 2.4.1 (Gordon)** *Let  $V^*$  be the optimal value function for an MDP  $M$  and let  $A$  be the mapping for an averager. Let  $V^A$  be the fixed point of  $A$  closest to  $V^*$ . Given  $\|V^A - V^*\|_\infty = \epsilon$ , the iteration of  $L \circ A$  converges to a value function  $V^{L \circ A}$  so that  $\|V^{L \circ A} - V^*\| \leq 2\gamma\epsilon/(1 - \gamma)$ . The approximate value iteration returns  $AV^{L \circ A}$  which satisfies  $\|AV^{L \circ A} - V^*\| \leq 2\epsilon + 2\gamma\epsilon/(1 - \gamma)$ .*

Note that above theorem expresses the error bound in terms of the distance between the *closest* fixed point of  $A$  and the optimal value function. The proof of the above theorem uses the fact that  $A$  is a *non-expansion mapping* in the infinity norm so that  $\|AV - AU\|_\infty \leq \|V - U\|_\infty$ . Thus,  $L \circ A$  is a contraction mapping and it is guaranteed to converge to a fixed point. Averagers include weighted  $k$ -nearest neighbor, linear regression, and various types of meshes. Note that the parameters of the averager are fixed throughout the value iteration algorithm to achieve stable approximation.

Koller and Parr [70] use weighted Euclidean norm in an attempt to optimize the parameters of the averager. Assume that we have a fixed policy  $\pi$  and we want to calculate the approximate value function of  $\pi$ . To find the best parameters of the averager, we need to calculate the stationary distribution  $P^\pi$  and find the best set of parameters that minimizes the error.

$$L_\pi V(s) \equiv \pi(s, a)[R(s, a) + \gamma \sum_{s'} T(s, a, s')V(s')]$$

and

$$AV(s) \equiv \alpha_1 h_1(s) + \dots + \alpha_k h_k(s)$$

where  $h_1, \dots, h_k$  are basis functions that take certain features of the state and map to real values, and  $\alpha_1, \dots, \alpha_k$  are the parameters of the approximation function. Note that this approximation is a special case of averager, by letting  $c_s \equiv \alpha_1 h_1(s) + \dots + \alpha_k h_k(s)$ ,  $\beta_s = 1$ , and  $\beta_{s,s'} = 0$  in Equation 2.9. Approximating  $V$  is done by calculating

$$\hat{V} \equiv \min_{\alpha_1, \dots, \alpha_k} \|AV - V\|_2^{P^\pi},$$

which minimizes the distance (error) between  $AV$  and  $V$  in  $\|\cdot\|_2^{P^\pi}$ . A solution to the above equation can be obtained by weighted projection of  $V$  with respect to  $P^\pi$ . Let mapping  $\Pi^{P^\pi}$  be the projection so that  $\hat{V} = \Pi^{P^\pi} V$ . It can be shown that  $\Pi^{P^\pi}$  is a non-expansion mapping and  $L_\pi$  is a contraction mapping in  $\|\cdot\|_2^{P^\pi}$ . Thus, the approximate value function evaluation defined as repeatedly applying  $\Pi^{P^\pi} \circ L_\pi$  is a contraction mapping, and it is guaranteed to converge to a fixed point. Koller and Parr later extend this result and develop an approximate policy iteration algorithm (Koller and Parr [71]).

## 2.5 Stochastic Bisimulation Equivalence in MDPs

In practice, the state space of an MDP is quite large. This prevents us from using standard algorithms such as value iteration described in Section 2.3. In some cases, we can reduce the



state space of an MDP by excluding unreachable states and by grouping those states that behave the same. In this section, we define the notion of *stochastic bisimulation equivalence* for partitioning the state space into blocks such that every state in a block behaves the same. A partition of state space  $S$  is a set of blocks  $\{B_1, \dots, B_n\}$  in which each block is a mutually disjoint set with other blocks and the union of the blocks is the whole state space  $S$ .

Stochastic bisimulation equivalence is closely related to the state equivalence relation for minimizing Finite State Automata (FSA) (Hopcroft and Ullman [54]). As shown in Section 2.1, Markov chains with finite state spaces can be represented as graphs with probabilistic transitions between vertices. Such a view is closely related to Probabilistic Finite State Automata (PFSA), which are finite state automata with probabilistic transitions. MDPs can also be regarded as PFSA with inputs being the actions and the outputs being the rewards. We present the characteristics of a block in the partition induced by stochastic bisimulation equivalence:

**Definition 2.5.1 (Stable Block and Homogeneous Partition for MDPs)** *A block  $C$  of a partition  $P$  is called stable with respect to a block  $B$  of  $P$  and action  $a$  if and only if every state in  $C$  has the same transition probability of going into block  $B$  by action  $a$ . Mathematically,*

$$\exists c \in [0, 1] \text{ such that } \forall s_t \in C, T(s_t, a, B) = c$$

where

$$T(s_t, a, B) = \sum_{s_{t+1} \in B} T(s_t, a, s_{t+1}).$$

*We say that  $C$  is stable if  $C$  is stable with respect to every block of  $P$  and action in  $A$ .  $P$  is called homogeneous if and only if every block is stable.*

Two states are stochastic bisimulation equivalent if their rewards are the same and they belong to the same block of a homogeneous partition. Thus, given a homogeneous partition  $P = \{B_1, \dots, B_n\}$  of the state space, for each  $B_i, B_j \in P$  and  $a \in A$  and  $s, s' \in B_i$ ,

$$\sum_{s'' \in B_j} T(s, a, s'') = \sum_{s'' \in B_j} T(s', a, s''),$$

which follows from

$$T(s, a, B_j) = T(s', a, B_j).$$

Since every state in a block of a homogeneous partition has the same transition probability of going into any block, we define the transition probability between blocks of a homogeneous partition as follows:

$$T(B_i, a, B_j) = T(s, a, B_j) \text{ for any } s \in B_i.$$

Once we have a homogeneous partition and the transition probabilities between blocks, we can construct an *aggregate* MDP in which we regard each block as a state. Note that the number of blocks in such an aggregate MDP is at most as many as the number of states in the original MDP. As such, the amount of time required to solve an aggregate MDP is at most as much as that required to solve the original MDP. For example, if we use the value iteration algorithm, the time complexity is  $O(|S|^3)$ . The cost of solving the aggregate MDP is minimized when the number of blocks of the partition is minimized. We say that the aggregate MDP constructed from a homogeneous partition is a *minimal model* of the original MDP if the number of blocks in the partition is minimum. On the other hand, a homogeneous partition which induces the aggregate MDP with the minimum number of states is called the *coarsest homogeneous partition*. An aggregate MDP is a minimal model if it is constructed from the coarsest homogeneous partition. Hence, we are interested in finding the coarsest homogeneous partition of the original MDP. Indeed, we can design an algorithm that is similar to the minimization algorithm for finite state automata (Hopcroft and Ullman [54]).

Before presenting the algorithm, we give a recursive definition for a pair of states *not* belonging to the same block in a homogeneous partition. Definition 2.5.1 provides the condition for a pair of states belong to the same block, but it is not intuitive to implement. The classical minimization algorithm for FSA also uses the notion of a pair of states that does not belong to the same aggregate state in the minimized FSA, defined as “distinguishable.” We present an iterative algorithm that, at every iteration, generates new pairs of states that are distinguishable. First, note that every pair of states that has different rewards is distinguishable:

State  $s$  is *distinguishable* from state  $s'$  if

$$R(s) \neq R(s')$$

Second, at any given iteration in the algorithm, there will be some pairs of states that have been determined to be distinguishable and other pairs that have not as yet been determined to be distinguishable. For the latter, they become distinguishable if the following definition is satisfied:

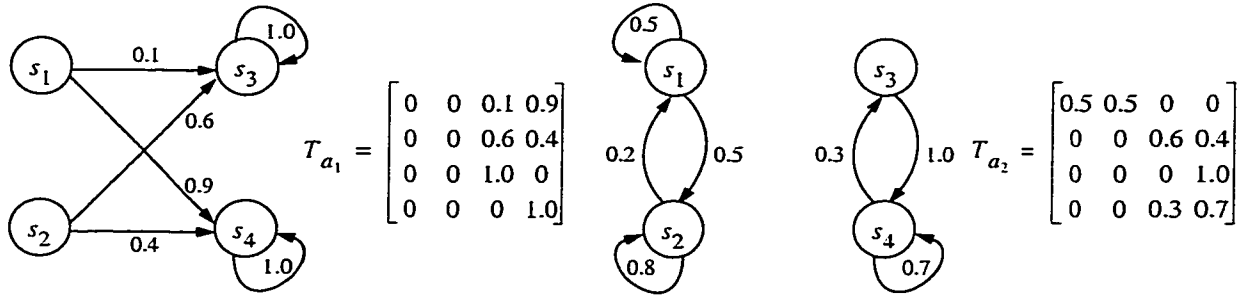


Figure 2.6: An MDP subject to minimization. The reward function is given as  $R(s_1) = R(s_2) = 0$  and  $R(s_3) = R(s_4) = 1$ .

State  $s$  is *distinguishable* from state  $s'$  if there exists action  $a$  and state  $s''$  such that

$$T(s, a, B_{s''}) \neq T(s', a, B_{s''}),$$

where  $B_{s''}$  is the set of all states that are not yet distinguishable from  $s''$ .

Given the above definition of distinguishable, we show how the algorithm works in a particular example. Let  $M$  be the MDP of Figure 2.6 and Define  $\leftrightarrow$  to be the relation distinguishable. We first note that  $s_1 \leftrightarrow s_3$ ,  $s_1 \leftrightarrow s_4$ ,  $s_2 \leftrightarrow s_3$ ,  $s_2 \leftrightarrow s_4$  from the reward function.

Next, based on what we currently have, we determine that  $s_1$  and  $s_2$  are not distinguishable by calculating

$$\begin{aligned} T(s_1, a_1, s_3) + T(s_1, a_1, s_4) &= 1.0 \\ T(s_2, a_1, s_3) + T(s_2, a_1, s_4) &= 1.0 \\ T(s_1, a_2, s_3) + T(s_1, a_2, s_4) &= 0 \\ T(s_2, a_2, s_3) + T(s_2, a_2, s_4) &= 0 \\ T(s_1, a_1, s_1) + T(s_1, a_1, s_2) &= 0 \\ T(s_2, a_1, s_1) + T(s_2, a_1, s_2) &= 0 \\ T(s_1, a_2, s_1) + T(s_1, a_2, s_2) &= 1.0 \\ T(s_2, a_2, s_1) + T(s_2, a_2, s_2) &= 1.0, \end{aligned}$$

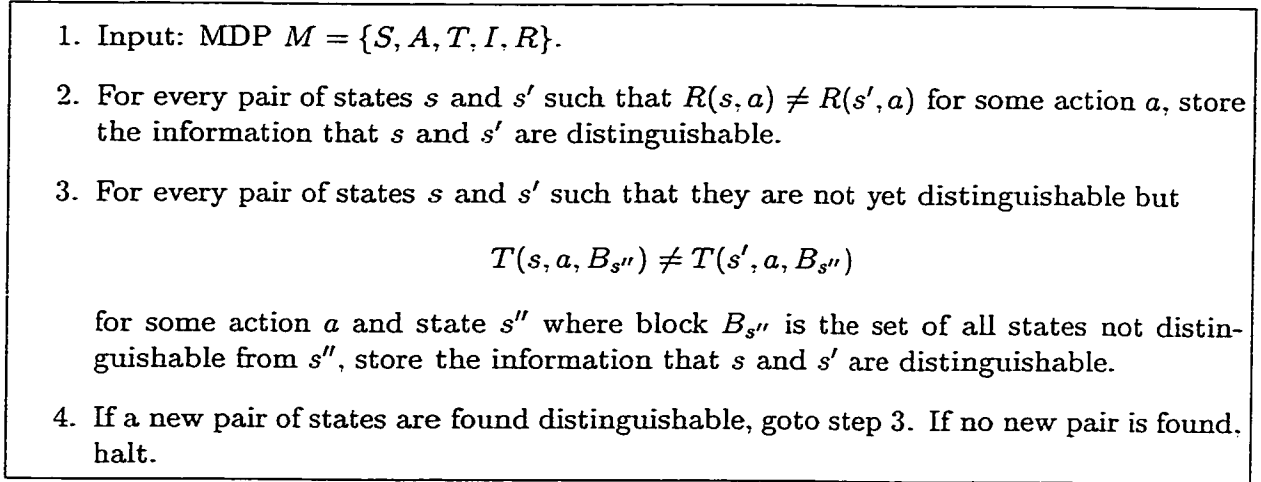


Figure 2.7: MDP minimization algorithm

and that  $s_3$  and  $s_4$  are not distinguishable by calculating

$$T(s_3, a_1, s_3) + T(s_3, a_1, s_4) = 1.0$$

$$T(s_4, a_1, s_3) + T(s_4, a_1, s_4) = 1.0$$

$$T(s_3, a_2, s_3) + T(s_3, a_2, s_4) = 1.0$$

$$T(s_4, a_2, s_3) + T(s_4, a_2, s_4) = 1.0$$

$$T(s_3, a_1, s_1) + T(s_3, a_1, s_2) = 0$$

$$T(s_4, a_1, s_1) + T(s_4, a_1, s_2) = 0$$

$$T(s_3, a_2, s_1) + T(s_3, a_2, s_2) = 0$$

$$T(s_4, a_2, s_1) + T(s_4, a_2, s_2) = 0.$$

Since we did not discover any new pair of distinguishable states, we determine that we obtained the coarsest homogeneous partition of two blocks  $\{\{s_1, s_2\}, \{s_3, s_4\}\}$ .

We show an MDP minimization algorithm in Figure 2.7. The algorithm identifies new pairs of distinguishable states at each iteration. This iterative procedure is equivalent to *refining* an initial partition (partition induced by reward). In Section 4.1, we extend the algorithm to handle individual states without explicitly enumerating all the states when the MDP is presented as an FMDP, a compact representation which we define in the next chapter.

## Chapter 3

# Factored MDPs (FMDPs)

### 3.1 Representation and Complexity

MDPs, as defined in Section 2.3, have been well studied. In terms of computational complexity, solving MDPs is known to be in the class P-complete, so we know that there exists a polynomial time algorithm that outputs the optimal policy. However, the classical representation of an MDP stores the probabilities and the rewards explicitly, enumerating all possible states and actions. It is often infeasible to do so in real world problems, where we are confronted with a large number of states. As such, people have been looking into factored models to achieve economy in representation. Most of the examples are tailored to specific problem domains. In this section, following the definition of Boutilier *et al.* [16], we present Factored MDP (FMDP), which is widely used as a general purpose factored representation for an MDP with a discrete,  $n$ -dimensional state space.

**Definition 3.1.1 (FMDP)** An FMDP  $M = \{\vec{X}, A, T, I, R\}$  is defined as

- $\vec{X} = [X_1, \dots, X_n]$  is the set of fluents that are used to describe the individual states. An assignment of values to each and every fluent defines the state. We use  $\Omega_{X_i}$  to denote the set of values that fluent  $X_i$  can take —  $\Omega_{X_i}$  is the sample space of  $X_i$  when  $X_i$  is treated as a random variable. Thus, the state space is  $\Omega_{\vec{X}} = \prod_i \Omega_{X_i}$ . We use the lowercase letter  $\vec{x} = [x_1, \dots, x_n]$  to denote a particular instantiation of the fluents.
- $A$  is the set of actions.
- $T$  is the set of transition probabilities, represented as the set of conditional probability

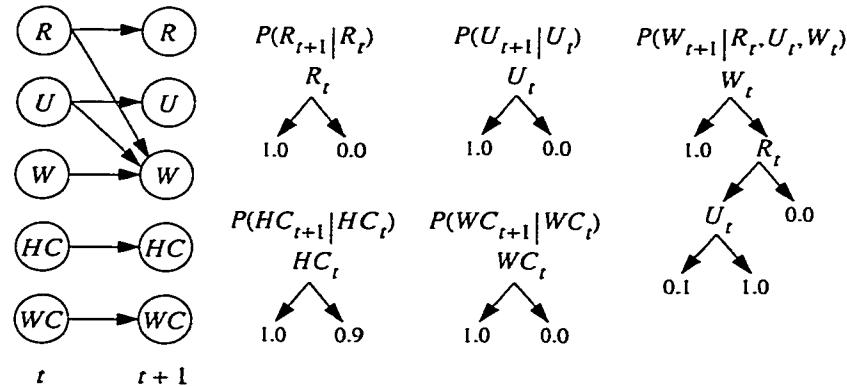


Figure 3.1: Graphical representation of an FMDP describing a toy robot domain.

distributions, one for each action and fluent:

$$T(\vec{x}_t, a, \vec{x}_{t+1}) = \prod_{i=1}^n P(x_{i,t+1} | \text{pa}(x_{i,t+1}), a) \quad (3.1)$$

where  $\text{pa}(X_{i,t+1})$  denotes the set of fluents that directly influence the value of  $X_{i,t+1}$ . Note that  $\forall i, \text{pa}(X_{i,t+1}) \subseteq \{X_{1,t}, \dots, X_{n,t}\}$ .

- $I$  is the set of initial probabilities:

$$I(\vec{x}_0) = P(\vec{x}_0)$$

where  $\vec{x}_0$  denotes the state of the environment at time 0.

- $R : \vec{X} \rightarrow \mathfrak{R}$  is the reward function. Without loss of generality, we define the reward to be determined by both state and action ( $R : \vec{X} \times A \rightarrow \mathfrak{R}$ ).

The FMDP representation is said to be a factored representation mainly due to Equation 3.1. The transition probabilities are factored into a product of probabilities of individual fluents conditioned on subsets of the fluents at the previous time step.

A common way of representing an FMDP is using a *graphical model*. Figure 3.1 shows the graphical model representing the dynamics of an FMDP example in which we execute an action in a domain with 5 boolean variables. The toy robot has to deliver a cup of coffee to the master whenever wanted. Unfortunately, the coffee bar is outside the building and the robot receives a small amount of punishment if it gets wet. Each fluent denotes a particular aspect of the world — the weather outside being rainy ( $R$ ), the robot having an umbrella ( $U$ ), the robot being wet ( $W$ ), the robot holding coffee ( $HC$ ), and the robot's master wanting coffee ( $WC$ ). Note that the probabilistic effect of each fluent at time  $t + 1$

is conditioned on, or in other words influenced by, a small number of fluents at time  $t$ . We call them parent fluents of a given fluent, and use the notation  $\text{pa}(X_{t+1})$  for the parents of fluent  $X_{t+1}$ . For example, the set of parent fluents for  $W_{t+1}$  for the above example is  $\text{pa}(W_{t+1}) = \{R_t, U_t, W_t\}$ .

Note that if we use tables for the conditional probabilities, the size of a table is exponential in the number of parent fluents. In the FMDPs we consider, the conditional probabilities are stored as decision trees, which are called conditional probability trees (CPTs) (Boutilier *et al.* [16]), rather than tables for the sake of savings in the size of the representation. CPTs exploit the regularity in the conditional probabilities, namely, some of the parent fluents become independent of the child fluent given a partial set of assignments for the parent fluents. We complete the description of the state dynamics by specifying CPTs for each fluent and action. The reward function  $R$  and initial probabilities  $I$  are also designated in terms of decision trees. In a detailed discussion on using decision trees for conditional probabilities, Boutilier *et al.* [17] show how CPTs can be used to exploit independence, which they call *context-specific independence*, in Bayesian networks.

Similar representations have been used to model Markovian processes with factored transition probabilities, such as the Two-stage Temporal Bayesian Network (2TBN) (Dean and Kanazawa [35]) and the Dynamic Bayesian Network (DBN) (Forbes *et al.* [41]).

As we can see, FMDPs exploit the conditional independence among the fluents given their parent fluents to achieve a compact, factored representation. Such a representation allows us to describe the domain without explicitly enumerating all the states and transition probabilities. However, a compact description of the domain does not readily lead to an efficient FMDP algorithm. There are two main difficulties in solving FMDPs. First, we can no longer directly use simple dynamic programming algorithms such as value iteration or policy iteration. These algorithms require calculating value functions, and when represented as vectors, the size of the representation for these functions can explode, since the size of the state space is exponential in the number of fluents. There are algorithms proposed that use alternative representations for value functions to compress the size and preclude explicit enumeration. Boutilier and Dearden [15] use decision trees for compactly representing the value functions and use pruning techniques to sacrifice the precision of the value function in the favor of truncating the size of the decision tree. Hoey *et al.* [53] use Algebraic Decision Diagrams (ADDs) that are more compact than decision trees and present pruning technique on ADDs. These algorithms are examples of Value Function Approximation (VFA) algorithms applied to FMDPs. Second, the optimal *stationary* policy, which most of the current algorithms produce as output, may require exponentially larger space compared

to that of the description of the FMDP. To achieve optimal performance, it may be necessary that the policy be complex enough to handle all the different aspects of state space. This problem is more serious than the first since it states that although there may exist an algorithm that does not calculate the value function at all, such an algorithm has to output an answer which is exponential in the size of the input. Some VFA algorithms lead to small *approximately* optimal policies (Koller and Parr [71]). In our research, we have shown that there exist FMDPs for which the history-dependent optimal policy is small whereas the history-independent optimal policy is huge. Thus, it may be more fruitful to search for the optimal policy in the space of history-dependent policies (Kim *et al.* [68]).

The bounds on the computational complexity of FMDP problems can be obtained from the complexity results for probabilistic STRIPS planning problems. Littman *et al.* [76] provide the computational complexity results for various classes of plan types for stochastic planning problems, and these plans are closely related to policies for FMDPs. Note that if we define a stationary policy for an FMDP in the same way as that for an MDP, we can end up with a table of size  $(\prod_i |\Omega_{X_i}|) \times |A|$ , which is unacceptable. Instead, researchers have been using structured representations for policies. One of the commonly used representations is the decision tree — the query nodes ask the value of each fluent and the leaf nodes store the action to execute.

The size of the decision tree for the optimal policy can be very large — exponential in the size of the problem. As such, it is necessary that we look for polynomial-size decision trees. In the remainder of this section, we present complexity results for solving FMDPs with decision tree policies. The goal of the theorems below is to show that solving FMDPs with decision tree policies is as hard as solving FMDPs with a class of policies conventionally known as *looping plans*. A looping plan is basically a history-dependent policy that can be represented as a Finite State Automaton (FSA), in which nodes store the action to execute and transitions are determined by some features of the states (See Figure 3.2. We present a formal definition of looping plans with probabilistic transitions in Section 3.2). Thus, although we conventionally search for an optimal policy in the space of decision trees, we claim that it is sometimes more fruitful to search in the space of looping plans. We show a convincing example in Section 3.2, and an algorithm for searching in the space of looping plans (Section 4.4).

The complexity result we show below uses a proof technique used in Littman *et al.* [76]. First, we formulate the problems as follows:

- **Policy Evaluation** : Given an FMDP  $M$ , a decision tree policy  $\pi$  of description size



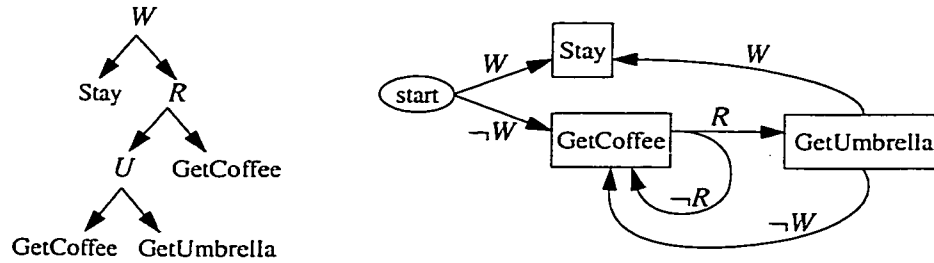


Figure 3.2: Comparing a decision tree policy (left) with a looping plan (right). In a decision tree policy, the action is found by following down the path from the root to the leaf determined by the current state. In a looping plan, the action is determined by following the outgoing edge determined by the current state. We can convert a decision tree policy to a looping plan by preparing node for each terminal node of the decision tree and labeling incoming edges with the path formula of the terminal node. Hence, the conversion can be done without the explosion in representation size, but not vice versa. The two examples in the figure are not equivalent.

$|\pi| \leq |M|$ , and value threshold  $v$ , determine whether the value of the policy is greater than  $v$ .

- **Policy Existence** : Given an FMDP  $M$ , size bound  $k$ , and value threshold  $v$ , determine whether there exists a decision tree policy  $\pi$  of size  $k$  with value greater than  $v$ .

We show that above decision tree policy problems are PSPACE-complete, which means that they are as hard as policy evaluation problem and policy existence problem for looping plans. We use the proof similar to Littman *et al.* [76] for the following theorem:

**Theorem 3.1.1** *The policy evaluation problem for decision tree policies is PSPACE-complete.*

**Proof** First, we prove containment in PSPACE. For an MDP with  $N$  states, evaluating the policy, *i.e.*, calculating the value function of a stationary policy, is in PL (probabilistic log-space) (Allender and Ogihara [1]). Thus, the policy evaluation of an FMDP with  $c^N$  states can be done in probabilistic space polynomial in the size of the input. Since probabilistic PSPACE is equal to PSPACE, it is clear that the problem is in PSPACE.

Second, to prove that it is PSPACE-hard, we show that we can convert a deterministic polynomial-space bounded Turing machine PTM to an FMDP  $M$  such that the evaluation of a policy  $\pi$  in  $M$  implies that PTM accepts an input  $x$ . The instantaneous description (ID) of PTM can be expressed as a polynomial-length bit string that encodes the contents

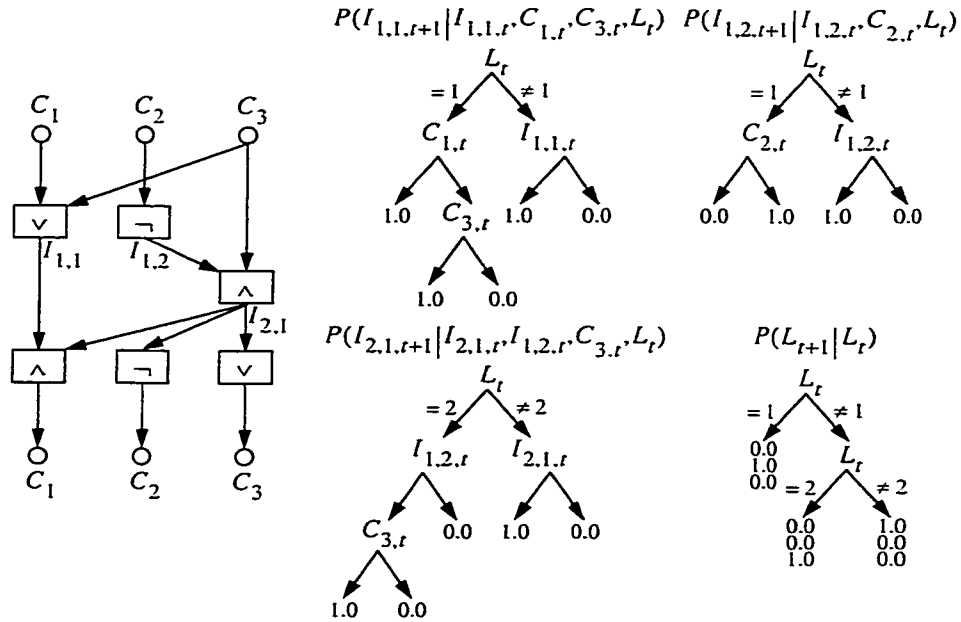


Figure 3.3: Constructing the FMDP from a circuit.

of PTM's tape, the location of read/write head, the state of PTM's controller, and whether or not the machine is in a final state.

The (deterministic) FMDP  $M$  has a single action “compute”. Since there is only one action, there is only one decision tree policy  $\pi$  which executes “compute” at each time step regardless of the state. We construct  $M$  in such a way that, given the ID of a PTM,  $M$  yields  $v_\pi > 0$  if and only if  $x$  is accepted by the PTM.

Given the ID of a PTM and  $x$ , we can construct a Turing machine TM that computes the move (one step transition) for the PTM — TM takes the ID of PTM as input and generates the next ID of PTM as output. The construction of TM takes polynomial time in the size of the ID of the PTM. Note that TM can be modeled as a polynomial sized circuit by a similar argument to Cook's theorem. Thus, this circuit takes the ID of the PTM as input and the next ID of the PTM as output.

We now construct  $M$  based on the circuit. Executing action “compute” a polynomial number of times results in computing one move of the PTM. Figure 3.3 shows how we construct  $M$  from the circuit.  $C_i$  represents the  $i$ -th bit from the encoding of the ID of the PTM.  $I_{j,k}$  represents the  $k$ -th gate in layer  $j$  of the circuit.  $M$  is composed of fluents corresponding to each  $C_i$  and  $I_{j,k}$  plus a special fluent  $L$ .  $L$  acts as a “clock” such that if  $L = l$ , only the gates in the layer  $l$  compute their outputs. Although the CPT for every gate is not shown in the figure, we can construct CPTs for AND, OR and NOT gates from

the examples of CPT for  $I_{j,k}$ . The reward function is chosen so that the FMDP states corresponding to accepting states of the PTM have reward of 1, and every other state has a reward of 0. The initial state distribution is deterministically set to the initial ID of the PTM.

In summary, we have shown that  $v_\pi > 0$  if and only if a deterministic polynomial-space bounded Turing machine PTM accepts input  $x$ .  $\square$

The original proof by Littman *et al.* does not involve the “clock” fluent since, in their planning model, a fluent can be dependent on a fluent at the same time step. In our definition of FMDP, we assume that this does not happen — there can be no arcs between fluents at the same time step. Note that, as we have seen in the above theorem, this restriction does not make the problem significantly harder. The proof of the next theorem concerning the policy evaluation problem is adopted from Littman *et al.* without modification.

**Theorem 3.1.2** *The policy existence problem for decision tree policies is PSPACE-complete.*

**Proof** To show that it is in PSPACE, we note that a decision tree policy can be generated in polynomial time and evaluated in PSPACE, the problem is in  $\text{NP}^{\text{PSPACE}} = \text{PSPACE}$ .

For the proof of the problem being PSPACE-hard, we use the FMDP in the proof of Theorem 3.1.1. Since there is only one action in the domain, there exists only one decision tree policy, which we define as  $\pi$ . Deciding whether  $v_\pi > 0$  is PSPACE-complete, thus the original problem is at least PSPACE-hard.

Hence, the policy existence problem for decision tree policies is PSPACE-complete.  $\square$

The following theorems state that the problems regarding *probabilistic decision tree policies* are also PSPACE-complete. A probabilistic decision tree policy has probabilities on actions in the leaves. Since the algorithm in Section 4.4 generates probabilistic versions of looping plans (a precise definition will follow in Section 3.2), it is also interesting to see how computationally hard it is to solve FMDPs with probabilistic decision tree policies.

**Theorem 3.1.3** *The policy evaluation problem for  $b$ -bit precision probabilistic decision tree policies is PSPACE-complete.*

**Proof** The problem is contained in the class PSPACE by the same argument used in Theorem 3.1.1. It is PSPACE-hard since a deterministic policy is a special case of probabilistic policies. Thus, the problem is PSPACE-complete.  $\square$

**Theorem 3.1.4** *The policy existence problem for  $b$ -bit precision probabilistic decision tree policies is PSPACE-complete.*

**Proof** Since the  $b$ -bit precision probabilistic decision tree policy can be generated in polynomial time and evaluated in PSPACE, the problem is within PSPACE. By the same argument in Theorem 3.1.2, the problem is PSPACE-hard. Thus, the problem is PSPACE-complete.  $\square$

By the same argument, we can also prove the following theorems. A probabilistic looping plan is a probabilistic version of a looping plan with probability distribution on actions and transitions.

**Theorem 3.1.5** *The policy evaluation problem for  $b$ -bit precision probabilistic looping plans is PSPACE-complete.*

**Theorem 3.1.6** *The policy existence problem for  $b$ -bit precision probabilistic looping plans is PSPACE-complete.*

## 3.2 History Dependent Policies

In general, history-dependent policies require an infinite amount of memory since decision making at time  $t$  depends on all the observations from time 0 to time  $t$ . Since it is not possible to store an unbounded amount of memory in a digital computer, we take a closer look at history dependent policies with finite amount of memory.

The probabilistic looping plan which we mentioned in the previous section is one such example. In general, a probabilistic looping plan can be thought of as a Probabilistic Finite State Automaton (PFSA), where there are a number states in the policy, the transition between the states depends on certain values of fluents, and the action to be executed depends on the states in the policy. We call it a PFSA policy, whose precise definition is as follows:

**Definition 3.2.1 (PFSA policy)** *A PFSA  $\pi$  on FMDP  $M = \{\bar{X}, A, T, I, R\}$  is represented by  $(\mathcal{V}, \mathcal{E}, \mathcal{F}, \Psi, \Delta_I, \Delta_T)$  such that*

- $\mathcal{V}$  is the set of vertices in the PFSA.
- $\mathcal{E}$  is the set of directed edges in the PFSA.

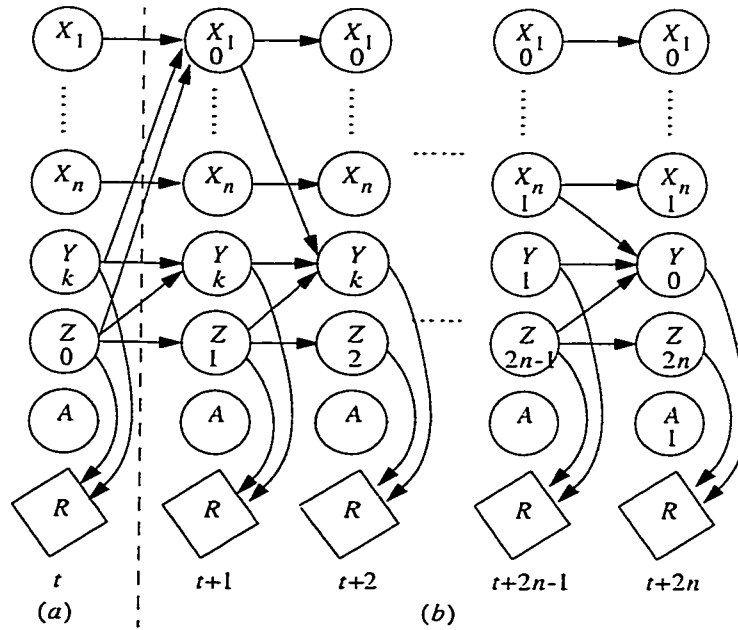


Figure 3.4: A task determining majority represented as a FMDP. (a) answer generation phase : In this example,  $Y$  is sampled to be  $k$  at time  $t$ . (b) problem instance generation phase : Then, exactly  $k$  fluents among  $X_1, \dots, X_n$  are set to 1 by the time  $t + 2n$ . In this example,  $X_1$  is sampled to be 0 and  $X_n$  to be 1.

- $\mathcal{F} = \{f_{v,k}(\vec{x}) | v \in \mathcal{V}, k = 1, \dots, K_v\}$  is the set of boolean functions of domain fluents so that for each vertex  $v$ ,  $f_{v,k}(\vec{x}) \wedge f_{v,k'}(\vec{x})$  is a contradiction if  $k \neq k'$  and  $\bigvee_k f_{v,k}(\vec{x})$  is a tautology.  $K_v$  is the number of outgoing edges for vertex  $v$ .
- $\Psi = \{\psi(v, a) | v \in \mathcal{V}, a \in A\}$  such that  $\psi(v, a)$  is the probability of choosing action  $a$  in vertex  $v$ .
- $\Delta_I = \{\delta_I(v) | v \in \mathcal{V}\}$  such that  $\delta_I(v)$  is the probability of initial vertex being  $v$ .
- $\Delta_T = \{\delta_T(v, f_{v,k}(\vec{x}), v') | v, v' \in \mathcal{V}, k = 1, \dots, K_v\}$  such that  $\delta_T(v, f_{v,k}(\vec{x}), v')$  is the probability of making transition from  $v$  to  $v'$  when  $f_{v,k}(\vec{x})$  is true. Note that

$$\sum_{v' \in \mathcal{V}} \delta_T(v, f_{v,k}, v') = 1.$$

for all  $v$  and  $k$ .

The Finite State Automaton (FSA) policy is a special class of PFSA policies, where  $\Psi$ ,  $\delta_I$  and  $\delta_T$  are deterministic. Littman *et al.* [76] shows that the problem of finding the optimal FSA/PFSA policy is PSPACE-complete. Note that although probabilistic decision

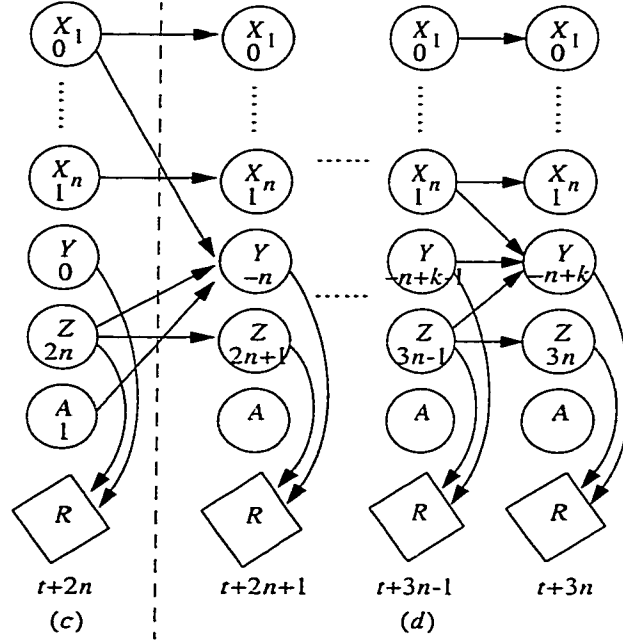


Figure 3.5: A task determining majority represented as a FMDP (continued from Figure 3.4). (c) answer guess phase: In this example, at time  $t+2n$ , the guess “yes” is fed into the FMDP. (d) answer verification phase: The guess answer is checked and the reward of 1.0 will be given at time  $t+3n$  if and only if  $-n/2 < -n+k \leq n/2$ .

tree policies are more restricted than PFSA policies, they are in the same computational complexity class when trying to compute the optimal policy, namely PSPACE-complete. Although these problems with two different types of policies are *equally* hard to compute in the worst case, we show that there are examples where representing the optimal policy as a decision tree is infeasible whereas the optimal policy can be represented as a small PFSA policy.

**Theorem 3.2.1** *There exists an FMDP where the optimal policy represented as a decision tree is of size exponential in the number of fluents and as a PFSA is of size polynomial in the number of fluents.*

**Proof** Consider the problem given as a FMDP shown in Figure 3.4 and continued in Figure 3.5. The FMDP is constructed in such a way that its state is a randomly generated instance of the majority problem and the reward is given when the correct answer is given as the action. Let  $M_{\text{MAJ}}$  denote this FMDP. The optimal policy is to determine whether the majority of bits in a random  $n$ -bit vector is set or not by examining the fluents. The FMDP representation is composed of  $n+2$  variables, where  $X_1, \dots, X_n$  represent the  $n$ -bit

vector encoding the problem instance,  $Y$  represents the *answer* to the problem (yes or no), and  $Z$  represents a counter for keeping track of the four *phases* of the process:

$$Z = \begin{cases} 0 & \text{answer generation phase,} \\ z \in [1, 2n] & \text{problem instance generation phase,} \\ 2n & \text{answer guess phase,} \\ z \in [2n + 1, 3n] & \text{answer verification phase.} \end{cases}$$

Assuming that the first phase starts at time  $t$ , the dynamics of  $M_{\text{MAJ}}$  is described as follows (Unless otherwise specified, all fluents except  $Z$  retain values from the previous time step, and  $Z$  is increased at each time step):

- In *answer generation phase*,  $M_{\text{MAJ}}$  randomly generates an answer.
  - time  $t$ :  $Y_t$  is set to a random number  $y_t \in [1, n]$ .  $Z_t$  is set to 0.
- In *problem instance generation phase*,  $M_{\text{MAJ}}$  randomly generates a problem instance corresponding to the answer generated in the previous phase.
  - time  $t + 1 \sim t + 2n$ : For  $1 \leq i \leq n$ ,  $X_{i,t+2i-1}$  is set to one with probability  $\min[y_{i,t+2i-2}/(n - i + 1), 1]$  at time  $t + 2i - 1$ .  $Y_{t+2i}$  is set to  $y_{t+2i-1} - x_{i,t+2i-1}$ . In short, throughout the problem generation phase,  $M_{\text{MAJ}}$  generates a random  $n$ -bit vector which has  $y_t$  bits set to 1, and  $Y$  serves as a counter keeping track of how many more bits should be set.
- In *answer guess phase*, the answer is guessed by the current policy.
  - time  $t + 2n$ : An action is taken, which corresponds to the guess whether the majority of the  $X_1, \dots, X_n$  are set or not. 0 means the answer “no” and 1 means “yes”.
- In *answer verification phase*,  $M_{\text{MAJ}}$  verifies the answer by counting the number of fluents that are set to 1 among fluents  $X_1, \dots, X_n$ .
  - time  $t + 2n + 1$ :  $Y_{t+2n+1}$  is set to  $a_{t+2n} * (-n) + x_{1,t+2n}$ .
  - time  $t + 2n + 2 \sim t + 3n$ : For  $2 \leq i \leq n$ ,  $Y_{t+2n+i}$  is set to  $y_{t+2n+i-1} + x_{i,t+2n+i-1}$ .
  - time  $t + 3n$ : If  $-n/2 < Y_{t+3n} \leq n/2$ , the guess was correct, so the reward of 1 is given. Otherwise, no reward is given.
  - time  $t + 3n + 1$ : The system gets back to the answer generation phase.

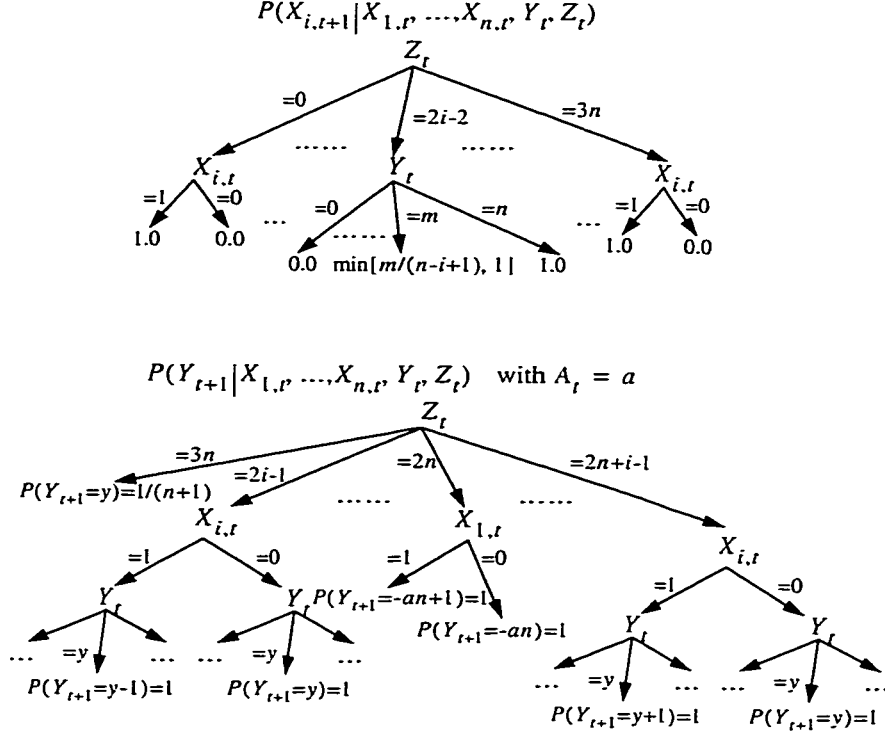


Figure 3.6: CPTs for fluents  $X_i$  (top) and  $Y$  (bottom).

A history independent policy represented as a tree must be able to count the number of bits set among  $X_1, \dots, X_n$ . As such, it will be a full tree with the number of leaves being  $2^n$ . On the other hand, a history dependent policy with sufficient amount of memory (size of 2 in the above case) to remember the answer can represent the optimal policy.

We show CPTs for fluents in Figure 3.6. The CPT for  $X_i$  remains persistent except when the value of fluent  $Z$  is  $2i - 2$ , which happens once in the problem instance generation phase. The CPT for  $Y$  is a little more complex. The first branch from  $Z$  with its value being  $3n$  represents that the previous problem just ended, so  $Y$  is sampled uniformly between 0 and  $n$  (answer generation phase). The second branch from  $Z$  with its value being  $2i - 1$  represents that  $X_i$  has been sampled, so the value of  $Y$  is adjusted to count the set bits (problem instance generation phase). The third branch from  $Z$  with its value being  $2n$  represents the start of the answer verification phase, so  $Y$  is initialized to  $-an$  and the value of  $X_1$  is added. The last branch from  $Z$  with its value being  $2n + i - 1$  represents that  $X_i$  is being examined, so the  $Y$  is adjusted to count the set bits (answer verification phase). All other branches are persistent trees.

Note that fluent  $Z$  has only one parent, which is the same fluent  $Z$  at the previous time



step, so the size of the CPT for fluent  $Z$  is not an issue. Even for fluents  $X_1, \dots, X_n$  and  $Y$ , the sizes of the CPTs are polynomial in the number of fluents due to context-specific transition — the probability depends on at most one of  $X_1, \dots, X_n$  given the value of  $Z$ .  $\square$

The memory of the PFSA policy maintains the context information of the FMDP that might otherwise require a large conditional table or decision tree: Since FMDPs assume full observability, all information necessary for selecting the optimal action is available encoded in the state at the time it is needed. In some cases, however, this information can be computationally difficult to decode and hence the anticipatory clues can afford a significant computational advantage. Particularly in Section 4.4, we explore this idea of searching for PFSA policies in FMDPs.

## Chapter 4

# FMDP Planning

As we have seen in the previous chapter, FMDP planning is a hard problem. Given that the problem is PSPACE-complete, there is not known to be any efficient algorithm that solves FMDP planning problems in polynomial time. Hence, the algorithms research in FMDP planning has been focused on identifying and exploiting various types of regularity in the domain. Before presenting the algorithms for FMDP planning, let us recall that there are a few challenges in adapting classical MDP algorithms to FMDPs.

First, the size of the representation for the value function may grow exponentially in the number of fluents. Any application of classical MDP algorithms based on value functions faces this problem. The common approach is to use Value Function Approximation (VFA) algorithms that restrict the class of the functions that  $V(\vec{x})$  can represent. For example, Boutilier and Dearden [15] use a tree representation for the table storing the value function, and prune the tree so that the table is of a maintainable size; Tsitsiklis and Van Roy [106] use a linear function for representing the value function in large MDPs (but not FMDPs), and demonstrate the difficulty in using any function more general than linear. In parallel to their work, Gordon [47] has pointed out (also summarized in Section 2.4) that a class of functions called *averagers*, which includes the linear function, can be used with classical MDP algorithms and guarantees convergence to a bounded approximation. The work by Koller and Parr [70, 71] explores stable VFA in FMDPs.

Second, the classical MDP algorithms enumerate all of the actions in the domain. This means that, for example, if we were to model a multi-controller problem where an action encodes “set controller knob #1 to high and controller knob #2 to low...”, or a logistics problem where an action encodes “ship  $N$  items to the retailer #1”, we may end up with

a very large domain representation. Hence, the classical MDP algorithms suffer from combinatorial explosion. Although this aspect of real world problems is not addressed in the standard representation of FMDPs, we will present how to cope with such circumstances by extending the standard definition and the FMDP algorithms.

Third, although the value function forms the basis for a very useful heuristic for calculating the optimal policy, since it is hard to compute, we may want to design techniques that do not depend on the value function. In fact, among reinforcement learning algorithms, although traditional algorithms like Q-learning (Watkins and Dayan [107]) are value function-based algorithms, there are algorithms like REINFORCE (Williams [111]) which do not directly calculate the value function. However, regardless of the algorithm, the size of the representation for the policy can explode. This is not surprising since if we treat the policy as a function that takes the state space as its domain and the action space as its range, the function can be arbitrarily complex to represent. In this case, we may want to trade the quality of the policy against the size of the representation, or we can search for other kinds of policy. Particularly, in Theorems 3.1.1 and 3.1.2, we have shown that searching for an optimal PFSA policy (Definition 3.2.1) is not significantly harder than searching for an optimal decision tree policy. In Section 4.4, we will present techniques that deal with the challenge of representing the policy.

The algorithms we provide in this chapter are said to be *structured* since they typically avoid enumerating all the states and actions in the domain. These algorithms attempt to exploit certain types of regularity in the domain to prevent combinatorial explosion. The algorithms in Section 4.1 exploit the fact that the FMDP may be reduced to a small MDP by aggregating states that behave the same. The algorithms in Section 4.2 exploit the fact that finding an optimal policy does not necessarily require the optimal value function. The algorithm in Section 4.3 exploits the fact that the domain may be decomposed into separate sub-domains. Section 4.4 provides an algorithm that remembers a past cue that is critical for the decision making at the current time step.

## 4.1 Homogeneous Aggregation Algorithms

Dean and Givan [32] introduce the notion of *stochastic bisimulation homogeneity* for FMDPs. Note that if we were to apply the algorithm in Section 2.5, we would have to enumerate all the states in the FMDP, and the number of states is exponential in the number of fluents. In this section, we present a structured algorithm in the sense that it does

not enumerate all the states<sup>1</sup>.

Recall that stochastic bisimulation homogeneity is an extension of the state equivalence relation for minimizing Finite State Automata (FSAs) (Hopcroft and Ullman [54]) to that of PFSAs. FMDPs can be seen as PFSAs with a large number of states with the output being the reward. Formally, the FMDP minimization algorithm partitions the state space into blocks. A partition of state space  $S$  is a set of blocks  $\{B_1, \dots, B_n\}$  where each block is a mutually disjoint set with other blocks and the union of the blocks is the whole state space  $S$ . We are especially interested in finding a partition with *stable* blocks. We restate the definition of a stable block (Definition 2.5.1) for FMDPs:

**Definition 4.1.1 (Stable Block and Homogeneous Partition)** *A block  $C$  of a partition  $P$  is called stable with respect to a block  $B$  of  $P$  and action  $a$  if and only if every state in  $C$  has the same transition probability of going into block  $B$  by action  $a$ . Mathematically,*

$$\exists c \in [0, 1] \text{ and } \exists c' \in \mathfrak{R} \text{ such that } \forall \bar{x}_t \in C, T(\bar{x}_t, a, B) = c \text{ and } R(\bar{x}_t) = c'$$

where

$$T(\bar{x}_t, a, B) = \sum_{\bar{x}_{t+1} \in B} T(\bar{x}_t, a, \bar{x}_{t+1}).$$

*We say that  $C$  is stable if  $C$  is stable with respect to every block of  $P$  and action in  $A$ .  $P$  is called homogeneous if and only if every block is stable.*

The idea behind the homogeneous partition of an FMDP is that it induces an explicit MDP that is *equivalent* to the original FMDP. Note that every state within a block of a homogeneous partition has the same state dynamics with respect to any policy, *i.e.*, the transition probability of moving into a block is the same for every state in the same block. Also, every state within a block of a homogeneous partition has the same reward. Hence, the homogeneous partition yields a reduced model of the original FMDP if the number of blocks in the partition is smaller than the number of states in the original FMDP.

Given the definition of homogeneous partition in FMDPs, the goal is to find the *coarsest homogeneous partition*. In other words, we want to obtain a homogeneous partition with the smallest number of blocks. Each block constitutes a state of the induced MDP. We call a state of the induced MDP an *aggregate state*, since each state of the induced MDP usually includes a large number of states of the original FMDP. We also want to avoid enumerating all the states of the FMDP. The algorithm we present iteratively refines the reward partition

---

<sup>1</sup>A preliminary work of the material presented here appeared in Dean *et al.* [33]

of an FMDP to achieve the coarsest homogeneous partition without enumerating all the states. The reward partition is defined as follows:

**Definition 4.1.2 (Reward Partition)** *Each block  $C$  in the reward partition of FMDP  $M$  is defined as follows:*

$$\vec{x} \in C \text{ and } \vec{x}' \in C \text{ if and only if } R(\vec{x}) = R(\vec{x}').$$

Note that a homogeneous partition is a refinement of the reward partition. A refinement  $P'$  of partition  $P$  is defined by

$$\forall C' \in P', \exists C \in P \text{ such that } C' \subseteq C.$$

To verify that a homogeneous partition is a refinement of the reward partition, observe that two individual states having the same reward is a necessary condition for belonging to the same block in a homogeneous partition.

Before we present how we obtain the coarsest homogeneous partition from the reward partition, we make another useful observation:

**Theorem 4.1.1 (Necessary Condition for Homogeneous Partition)** *Assume partition  $P$  such that the coarsest homogeneous partition is a refinement of  $P$ . Given partition  $P$ , blocks  $B, C \in P$ , and  $\vec{x}_t, \vec{x}'_t \in C$ , if*

$$\exists a \in A \text{ such that } T(\vec{x}_t, a, B) \neq T(\vec{x}'_t, a, B)$$

*then  $\vec{x}_t$  and  $\vec{x}'_t$  do not belong to the same block of the coarsest homogeneous partition.*

**Proof** Assume that  $\vec{x}_t$  and  $\vec{x}'_t$  belong to the same block of the coarsest homogeneous partition. To verify that there exists block  $B$  such that  $T(\vec{x}_t, a, B) = T(\vec{x}'_t, a, B)$ , let us say  $B$  is a union of mutually disjoint blocks  $B_1, \dots, B_k$  from the coarsest homogeneous partition. We can find such blocks since the coarsest homogeneous partition is a refinement of  $P$ . Since  $B_1, \dots, B_k$  are mutually disjoint, we have

$$T(\vec{x}_t, a, B) = \sum_{i=1}^k T(\vec{x}_t, a, B_i),$$

and

$$T(\vec{x}'_t, a, B) = \sum_{i=1}^k T(\vec{x}'_t, a, B_i).$$

It follows that  $T(\vec{x}_t, a, B) = T(\vec{x}'_t, a, B)$  since the right hand sides of the two equations are same from the definition of a homogeneous partition.  $\square$

Since the coarsest homogeneous partition is a refinement of the reward partition, the algorithm repeatedly refines the reward partition by *splitting* a block that is not stable. The algorithm uses the SPLIT operator to generate a refinement at each iteration, which we define as follows.

The model minimization algorithm using the SPLIT operator is defined as follows: for a pair of blocks  $(B, C)$  in the partition, check whether  $C$  is stable with respect to  $B$  for each action. When  $C$  is found to be not stable for action  $a$ , replace  $C$  by sub-blocks  $C_i$ 's such that each sub-block is stable with respect to  $B$  and  $a$ . We denote the resulting partition of  $C$  by  $\text{SPLIT}(B, C, P, a)$  and say that  $C$  is *refined*. Note that the homogeneous partition obtained by repeated application of the SPLIT operator does not necessarily lead to the coarsest homogeneous partition. For example, among the newly generated blocks from  $\text{SPLIT}(B, C, P, a)$ , there may exist blocks that may be *merged* together while still holding the homogeneity with respect to block  $B$ . The following theorem states that we obtain the coarsest homogeneous partition if the SPLIT operator generates the coarsest refinement of  $C$ :

**Theorem 4.1.2 (Coarsest Homogeneous Partition and SPLIT Operator)** *Define  $\text{SPLIT}(B, C, P, a)$  which generates the coarsest refinement of  $C$  by  $\text{SPLIT}_{\text{Opt}}(B, C, P, a)$ . Given an initial partition  $P_0$ , the model minimization algorithm with  $\text{SPLIT}_{\text{Opt}}$  operator computes the coarsest homogeneous refinement of  $P_0$ .*

**Proof** Assume that the algorithm produced a non-coarsest homogeneous refinement of  $P_0$ , and let us call it  $P'$ . This implies that there are two stable blocks  $C_1$  and  $C_2$  that are split during the course of the algorithm but could have been merged together without violating the homogeneity condition. Consider the first time  $C_1$  and  $C_2$  get separated, and let us define this event by  $\{C'_1, C'_2, \dots\} = \text{SPLIT}_{\text{Opt}}(B, C', P, a)$  such that  $C_1 \subseteq C'_1$  and  $C_2 \subseteq C'_2$ . Since  $C'_1$  and  $C'_2$  are stable with respect to block  $B$ , it follows that

$$T(C'_1, a, B) = T(C_1, a, B),$$

and

$$T(C'_2, a, B) = T(C_2, a, B),$$

and by assumption,  $T(C_1, a, B) = T(C_2, a, B)$ . Thus,  $C'_1$  and  $C'_2$  can be merged together and this implies that  $\text{SPLIT}_{\text{Opt}}$  did not produce coarsest refinement.  $\square$

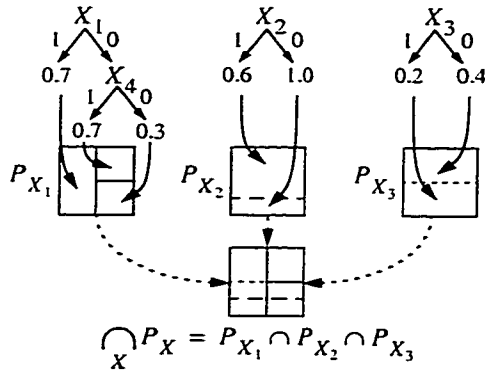


Figure 4.1: Partition of the state space induced by CPTs

Generating a coarsest refinement of an unstable block is an NP-hard problem, since it requires checking satisfiability of block formulas. Thus, in practice, we may be satisfied with a SPLIT operator which generates non-coarsest but still homogeneous blocks.

Given the decision tree representation of the reward function, obtaining the reward partition can be done in a polynomial time in the size of the decision tree: The decision tree uses fluents to branch at non-terminal nodes. Each terminal node represents a block of the partition and the formula associated with it is the conjunction ( $\wedge$ ) of the fluents and their values found in the path from the leaf to the root. Since the model minimization algorithm with a SPLIT operator induces the partition of the states which has same dynamics, if we refine it with the partition induced by the reward function, we obtain the reduced model, if not the minimal model, which is *equivalent* to the original MDP. Thus, if we solve this reduced MDP, we get a solution to the original FMDP.

Figure 4.1 shows illustrates how  $\text{SPLIT}(B, C, P, a)$  operator works. We first collect CPTs for the fluents used in the description for block  $B$ . In the figure, we assume that  $B$  is labeled as  $(X_1 \wedge X_2 \wedge X_3)$ . Since CPTs induce a partition of the state space, they refine block  $C$ . The figure shows how  $X_1, X_2$  and  $X_3$  differently refine  $C$ . The refined partitions are shown as  $P_{X_1}, P_{X_2}$  and  $P_{X_3}$ . The combination of all the refined partitions is shown in the bottom, which is  $\bigcap_X P_X$ .

In Dean *et al.* [33], we extended the model minimization algorithm to FMDPs with factored action spaces, which we call FA-FMDP hereafter. The action space is specified in terms of action fluents, *i.e.*,  $\vec{A} = [A_1, \dots, A_m]$ . Given  $\vec{x}_t, \vec{x}_{t+1} \in \vec{X}$  and  $\vec{a} \in \vec{A}$ , the transition probability is defined as follows:

$$T(\vec{x}_t, \vec{a}, \vec{x}_{t+1}) = \prod_{i=1}^n P(x_{i,t+1} | \text{pa}(x_{i,t+1}))$$

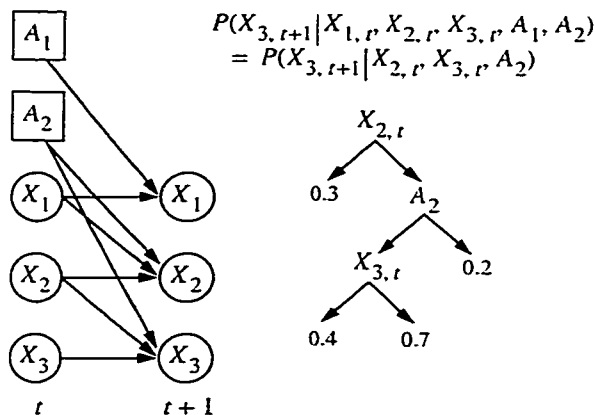


Figure 4.2: Graphical representation of a FA-FMDP.

The difference from the definition of  $T$  in an FMDP is that the parent set  $\text{pa}(\cdot)$  of each fluent includes action fluents. Figure 4.2 presents an example of the graphical model for a FA-FMDP.

Having a factored action space allows us to compactly model domains with a large number of actions, such as controlling multiple effectors on a robot or allocating a set of resources. To prevent us from facing the combinatorial explosion, we are interested in exploiting the regularity in state space *and* action space. For example, a robot traversing a darkened room can adjust its camera in a variety of ways, but only the sonar and infrared sensors will have any impact on the robot's success in navigating the room. If we model this example as a FA-FMDP, there are three action fluents, and only two of them are relevant to an optimal policy, namely sonar and infrared, when the state fluent indicating the luminosity of the room is set to "dark". The idea behind the model minimization algorithm for FA-FMDPs is that it will abstract out useless actions from consideration in appropriate sets of states.

First, we make additional definitions regarding FA-FMDPs for use with the model minimization algorithm:

**Definition 4.1.3** *Given a FA-FMDP  $M = \{\vec{X}, \vec{A}, T, I, R\}$ , a partition  $P$  of the space  $\Omega_{\vec{X}} \times \Omega_{\vec{A}}$  (which is a refinement of the reward partition), we make the following definitions:*

- **Projection** : *The projection  $P|_{\vec{X}}$  of  $P$  onto  $\vec{X}$  is the partition of  $\Omega_{\vec{X}}$  such that two states  $\vec{x}$  and  $\vec{x}'$  are in the same block if and only if for every action  $\vec{a}$  in  $\vec{A}$ , the pairs  $(\vec{x}, \vec{a})$  and  $(\vec{x}', \vec{a})$  are in the same block of  $P$ . Likewise, the projection  $P|_{\vec{A}}$  of  $P$  onto  $\vec{A}$  is the partition of  $\Omega_{\vec{A}}$  such that two actions  $a$  and  $a'$  are in the same block if and only if for every state  $\vec{x}$  in  $\vec{X}$ , the pairs  $(\vec{x}, a)$  and  $(\vec{x}, a')$  are in the same block of  $P$ .*



1. Input: FA-FMDP  $M = \{\vec{X}, \vec{A}, T, I, R\}$  and reward partition  $P$ .
2. Calculate projection  $P|_{\vec{X}}$ .
3. For each  $B \in P|_{\vec{X}}$  and  $C \in P$ , do  $\text{SPLIT}(B, C, P)$  to examine whether  $C$  gets split. If block  $C$  gets split, replace  $C$  in  $P$  by  $\text{SPLIT}(B, C, P)$  and go to step 2.
4. If no block in  $P$  gets split in the previous step, output the quotient MDP  $M_P = \{P|_{\vec{X}}, P|_{\vec{A}}, T, I, R\}$ .

Figure 4.3: The first version of FA-FMDP minimization algorithm.

- **Stability** : A block  $C$  of a partition  $P$  is called stable with respect to a block  $B$  of  $P|_{\vec{X}}$  if and only if every two pairs  $(\vec{x}, \vec{a})$  and  $(\vec{x}', \vec{a}')$  in block  $C$  satisfy  $T(\vec{x}, \vec{a}, B) = T(\vec{x}', \vec{a}', B)$ . A block  $C$  is said to be stable if  $C$  is stable with respect to every block of  $P|_{\vec{X}}$ .
- **Homogeneity** :  $P$  is homogeneous if and only if every block is stable.
- **Quotient MDP** : When  $P$  is homogeneous, the Quotient MDP is defined to be the explicit MDP  $M_P = \{P|_{\vec{X}}, P|_{\vec{A}}, T_P, I_P, R_P\}$  where  $T_P$  is defined as the transition probabilities between the blocks of  $P$ ,  $I_P$  is defined as the initial probabilities on the blocks of  $P$  and  $R_P$  is defined as the reward function on the blocks of  $P$ .

It follows that the optimal value function  $V^*$  and the optimal policy  $\pi^*$  of the quotient MDP obtained by the minimization algorithm is also the optimal value function and the optimal policy for the original FA-FMDP. The new minimization algorithm is based on the new definitions of stability and homogeneity.

Figure 4.3 shows our first version of FA-FMDP minimization algorithm. The new  $\text{SPLIT}(B, C, P)$  operator of the algorithm takes a partition  $P$  of  $\Omega_{\vec{X}} \times \Omega_{\vec{A}}$ , block  $B$  in  $P|_{\vec{X}}$ , and block  $C$  in  $P$  as input and returns sub-blocks  $C_i$  of  $C$  such that each of the sub-blocks is stable with respect to  $B$ . Note that computing projections is an NP-hard task itself, hence we want to avoid doing it every time a block gets split.

**Theorem 4.1.3** *Computing projections  $P|_{\vec{X}}$  and  $P|_{\vec{A}}$  are NP-hard problems.*

**Proof** Assume FA-FMDP  $M = \{\vec{X}, \vec{A}, T, I, R\}$  with  $n + 2$  state fluents and one action fluent. Given propositional formula  $\Phi$  of state fluents  $X_1, \dots, X_n$ , unused state fluents  $X_{n+1}, X_{n+2}$ , an action fluent  $A_1$ , consider the projection of the partition  $P = \{C_1, C_2, C_3\}$

1. Input: FA-FMDP  $M = \{\vec{X}, \vec{A}, T, I, R\}$  and reward partition  $P$ .
2. Set  $P' = P$  and calculate  $P = \text{BACKUP}(P')$ .
3. If  $|P| > |P'|$ , goto step 2.
4. Output the quotient MDP  $M_P = \{P|_{\vec{X}}, P|_{\vec{A}}, T, I, R\}$ .

Figure 4.4: The second version of FA-FMDP minimization algorithm.

defined by

$$\begin{aligned} C_1 &\equiv (\Phi \wedge A_1) \vee (X_{n+1} \wedge X_{n+2}), \\ C_2 &\equiv (\Phi \wedge \overline{A_1}) \vee (X_{n+1} \wedge \overline{X_{n+2}}), \\ C_3 &\equiv \overline{\Phi} \vee \overline{X_{n+2}}, \end{aligned}$$

onto the action space, which consists of only one action fluent  $A_1$ . The projected partition has two blocks if  $\Phi$  is satisfiable and one block if not. The fluents  $X_{n+1}$  and  $X_{n+2}$  are for ensuring that none of  $C_1, C_2$  and  $C_3$  is empty when  $\Phi$  is a tautology or a contradiction. Thus, calculating  $P|_{\vec{A}}$  is NP-hard. Exchange state fluents and action fluents to obtain the same result for  $P|_{\vec{X}}$ .  $\square$

Instead of computing the projection  $P|_{\vec{X}}$  whenever a block gets split, we define  $\text{BACKUP}(P)$  operator for input partition  $P$ , which uses partition  $P_X$  for each state space fluent  $X$  provided as CPTs. Let  $\vec{X}_P$  be the set of all state space fluents which are mentioned in any formula in the representation of the current partition  $P$ . First, construct the intersection  $I_P$  of all partitions  $P_X$  such that the fluent  $X$  is in  $\vec{X}_P$ :

$$I_P \equiv P \cap \bigcap_{X \in \vec{X}_P} P_X \quad (4.1)$$

$I_P$  makes all the distinctions needed within  $\vec{X} \times \vec{A}$  so that within each block of  $I_P$ , the block transition probability to each block of  $P|_{\vec{X}}$  is the same, which implies that blocks in  $I_P$  are stable with respect to the blocks in  $P$ . For a formal discussion of stability in  $P$ , refer to the definition of block transition probability in the proof of Theorem 4.1.4. Note, however, that calculating the intersection is also NP-hard since it also requires determining whether a block is empty or not, which is again a propositional satisfiability problem when we allow general boolean functions to represent partitions. Note also that  $I_P$  may make distinctions that do not need to be made, thus lead to a non-minimal partition. For this reason, it is necessary to define  $\text{BACKUP}(P)$  as the *clustering* of the partition  $I_P$  in which blocks of  $I_P$

are merged if they have identical block transition probabilities to the blocks of  $P|_{\bar{x}}$ . We merge two blocks taking the disjunction of the block formulas for those blocks.

The computational intractabilities we have seen so far come from the fact that we are allowing *general propositional formula* for representing partitions. Hence, if we restrict the class of block formulas to, say, simple conjuncts, the key operations such as projection, intersection between two partitions, and clustering become tractable. Note that such a restriction may prevent us from merging certain blocks even if they can be merged. For example, two blocks with block formulas  $X_1 \wedge \overline{X_2}$  and  $\overline{X_1} \wedge X_2$  cannot be merged although they have the same transition probabilities with respect to other blocks and the same reward. Such phenomena can lead to exponentially larger partitions than those corresponding to the minimal model. In the process of restricting the class of block formulas, we are trading the potentially increased computational cost for BACKUP for the decreased number of blocks in the final homogeneous partition. In Section 4.1.1, we show experimental results of our implementation using simple conjuncts represented as trees and certain propositional formulas represented as decision diagrams. For theoretical purposes, let us assume that we use a general propositional formula for representing the partitions. The algorithm works by repeatedly applying the BACKUP operator on the reward partition, and it has the following property:

**Theorem 4.1.4 (Coarsest Homogeneous Partition of FA-FMDP)**

*Let  $\text{BACKUP}_{\text{Opt}}(P)$  be the operator that generates the coarsest refinement of  $P$  among all BACKUP operators. Given an initial partition  $P_0$ , the model minimization algorithm with  $\text{BACKUP}_{\text{Opt}}$  operator computes the coarsest homogeneous refinement of  $P_0$ .*

**Proof** Given a homogeneous partition  $P$  of  $\Omega_{\bar{x}} \times \Omega_{\bar{A}}$ , define block transition probability from block  $C \in P$  to block  $B \in P|_{\bar{x}}$  by

$$T(C, B) \equiv \sum_{(\bar{x}', \bar{a}') \in B} T(\bar{x}, \bar{a}, \bar{x}')$$

for any  $(\bar{x}, \bar{a}) \in C$ . This is well defined since  $C$  is stable with respect to all blocks in  $P|_{\bar{x}}$ .

Assume that the minimization algorithm produced a non-coarsest homogeneous refinement of  $P_0$ , which we call  $P'$ . This implies that there are two stable blocks  $C_1$  and  $C_2$  that are split during the course of the algorithm but could have been merged without violating the homogeneity condition. Consider the first time  $C_1$  and  $C_2$  get separated, and let us define this event by  $\{C'_1, C'_2, \dots\} = \text{BACKUP}_{\text{Opt}}(P)$  such that  $C_1 \subseteq C'_1$  and  $C_2 \subseteq C'_2$ . Since

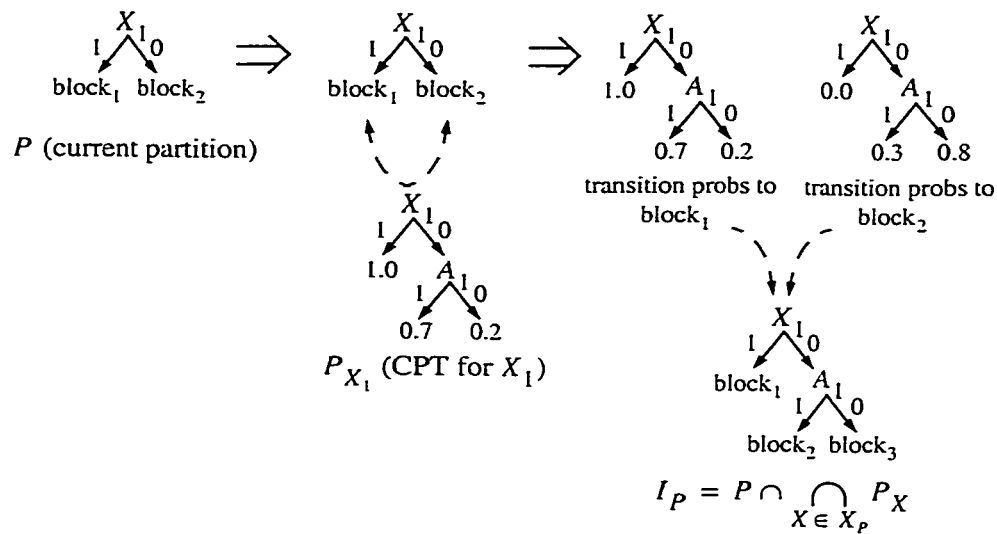


Figure 4.5:  $\text{BACKUP}(P)$  in FA-FMDP minimization using decision trees. We construct intermediate partitions storing the probabilities for each terminal node in  $P$ , then take the intersection to calculate  $I_P$ .

$C'_1$  and  $C'_2$  are stable, it follows that

$$\forall B \in P, T(C'_1, B) = T(C_1, B),$$

and

$$\forall B \in P, T(C'_2, B) = T(C_2, B),$$

and by the assumption that  $C_1$  and  $C_2$  are blocks from a homogeneous partition,  $\forall B \in P, T(C_1, B) = T(C_2, B)$ . Thus,  $C'_1$  and  $C'_2$  can be merged together and this implies that  $\text{BACKUP}_{\text{Opt}}$  did not generate the coarsest refinement of  $P$ .  $\square$

The two versions of FA-FMDP minimization algorithms are not very different from one another. The first version takes a pessimistic approach assuming that the splitting may result in a large number of blocks, so it works on only one block at each iteration. The second version takes an optimistic approach assuming that the splitting, in other words intersection, does not result in a large partition, so it works on all blocks in the partition at each iteration. As the above theorem states, both versions lead to the coarsest homogeneous partition.

We implemented two algorithms using different representations for partitions.

The first algorithm uses decision trees. The calculation of  $\text{BACKUP}(P)$  is done by grafting one tree to another to obtain intersection of the partitions. Collecting the relevant

fluent set  $\bar{X}_P$  in Equation 4.1 is done by traversing the decision tree representation of  $P$  and selecting the non-terminal nodes. Once we obtain  $\bar{X}_P$ , we graft each CPT of fluent  $X \in \bar{X}_P$  to the decision tree representing  $P$ . This procedure results in  $I_P$  of Equation 4.1. Figure 4.5 shows an example.

The last stage of BACKUP is to merge the blocks that have the same reward and same transition probabilities with respect to the blocks in  $P$ . To this end, we calculate the rewards and transition probabilities and store them in appropriate blocks in  $I_P$ . After this procedure, merging the equivalent blocks is done by simplifying the decision tree representing  $I_P$ . Traversing up from the terminal nodes, we check whether the left subtree and right subtree are identical for each non-terminal node, and if they are, we replace the non-terminal node with one of the subtrees. We do not consider reordering of the fluents within the trees since it is costly to do so.

Finally, we calculate the projections  $P|_{\bar{X}}$  and  $P|_{\bar{A}}$  of a homogeneous partition  $P$  resulting from repeated application of BACKUP. The projection onto the state space  $P|_{\bar{X}}$  is done by replacing every non-terminal node representing an action fluent by the intersection of its left and right subtrees. Again, as in BACKUP, the intersection of left and right subtrees is done by grafting one subtree to the other. The projection onto the action space  $P|_{\bar{A}}$  is done in the same manner, replacing every non-terminal node representing a state fluent by the intersection of its left and right subtrees.

The second algorithm uses Algebraic Decision Diagrams (ADDs). The ADD is a generalization of the Boolean Decision Diagram (BDD), in which the terminal nodes can have an arbitrary value instead of a boolean value in  $\{0, 1\}$  (Bryant [25], Bahar *et al.* [3]). An ADD represents a function  $f : \{0, 1\}^n \rightarrow \mathfrak{R}$  with a set of non-terminal nodes and terminal nodes. The non-terminal nodes form vertices in a directed acyclic graph  $(V, E, W)$  such that every out-degree of a vertex  $v \in V$  is 2. Every node has a label  $l(v) \in \{X_1, \dots, X_n\}$  so that it identifies the boolean domain variable on which function  $f$  depends. Additionally, there is an ordering of the variables, defined as  $\succ$ , so that if there is an edge from a non-terminal node  $v$  to another non-terminal node  $v'$ , then  $l(v) \succ l(v')$ . There is a special non-terminal node designated as the start node. An edge can also have a terminal node as its destination. For every pair of outgoing edges, one of them is labeled 1 and the other is labeled 0. The terminal nodes hold constant values and their out-degrees are 0. Figure 4.6 presents an example of an ADD. The function that an ADD represents is defined as follows:

- The function of a terminal node is a constant function of the value stored in the terminal node.

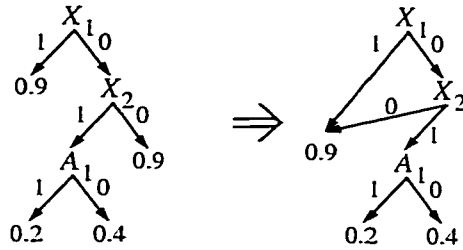


Figure 4.6: An example of a decision tree and an ADD.

- The function of a non-terminal node  $v$  is defined as

$$l(v) \cdot f_1 + \overline{l(v)} \cdot f_0$$

where  $f_1$  is the function of the child node connected by the edge labeled 1, and  $f_0$  is the function of the child node connected by the edge labeled 0.

- The function  $f$  of an ADD is the function of the start node.

Now let us return to thinking about the use of decision trees in FA-FMDP minimization algorithms. These decision trees are actually functions that map an assignment of the fluents to values such as block number in a partition or transition probabilities. Representing such functions as decision trees motivates us to use more compact representations like ADDs.

To use ADDs, we first convert the domain into ADDs. We obtain ADD representations for CPTs and the reward, which are given as decision trees. This procedure is done in the same manner as Hoey *et al.* [53]. The CPT for fluent  $X_i$  can be seen as function

$$\begin{aligned} T_{X_{i,t+1}}(X_{1,t}, \dots, X_{n,t}, A_{1,t}, \dots, A_{m,t}) \\ \equiv X_{i,t+1} \cdot P(X_{i,t+1} | X_{1,t}, \dots, X_{n,t}, A_{1,t}, \dots, A_{m,t}) \\ + \overline{X_{i,t+1}} \cdot (1 - P(X_{i,t+1} | X_{1,t}, \dots, X_{n,t}, A_{1,t}, \dots, A_{m,t})), \end{aligned} \quad (4.2)$$

and hence, the transition probability is

$$\begin{aligned} T(X_{1,t}, \dots, X_{n,t}, A_{1,t}, \dots, A_{m,t}, X_{1,t+1}, \dots, X_{n,t+1}) \\ = \prod_{i=1}^n T_{X_{i,t+1}}(X_{1,t}, \dots, X_{n,t}, A_{1,t}, \dots, A_{m,t}). \end{aligned}$$

After we obtain ADD representations of  $T_{X_{i,t+1}}$  for all  $i$ , we can construct the ADD for the transition probability through multiplication, which is a standard operation that generally comes with ADD software packages. Constructing ADD representations of  $T_{X_{i,t+1}}$  (Equation 4.2) is done through traversing the CPT of  $P(X_{i,t+1} | X_{1,t}, \dots, A_{1,t}, \dots)$ . When

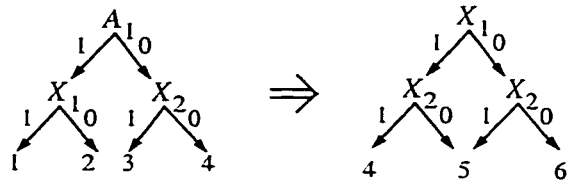


Figure 4.7: A bad case of existential abstraction. When we sum over action fluent  $A_1$ , two blocks merge since the additions of the block numbers are the same.

1. Input: Partition  $P$  of  $\Omega_{\bar{X}} \times \Omega_{\bar{A}}$ , in ADD representation.
2. For each action fluent  $A_i$ ,
  - (a) Let  $f = A_i \cdot (|P| + 1) + \bar{A}_i$ .
  - (b) Do existential abstraction (see the text) of  $f \cdot P$  over  $A_i$  and store the result as  $P$ .
  - (c) Normalize the terminal nodes in  $P$  so that they are numbered from 1 to  $|P|$ .
3. Output  $P$ .

Figure 4.8: The algorithm for projecting ADD representation of a homogeneous partition onto the state space.

we encounter non-terminal node  $X_i$  during the traversal, we first calculate the ADD representation for left subtree and right subtree, which we respectively call  $f_0$  and  $f_1$ , and then replace the node by the ADD representation

$$X_i f_1 + \bar{X}_i f_0.$$

The reward function given as a decision tree is converted in the same manner.

Once we have constructed ADD representations of the transition probability and the reward function, we can implement the BACKUP operator in terms of ADD operators. First, assume that the ADD representation of partition  $P$  is a mapping from fluents to the block number in the partition. Hence, given an assignment of values to fluents, the evaluation of the ADD results in 1 if the assignment belongs to the first block, 2 if second block, and so on. Projection of  $P$  onto the state space is done by summing over all possible values taken by the action fluent. This is called *existential abstraction*, which is supplied as a standard operator in ADD software packages. Note that we can accidentally merge two blocks during the existential abstraction. Figure 4.7 shows such a case. In the example, when we sum over action fluent  $A_1$ , two blocks merge although they should not. We want to differentiate  $X_1 \wedge \bar{X}_2$  and  $\bar{X}_1 \wedge X_2$  since they represent different blocks in the partition. This

1. Input: Partition  $P$  of  $\Omega_{\bar{X}} \times \Omega_{\bar{A}}$  and transition probabilities  $T$ , all in ADD representation.
2. Calculate projection of  $P$  onto the state space and call it  $P|_{\bar{X}}$ .
3. For each block  $B \in P|_{\bar{X}}$ ,
  - (a) Construct ADD for representing block  $B$  so that the ADD returns 1 if the assignment of the state fluents belongs to  $B$ , and returns 0 otherwise. Call it  $P_B$  and mark the fluents to be at time step  $t + 1$ .
  - (b) Perform ADD matrix-vector multiplication of  $T$  and  $P_B$  by summing over all the variables at time step  $t + 1$ . Let the result be  $P'_B$ .
  - (c) Normalize the terminal nodes in  $P'_B$  so that they are numbered from 1 to  $|P'_B|$ .
  - (d) Calculate  $P = P \cdot |P'_B| + P'_B$ .
  - (e) Normalize the terminal nodes in  $P$  so that they are numbered from 1 to  $|P|$ .
4. Output  $P$ .

Figure 4.9: The algorithm for the BACKUP operator on ADD representation of a partition.

behavior can be simply avoided by rescaling the block numbers so that the blocks with  $A_1$  are multiplied by the size of the partition and those with  $\bar{A}_1$  remains the same. The overall algorithm of projection onto the state space is shown in Figure 4.8. Avoiding accidentally merging blocks is in steps 2.(a) and 2.(b). The projection onto the action space is done in the same manner, except that we sum over all state fluents.

Figure 4.9 shows the core part of the minimization algorithm, the BACKUP operator.  $P'_B$  represents the partition of the state and action space so that every state-action pair in a block has the same transition probability into block  $B$ . The intersection of these partitions with the input partition is again done by existential abstraction. Note the normalization trick to avoid accidentally merging two different blocks.

### 4.1.1 Experiments

In this section, we present results from experiments regarding the homogeneous aggregation algorithms introduced in this section. As mentioned in the previous section, we have implemented two versions of the minimization algorithm for FA-FMDPs using different representation for partitions.

The first version uses decision trees for representing partitions. Note that using decision trees only allows simple conjunctions of fluents for representing blocks. A preliminary result from this implementation also appeared in Dean *et al.* [33].



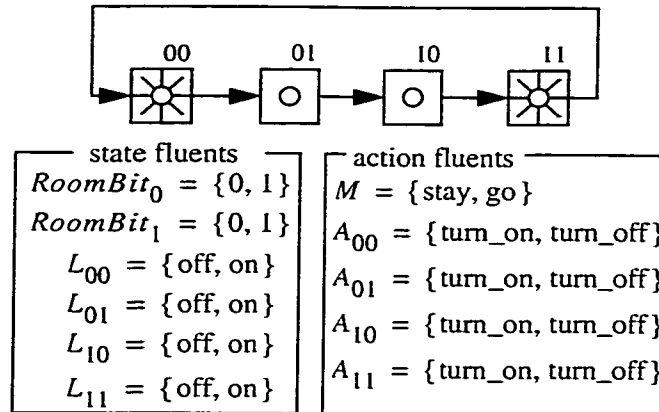


Figure 4.10: State and action fluents for the ROBOT- $k$  domain. This is an example where there are 4 rooms ( $k = 4$ ). The room numbers are encoded as bit vectors.

The second version uses Algebraic Decision Diagrams (ADDs) for representing partitions. We use C++ implementation of Colorado University Decision Diagram (CUDD) package (Somenzi [100]) for manipulating ADDs.

We tested both versions on a few selected test domains. Note that, due to the restriction in CUDD that only allows binary fluents, the domains only consist of binary fluents.

**Robot navigation in dark rooms (ROBOT- $k$ )** Consider a robot navigating in a simple collection of rooms. Suppose that there are  $k$  rooms connected in circular chain and each room has a light which is either on or off. The robot is in exactly one of the  $k$  rooms and so there are  $k2^k$  states. At each stage, the robot can choose to go forward or stay in the current room and choose to turn on or off any of the  $k$  lights, so there are  $2^{k+1}$  actions. Figure 4.10 shows the basic layout and the relevant state and action fluents. The fluents  $RoomBit_b$  encode the  $b$ -th bit for the binary representations of the room numbers. The fluents  $L$  represent the statuses of the lights in the rooms.

The robot receives a reward of 1 for being in the room  $k$  and otherwise receives no reward. If the light is on in the room where the robot is currently located and the robot chooses to go, then it will end up in the next room with probability 1 at the next time step. If the light is off and it chooses to go, it will remain in the current room with probability 1. If the robot chooses to stay, then it will remain in the current room with probability 1. If the robot sets the action parameter  $A_i$  to “turn\_on” or “turn\_off”, then with probability 1 at the next time step, the light in room  $i$  will “on” or “off”, respectively.

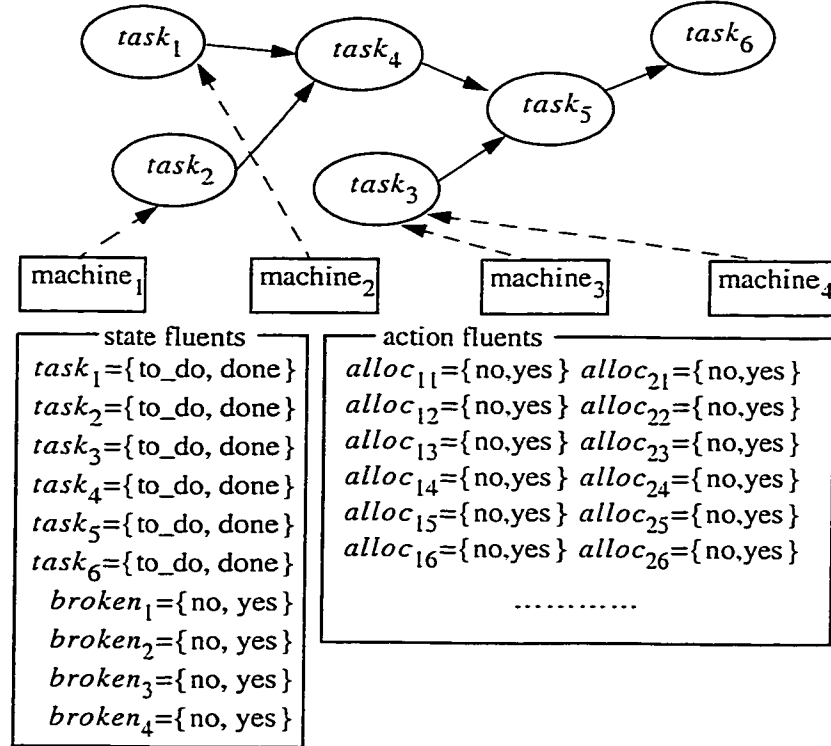


Figure 4.11: An example of JOBSHOP- $k$ - $l$  domain with 4 machines and 6 tasks. The solid arrows between tasks specify dependencies as pre-requisites. The dashed arrows between machines and tasks specify allocations.

**Jobshop scheduling (JOBSHOP- $k$ - $l$ )** Consider a machine shop that needs to allocate  $k$  machines to  $l$  tasks. The tasks are partially ordered, meaning that some tasks have pre-requisite tasks. Each task requires one or more specific machines to be allocated. Each machine is capable of working on only one task at each time step. Once a task succeeds in occupying the required set of machines, the task gets accomplished at the next time step with some probability. Thus, there are  $task_1, \dots, task_l$  state fluents holding the value either “to\_do” or “done”. All the machines become available at the next time step whether the assigned task gets accomplished or not. Hence, there are no fluents that specify the statuses of machines.

As action fluents, we have  $alloc_{ij}$  for  $1 \leq i \leq k$ ,  $1 \leq j \leq l$  holding the value either “yes” or “no”. If  $alloc_{ij}$  is set “yes”, task  $j$  succeeds in occupying machine  $i$  between the current time step and the next time step. The constraint that there is only one task allocated to each machine is implemented by adding state fluents  $broken_1, \dots, broken_k$ .  $broken_i$  represents whether the machine  $i$  is broken, and this fluent becomes “yes” whenever  $alloc_{ij}$  and  $alloc_{i'j'}$

Domain	Number of states	Number of actions	Number of state blocks	Number of action blocks	Running Time
ROBOT-2	8	8	4	16	0.01 s
ROBOT-4	64	32	8	32	0.03 s
ROBOT-8	2,024	512	16	64	1.9 s
ROBOT-16	1,048,576	131,072	32	128	*
JOBSHOP-4-6	1,024	16,777,216	*	*	*
JOBSHOP-5-8	8,192	$1.10 \times 10^{12}$	*	*	*
JOBSHOP-6-10	65,536	$1.15 \times 10^{18}$	*	*	*

Table 4.1: Results from the FA-FMDP minimization algorithm using decision trees.

for different  $j$  and  $j'$  are set “yes”, which means we tried to allocate the machine to two different tasks at the same time. Once  $broken_i$  is set “yes”, it never becomes “false”. When a specified final task is done, the reward of 1 is given, but if any of  $broken_i$  is set “yes”, then reward of -10 is given. Figure 4.11 shows a graphical illustration of an example.

**Results** Table 4.1 shows experimental results from the minimization algorithm using decision trees. We show as well as the running time of the algorithm, the number of states and actions of the unminimized MDP, and the number of aggregate states and actions of the minimized MDP. For small domains such as ROBOT-2, ROBOT-4, ROBOT-8, the algorithm was able to find the minimal MDP within a reasonable amount of time.

The algorithm suffered from generating large intermediate partitions for large domains such as ROBOT-16, JOBSHOP-4-6, JOBSHOP-5-6 and JOBSHOP-6-10. Notably, JOBSHOP-5-6 and JOBSHOP-6-10 are randomly generated domains. Each task is a prerequisite of a task with probability 0.1, and each machine is required by each task with probability 0.2. The dependencies between tasks and the requirements of machines are chosen randomly with 0.1. We noticed that success probabilities randomly chosen in the interval  $[0,1]$  lead to combinatorial explosion, since the decision trees and ADDs have to track all the possible combination of success probabilities. Hence, the success probabilities were chosen 1.0 with probability 1/2, and uniformly among 0.95, 0.9, 0.85, 0.8, and 0.5 with probability 1/2.

In these domains, the algorithm was not able to complete within an hour. Although the number of states and actions of the minimized MDP may be small, the algorithm at each iteration generates a potentially large number of refined blocks and then merges the blocks that have the same transition probabilities. The table shows the results using the algorithm with the BACKUP operator; the algorithm with SPLIT operator is not able to find the minimized MDP for any of ROBOT-16, JOBSHOP-4-6, JOBSHOP-5-6 or JOBSHOP-6-10.

For ROBOT- $k$  domains, we can analytically produce the states and actions of the minimized MDP. For example, in ROBOT-4 domain, the aggregate states are

$$\begin{aligned} &(RoomBit_0 = b_0) \wedge (RoomBit_1 = b_1) \wedge (L_{b_0b_1} = \text{off}), \\ &(RoomBit_0 = b_0) \wedge (RoomBit_1 = b_1) \wedge (L_{b_0b_1} = \text{on}) \end{aligned}$$

for  $\forall b_0, b_1, b_2 \in \{0, 1\}$ . Instead of obtaining the aggregate actions by calculating projection  $P|_{\bar{A}}$  of homogeneous partition  $P$ , we calculate the aggregate actions for each aggregate state. In other words, for each aggregate state  $s$ , we collect blocks in  $P$  whose state representation is contained within  $s$ , and then project these blocks onto the action space  $\bar{A}$ . The reason for doing this alternative calculation will follow shortly. We can verify that for aggregate state

$$(RoomBit_0 = b_0) \wedge (RoomBit_1 = b_1) \wedge (L_{b_0b_1} = \text{off}),$$

the aggregate actions are

$$\begin{aligned} &(A_{b_0b_1} = \text{turn\_on}), \\ &(A_{b_0b_1} = \text{turn\_off}), \end{aligned}$$

and for aggregate state

$$(RoomBit_0 = b_0) \wedge (RoomBit_1 = b_1) \wedge (L_{b_0b_1} = \text{on}),$$

the aggregate actions are

$$\begin{aligned} &(M = \text{stay}) \wedge (A_{b_0b_1} = \text{turn\_on}), \\ &(M = \text{stay}) \wedge (A_{b_0b_1} = \text{turn\_off}), \\ &(M = \text{go}) \wedge (A_{b_0b_1+1} = \text{turn\_on}), \\ &(M = \text{go}) \wedge (A_{b_0b_1+1} = \text{turn\_off}) \end{aligned}$$

for  $\forall b_0, b_1 \in \{0, 1\}$ .  $b_0b_1 + 1$  denotes the binary representation of the room to next room  $b_0b_1$ . Unfortunately, since  $P$  is represented as a decision tree, it makes extra refinements in  $P$  and this results in extra refinements in the projections. For example, according to our experiments, the blocks in the homogeneous partition pertinent to room 00 are as follows:

$$\begin{aligned} &(RoomBit_0 = 0) \wedge (RoomBit_1 = 0) \wedge (M = \text{stay}) \wedge (A_{00} = \text{turn\_on}) \\ &(RoomBit_0 = 0) \wedge (RoomBit_1 = 0) \wedge (M = \text{stay}) \wedge (A_{00} = \text{turn\_off}) \\ &(RoomBit_0 = 0) \wedge (RoomBit_1 = 0) \wedge (L_{00} = \text{off}) \wedge (M = \text{go}) \wedge (A_{00} = \text{turn\_on}) \\ &(RoomBit_0 = 0) \wedge (RoomBit_1 = 0) \wedge (L_{00} = \text{on}) \wedge (M = \text{go}) \wedge (A_{01} = \text{turn\_on}) \\ &(RoomBit_0 = 0) \wedge (RoomBit_1 = 0) \wedge (L_{00} = \text{off}) \wedge (M = \text{go}) \wedge (A_{00} = \text{turn\_off}) \\ &(RoomBit_0 = 0) \wedge (RoomBit_1 = 0) \wedge (L_{00} = \text{on}) \wedge (M = \text{go}) \wedge (A_{01} = \text{turn\_off}) \end{aligned}$$

Domain	Number of states	Number of actions	Number of state blocks	Number of action blocks	Running Time
ROBOT-2	8	8	4	12	0.15 s
ROBOT-4	64	32	8	24	0.18 s
ROBOT-8	2,024	512	16	48	1.02 s
ROBOT-16	1,048,576	131,072	32	96	2.49 s
JOBSHOP-4-6	1,024	16,777,216	13	44	3.03 s
JOBSHOP-5-8	8,192	$1.10 \times 10^{12}$	45	109	6.20 s
JOBSHOP-6-10	65,536	$1.15 \times 10^{18}$	57	143	13.68 s

Table 4.2: Results from the FA-FMDP minimization algorithm using ADDs.

Hence, when we project these blocks onto the action space constrained to  $RoomBit = 00$  and  $L_{00} = \text{off}$ , we lose the information that the value of  $M$  does not matter. Thus we end up with 4 aggregate actions for each block. This phenomenon generalizes to every ROBOT- $k$  domain with  $k$  a power of 2.

The reason why we made separate projection onto the action space for each aggregate block comes from an observation similar to the above. There are a few action fluents pertinent to an aggregate block, but as we project onto the action space via existential abstraction, we lose such information. As such, we end up with a large number of actions that we cannot call “aggregate” actions any more. In our experiments, the projection of the homogeneous partition produces the same number of actions as in the unminimized MDP. The number of action blocks shown in the table is the sum of the number of action blocks obtained through projection constrained to each state block.

Although the conventional definition of minimized MDPs does not exactly specify the aggregate actions as the above, we are not violating the definition either. Our representation of minimized MDPs maps the same assignment to action fluents to different aggregate actions depending on the current state block, contrary to the previous approach by Dean *et al.* [33] having a unified mapping regardless of the current state block.

Table 4.2 shows the results from experiments using the minimization algorithm with ADDs. The algorithm can minimize the domains that the decision tree version cannot. Note also that ADDs are able to find the coarsest homogeneous partition, thus we do not observe extra refinements in action blocks. The number of action blocks is counted in the same way as in the minimization algorithm with decision trees. Figure 4.12 shows the algorithm for the projection of the homogeneous partition onto the action space. The projection algorithm constructs ADD for each state block to focus on relevant blocks in the homogeneous partition. Multiplying this ADD with the ADD of the homogeneous partition

1. Input: Partition  $P$  of  $\Omega_{\bar{X}} \times \Omega_{\bar{A}}$ , and  $P|_{\bar{X}}$ , all in ADD representation.
2. For each block  $B \in P|_{\bar{X}}$ ,
  - (a) Construct ADD for representing block  $B$  so that the ADD returns 1 if the assignment of the state fluents belongs to  $B$ , and returns 0 otherwise. Call it  $P_B$ .
  - (b) Calculate  $P'_B = P_B \cdot P$ .
  - (c) Normalize the terminal nodes in  $P'_B$  so that they are numbered from 1 to  $|P'_B|$ .
  - (d) For each state fluent  $X_i$ ,
    - i. Let  $f = X_i \cdot (|P'_B| + 1) + \bar{X}_i$ .
    - ii. Do existential abstraction of  $f \cdot P'_B$  over  $X_i$  and store the result as  $P'_B$ .
    - iii. Normalize the terminal nodes in  $P'_B$  so that they are numbered from 1 to  $|P'_B|$ .
3. Output  $P'_B$  for all  $B$ .

Figure 4.12: The algorithm for projecting the ADD representation of a homogeneous partition onto the action space.

allows us to ignore all irrelevant blocks, which is the same strategy used inside the BACKUP operator.

### 4.1.2 Related Work

The model minimization technique draws on work in automata theory that attempts to account for structure in finite state automata (Hartmanis and Stearns [49]). Specifically, the technique builds on the work of Lee and Yannakakis [73] involving on-line model minimization algorithms, extending their work to handle Markov decision processes with very large state and action spaces. Previous work by Dean and Givan [32] copes with model minimization with very large state spaces only.

Dearden and Boutilier [38] present abstraction strategies for FMDPs as well. Their work is oriented towards finding a small set of good fluents that are relevant to decision making. Once this set of relevant fluents is found, an MDP with state space exponential in the size of the set is constructed and solved. A series of works by Boutilier and others [16, 15, 53, 101] focus on finding a partition in the state space so that all the states in a block have the same value. This leads to a compact representation for value functions.

An extension of the original minimization algorithm for FMDPs by Dean *et al.* [34] and Givan *et al.* [43] presents  $\epsilon$ -homogeneity for finding an approximately homogeneous

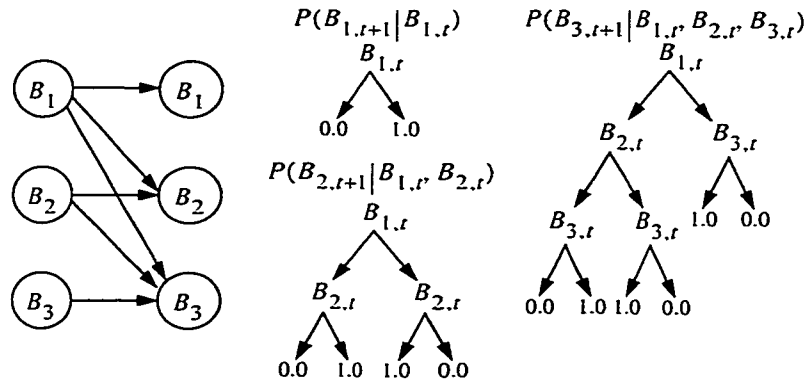


Figure 4.13: An FMDP representing a 3-bit counter. Note that CPTs are still of size polynomial in the number of fluents since they are skewed trees. There is only one action in the FMDP. The size of the coarsest homogeneous partition is  $2^3 = 8$ . We can generalize this example to an  $n$ -bit counter. See Boutilier *et al.* [16] for a similar example with  $n$  actions.

minimal partition. Define the associated equivalence relation that induces partition of the state space by

$$\exists c \in [0, 1] \text{ such that } \forall \vec{x}_t \in C, |T(\vec{x}_t, a, B) - c| \leq \epsilon. \quad (4.3)$$

This is the condition for block  $C$  being  $\epsilon$ -stable with respect to  $B$  and  $a$ . If every block of partition  $P_\epsilon$  is  $\epsilon$ -stable with respect to every other block of  $P_\epsilon$  and every action, we say  $P_\epsilon$  is an  $\epsilon$ -homogeneous partition. Qualitatively speaking, by relaxing the equality to be *approximately equal with error within  $\epsilon$* , we get a smaller partition than that from the original homogeneous refinement algorithm since the  $\epsilon$ -homogeneous refinement algorithm merges approximately behaving blocks although they do not behave the same. The error in the optimal value function computed from  $\epsilon$ -homogeneous partition  $P_\epsilon$  instead of homogeneous partition  $P$  can be obtained using the work of Singh and Yee [99] and White and Eldeib [108]. In terms of complexity, recent work by Goldsmith and Sloan [45] show that given a block in a partition, determining whether it is  $\epsilon$ -stable or not is  $\text{coNP}^{\text{PP}}$ -complete.

## 4.2 Non-homogeneous Aggregation Algorithms

There are two main questions remaining to be answered about the homogeneous refinement techniques for FMDPs<sup>2</sup>. First, what class of functions should we allow as block formulas? If we restrict the representation power of the formulas too much, we may end up with large partitions, although the management of block formulas is easy. If we adopt general boolean

<sup>2</sup>A preliminary work of the material presented here will appear in Kim and Dean [67]

functions, management of block formulas for every SPLIT operation is NP-hard although we can compute the coarsest homogeneous partition, which may be small. Second, what if the coarsest homogeneous partition is still exponentially large compared to the description size of the FMDP? We can indeed build a simple example of such phenomena. Figure 4.13 shows an extreme case in which each block of the coarsest homogeneous partition contains only one atomic state. Using the definition of  $\epsilon$ -stability (Equation 4.3) and  $\epsilon$ -homogeneous partition helps, but it is not hard to show that there are problems with threshold  $\delta$  where the size of the partition is large for  $\epsilon < \delta$  and the partition collapses into a small number of blocks for  $\epsilon = \delta$ . Note that Figure 4.13 also shows this kind of behavior since it is a deterministic domain in which all the transition probabilities are either 0 or 1, and any  $\epsilon \in [0.5, 1]$  collapses the partition into three blocks and any  $\epsilon \in [0, 0.5)$  does not make any change.

Based on the above observation, we present an aggregation algorithm for FMDPs that is not based on the notion of stochastic bisimulation equivalence. We are not interested in finding a minimized MDP that is equivalent to the original FMDP. We are only interested in finding a reasonably sized partition of the state space so that it leads us to a policy close to the optimum.

The algorithm iteratively refines the current partition as the FMDP minimization algorithm does. However, as mentioned above, our criterion for splitting a block is different from making the block stable or obtaining a homogeneous partition. Since we discard the notion of stochastic bisimulation equivalence, a block can be split in many ways. The algorithm examines all the possible splits of a block and selects the best refinement from the current partition. Note that if we allow a general propositional formula for representing blocks in a partition, there are exponentially many ways to split a block. In this work, we assume that the partition is represented as a decision tree. In other words, the block formulas are simple conjunctions. We also restrict our split to be dependent on only one fluent in the FMDP. The number of ways we can split a block in the current partition now becomes  $n - |\vec{X}_B|$  where  $n$  is the total number of fluents in the domain and  $\vec{X}_B$  is the set of relevant fluents that are used to describe block  $B$ . In other words, a block can be split as many ways as the number of fluents not relevant to the block. Now the crucial question becomes, how do we select the best block-fluent pair for split?

To this end, we first define how we construct an MDP from a non-homogeneous partition. In summary, the MDP is an averaged model of the original MDP.

**Definition 4.2.1 (MDP from Non-homogeneous Partition)** *Given an FMDP  $M =$*



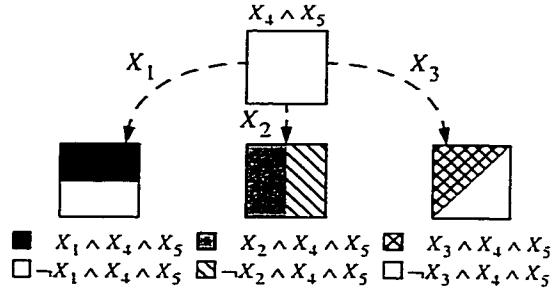


Figure 4.14: Chopping the block  $X_4 \wedge X_5$  with respect to  $X_1$ ,  $X_2$  and  $X_3$ . Note that the blocks resulting from the CHOP operator are not stable in general.

$\{\bar{X}, A, T, I, R\}$  and a non-homogeneous partition  $P$  of the state space  $\Omega_{\bar{X}}$ , the explicit MDP induced by  $P$ , denoted as  $M_P = \{P, A, T_P, I_P, R_P\}$ , is defined as

$$\begin{aligned}
 T_P(C, a, B) &\equiv \frac{1}{|C|} \sum_{\bar{x} \in C, \bar{x}' \in B} T(\bar{x}, a, \bar{x}') \\
 I_P(C) &\equiv \sum_{\bar{x} \in C} I(\bar{x}) \\
 R_P(C, a) &\equiv \frac{1}{|C|} \sum_{\bar{x} \in C} R(\bar{x}, a)
 \end{aligned}$$

for all  $B, C \in P$  and  $a \in A$ .  $V_P^*$  denotes the optimal value function of  $M_P$ , which is a mapping from  $\Omega_{\bar{X}}$  to  $\mathbb{R}$  with the restriction that, for any  $\bar{x}$  and  $\bar{x}'$  in the same block of  $P$ ,  $V_P^*(\bar{x}) = V_P^*(\bar{x}')$  and  $\pi^*(\bar{x}) = \pi^*(\bar{x}')$ .

Given an arbitrary partition of the state space, we argue that this averaged model is the best representative of the original MDP. Since the partition is non-homogeneous, if we examine each state in a block, we make contradictory observations of reward and transition probabilities. The best model that explains the original MDP is the average of these observations.

Given a partition  $P$ , the algorithm selects block  $C$  and fluent  $X$  for generating a refined partition  $P'$ .  $P'$  has the same blocks as  $P$  except for  $C$  which is replaced by  $\text{CHOP}(P, C, X)$ . Figure 4.14 shows how the CHOP operator splits up the block  $C$ . Letting  $n$  be the number of fluents in the domain, we note that there are at most  $n|P|$  different refined partitions that can be obtained by the CHOP operator. For each refined partition  $P'$  generated by the CHOP operator, the algorithm constructs  $M_{P'}$  according to Definition 4.2.1 and calculates the optimal value function  $V_{P'}^*$  of  $M_{P'}$ . How should we find the best refined partition? First, we show that for any partition  $P$ , the optimal value function of  $M_P$ , which we define as  $V_P^*$ , is within a bounded distance from the best approximation to the true value function  $V^*$ .

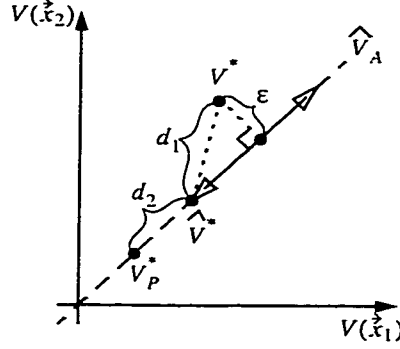


Figure 4.15: Value functions for an MDP constructed from non-homogeneous partition. The figure illustrates an example where the MDP has two states and the partition has one block. The dotted line is the set of representable approximate value functions.  $d_1 \leq 2(1 + \frac{\gamma}{1-\gamma})\epsilon$  and  $d_2 \leq \frac{\|LV_P^* - V_P^*\|_\infty}{1-\gamma}$ .

**Theorem 4.2.1 (Bounded Distance between  $V^*$  and  $V_P^*$ )** Given an FMDP  $M = \{\bar{X}, A, T, I, R\}$  and a partition  $P$  of the state space  $\Omega_{\bar{X}}$ , the optimal value function of  $M$  given as  $V^*$  and the optimal value function of  $M_P$  given as  $V_P^*$  satisfy the bound on the distance

$$\|V^* - V_P^*\|_\infty \leq 2\left(1 + \frac{\gamma}{1-\gamma}\right)\epsilon + \frac{\|LV_P^* - V_P^*\|_\infty}{1-\gamma} \quad (4.4)$$

where letting  $\mathcal{V}_P$  be the set of any mapping  $V_P$  from  $\Omega_{\bar{X}}$  to  $\mathbb{R}$  with the restriction that  $V_P(\bar{x}) = V_P(\bar{x}')$  for any  $\bar{x}$  and  $\bar{x}'$  in the same block of  $P$ ,

$$\epsilon \equiv \min_{V_P \in \mathcal{V}_P} \|V^* - V_P\|_\infty$$

and  $LV_P^*$  is the Bellman backup (Equation 2.5) of  $V_P^*$  defined as

$$LV_P^*(\bar{x}) \equiv \max_a \left[ R(\bar{x}, a) + \gamma \sum_{\bar{x}'} T(\bar{x}, a, \bar{x}') V_P^*(\bar{x}') \right]$$

**Proof** Recall from the definition that  $\forall \bar{x} \in C$ ,

$$V^*(\bar{x}) \equiv \max_a \left[ R(\bar{x}, a) + \gamma \sum_{\bar{x}' \in \bar{X}} T(\bar{x}, a, \bar{x}') V^*(\bar{x}') \right]$$

and

$$V_P^*(\bar{x}) \equiv \max_a \left[ \frac{1}{|C|} \sum_{\bar{x} \in C} R(\bar{x}, a) + \gamma \frac{1}{|C|} \sum_{B \in P} \sum_{\bar{x}' \in B} T(\bar{x}, a, \bar{x}') V_P^*(B) \right].$$

We define another value function

$$\widehat{V}^*(\vec{x}) \equiv \frac{1}{|C|} \sum_{\vec{x} \in C} \max_a [R(\vec{x}, a) + \gamma \sum_{B \in P} \sum_{\vec{x}' \in B} T(\vec{x}, a, \vec{x}') \widehat{V}^*(B)].$$

To calculate the maximum distance between  $V^*$  and  $\widehat{V}^*$ , we define an averager  $A_P$  (Definition 2.4.1) as follows:

$$\widehat{V}(\vec{x}) \equiv (A_P V)(\vec{x}) = \sum_{\vec{x}' \in C} \frac{1}{|C|} V(\vec{x}') \text{ for } \forall C \in P, \forall \vec{x} \in C.$$

Note that the fixed points of  $A_P$  are vectors with the constraint that the values of two components are the same if these two components belong to the same block. For the example shown in Figure 4.15, the straight dashed line satisfying  $\widehat{V}_A(\vec{x}_1) = \widehat{V}_A(\vec{x}_2)$  is the set of fixed points for  $A_P$  if we assume that  $\vec{x}_1$  and  $\vec{x}_2$  belong to the same block. More formally, any fixed point  $\widehat{V}_A$  satisfies

$$\widehat{V}_A(\vec{x}) = \widehat{V}_A(\vec{x}') \text{ if } \exists C \in P \text{ such that } \vec{x} \in C \text{ and } \vec{x}' \in C.$$

Since  $P$  is a refinement of the reward partition,  $\widehat{V}^*$  is the fixed point of the mapping  $L \circ A_P$ , and from Theorem 2.4.1, we have that

$$\|\widehat{V}^* - V^*\|_\infty \leq 2(1 + \frac{\gamma}{1-\gamma})\epsilon \quad (4.5)$$

where  $\epsilon$  is the distance between  $V^*$  and the closest fixed point of  $A_P$ .

The distance between  $\widehat{V}^*$  and  $V_P^*$  is calculated as follows: For any  $\vec{x} \in C$ ,

$$\begin{aligned} |\widehat{V}^*(\vec{x}) - V_P^*(\vec{x})| &= \left| \frac{1}{|C|} \sum_{\vec{x} \in C} \max_a [R(\vec{x}, a) + \gamma \sum_{B \in P} \sum_{\vec{x}' \in B} T(\vec{x}, a, \vec{x}') \widehat{V}^*(B)] \right. \\ &\quad - \frac{1}{|C|} \sum_{\vec{x} \in C} \max_a [R(\vec{x}, a) + \gamma \sum_{B \in P} \sum_{\vec{x}' \in B} T(\vec{x}, a, \vec{x}') V_P^*(B)] \\ &\quad + \frac{1}{|C|} \sum_{\vec{x} \in C} \max_a [R(\vec{x}, a) + \gamma \sum_{B \in P} \sum_{\vec{x}' \in B} T(\vec{x}, a, \vec{x}') V_P^*(B)] \\ &\quad \left. - \max_a \left[ \frac{1}{|C|} \sum_{\vec{x} \in C} R(\vec{x}, a) + \gamma \frac{1}{|C|} \sum_{B \in P} \sum_{\vec{x}' \in B} T(\vec{x}, a, \vec{x}') V_P^*(B) \right] \right| \\ &\leq \gamma \|\widehat{V}^* - V_P^*\|_\infty + \|(A_P \circ L)V_P^* - V_P^*\|_\infty \\ &= \gamma \|\widehat{V}^* - V_P^*\|_\infty + \|LV_P^* - V_P^*\|_\infty \end{aligned}$$

Since the above inequality holds for  $\forall C \in P$ , we have

$$\|\widehat{V}^* - V_P^*\|_\infty \leq \frac{\|LV_P^* - V_P^*\|_\infty}{1-\gamma} \quad (4.6)$$

1. Input: Threshold  $\xi$ , initial partition  $P$  of the state space (reward partition), and the optimal value function  $V_P^*$  for  $M_P$ .
2. For each block  $C \in P$  and fluent  $X_i, 1 \leq i \leq n$ , compute  $M_{P'}$  in which  $P'$  is the same as  $P$  except  $C$  is replaced by  $\text{CHOP}(P, C, X_i)$ .
3. For each  $P'$ , compute  $V_{P'}^*$  and then select  $P^* \equiv \arg \max_{P'} \|V_{P'}^* - V_P^*\|$ .
4. If  $\|V_{P^*}^* - V_P^*\| \leq \xi$ , output  $\pi_{P^*}^*$  and halt.
5. Set  $P = P^*$  and goto step 2.

Figure 4.16: The algorithm for finding a non-homogeneous partition for an approximately optimal policy

By combining Equation 4.5 and Equation 4.6, we have shown that

$$\begin{aligned} \|V^* - V_P^*\|_\infty &\leq \|V^* - \widehat{V}^*\|_\infty + \|\widehat{V}^* - V_P^*\|_\infty \\ &\leq 2\left(1 + \frac{\gamma}{1-\gamma}\right)\epsilon + \frac{\|LV_P^* - V_P^*\|_\infty}{1-\gamma} \end{aligned}$$

□

There are two remarks on the bound in the above theorem.

First,  $\epsilon \equiv \min_{V_P} \|V^* - V_P\|_\infty$  that appears in the first additive term in Equation 4.4 measures how fine the partition  $P$  is.  $\epsilon$  is the minimum error that we can achieve by arbitrarily assigning the values to the blocks to be as close as possible to the true optimal value function  $V^*$ . Thus, we naturally expect that a finer partition will achieve a smaller  $\epsilon$ , although it depends on how the state space is partitioned. At least, given a partition  $P$  and its refinement  $P' \subseteq P$ , and letting  $\epsilon_P$  and  $\epsilon_{P'}$  be the  $\epsilon$  of  $P$  and  $P'$  respectively, we can say that  $\epsilon_P \geq \epsilon_{P'}$ . Note also that  $\epsilon$  becomes 0 for a homogeneous partition.

Second,  $\|LV_P^* - V_P^*\|_\infty$  in the second additive term also vanishes as the partition becomes finer. Although there is no guarantee that the error is monotonically smaller for a finer partition and large for a coarse partition, at least this error serves as an upper bound on the distance.

Given that the calculated value function  $V_P^*$  is a good approximation to the original value function  $V^*$  (in the sense that the bound on the distance becomes smaller as the partition  $P$  becomes finer), the algorithm selects the best refined partition given by formula

$$\arg \max_{P'} \|V_{P'}^* - V_P^*\|_\infty.$$

The algorithm runs iteratively refining the partition until the difference between the optimal value functions of consecutive refinements is below a threshold. Figure 4.16 shows the

algorithm. Note that we can accelerate the algorithm if we can find a good heuristic for selecting the best block-fluent pair in steps 2 and 3. In Section 4.2.1, we show some experimental results using a heuristic approach rather than constructing and solving  $M_{P'}$  for each block-fluent pair.

Note that constructing  $M_{P'}$  from the refined partition  $P'$  can be done efficiently by reusing the parameters from  $M_P$ . The new transition probability matrix  $T_{P'}$  has a lot of components that are the same as those of  $T_P$ . By reusing the components already calculated, we can construct  $M_{P'}$ s without computing transition probabilities and rewards for all blocks. Assume that we perform  $\text{CHOP}(P, C, X_i)$  so that the block  $C$  splits into  $|\Omega_{X_i}|$  blocks,  $\{C_1, \dots, C_{|\Omega_{X_i}|}\}$ . Then we have

$$T_{P'}(B, a, B') = T_P(B, a, B'), \quad \forall a \in A, \forall B \neq C, \forall B' \neq C.$$

All we have to do is to recompute some of the transition probabilities in  $P'$  which are

$$T_{P'}(B, a, B') \quad \forall a \in A, \text{ and for all } B \in \{C_1, \dots, C_{|\Omega_{X_i}|}\} \text{ or } B' \in \{C_1, \dots, C_{|\Omega_{X_i}|}\}.$$

Using decision trees or ADDs, we can calculate new transition probabilities without explicitly enumerating all the states in  $C$ . The calculation is done in a similar manner as the SPLIT operator (Figure 4.1) provided in the previous section. Figure 4.17 shows the algorithm for calculating the new transition probabilities of  $M_{P'}$ . The core operation is the intersection of the partitions. Note that, when calculating  $T_{P'}(B, a, B')$ , all the intersections of the partitions are also intersected with the block  $B$ . When we use decision trees and graft CPTs to calculate the intersection, we can eliminate the subtrees in CPTs that do not intersect with the block  $B$ . For example, in Figure 4.18, we can eliminate the left subtree of  $X_1$  in  $P_{X_1}$ , and the left subtree of  $X_2$  in  $P_{X_2}$  since the block representation of  $B$  tells us that  $X_1$  and  $X_2$  should be false. Assuming that the block being split is  $B$ , the size of a block represented by a terminal node  $v$  in the resulting decision tree is determined by

$$\frac{\prod_{X \notin \vec{X}_B} |\Omega_X|}{\prod_{Y \in \vec{X}_v} |\Omega_Y|}$$

where  $\vec{X}_B$  is the set of relevant fluents used to describe the block  $B$  and  $\vec{X}_v$  is the set of fluents that appear on the path from the root of the tree to the terminal node  $v$ . When we use ADDs, the intersection is done by the multiplication of ADDs. The size of a block represented by a terminal node in the resulting ADD is determined by first constructing the ADD that represents the block (1 if the assignment of the fluents belongs to the block, and 0 otherwise) and summing over all fluents in the FMDP.

1. Input: The original partition  $P$  and the refined partition  $P'$  such that  $P'$  is the same as  $P$  except the block  $C \in P$  is replaced by  $\text{CHOP}(P, C, X_i) = \{C_1, \dots, C_{|\Omega_{X_i}|}\}$ .
2. For each action  $a$ , and each pair of blocks  $B$  and  $B'$  in partition  $P'$ , we calculate the transition probability  $T_{P'}(B, a, B')$  as follows:

- (a) If  $B \notin \{C_1, \dots, C_{|\Omega_{X_i}|}\}$  and  $B' \notin \{C_1, \dots, C_{|\Omega_{X_i}|}\}$ , the transition probability is the same as  $T_P$ :

$$T_{P'}(B, a, B') = T_P(B, a, B')$$

- (b) If  $B \notin \{C_1, \dots, C_{|\Omega_{X_i}|}\}$  and  $B' = C_i$  for some  $i$ ,

- i. Calculate  $B \cap \bigcap_{X \in \bar{X}_{C_i}} P_X$  where  $\bar{X}_{C_i}$  is the set of relevant fluents for block  $C_i$  and  $P_X$  is the partition induced by the CPT of fluent  $X$ . As was the case with SPLIT operator,  $\bar{X}_{C_i}$  is the set of fluents used in the description of block  $C_i$ .

- ii. From the result obtained in the previous step, average the probabilities contained in the split blocks weighted by the sizes of the blocks.

- (c) If  $B = C_i$  for some  $i$  and  $B' \notin \{C_1, \dots, C_{|\Omega_{X_i}|}\}$ ,

- i. Calculate  $C_i \cap \bigcap_{X \in \bar{X}_{B'}} P_X$ .

- ii. From the result obtained in the previous step, average the probabilities contained in the split blocks weighted by the sizes of the blocks.

- (d) If  $B = C_i$  for some  $i$  and  $B' = C_j$  for some  $j$ ,

- i. Calculate  $C_i \cap \bigcap_{X \in \bar{X}_{C_j}} P_X$ .

- ii. From the result obtained in the previous step, average the probabilities contained in the split blocks weighted by the sizes of the blocks.

3. Output  $T_{P'}$ .

Figure 4.17: The algorithm for calculating  $T_{P'}$

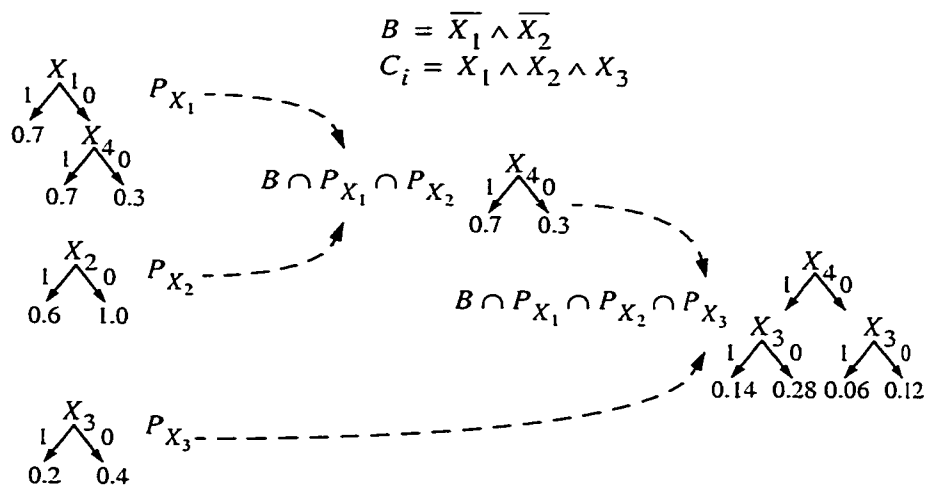


Figure 4.18: Calculating  $T_{P'}(B, a, C_i)$  with decision trees.

### 4.2.1 Experiments

The test problems used in our experiments are adopted from Hoey *et al.* [53] and involve domains with 6 to 17 binary fluents. The initial probabilities are given as a uniform distribution. For each domain, we show the performance of the policy derived from the non-homogeneous partition and the cumulative elapsed time at each iteration. We used ILOG CPLEX 6.5 for calculating optimal value functions of aggregate MDPs constructed from non-homogeneous partitions. For evaluating the policies and calculating the optimal value functions of the original FMDP, we use the CUDD package (Somenzi [100]) to implement structured versions of iterative algorithms similar to SPUDD (Hoey *et al.* [53]).

Figure 4.19, Figure 4.20, Figure 4.21, and Figure 4.22 illustrate the performance of the algorithm shown in Figure 4.16. We ran the algorithm for 100 iterations, except for the COFFEE domain (only 64 states). For each figure, the graph on the left side shows the actual performance (actual value of the optimal policy calculated from the aggregated MDP) and the heuristic value (the optimal value of the aggregated MDP) after each iteration. The graph on the right side shows the plot of cumulative elapsed time (in seconds) after each iteration. Note that in some domains, there is a small gap between the actual performance and the heuristic value even when the policy from the aggregated MDP reached the optimal performance. This is due to the fact that the evaluation is done through an iterative method with the stopping condition  $\|V_{i+1} - V_i\| \leq \delta$ , which we set  $\delta$  to 0.01.

The optimal value function of the EXPON domain (Figure 4.21) has thousands of internal nodes even when represented as an ADD. The non-homogeneous partitioning algorithm

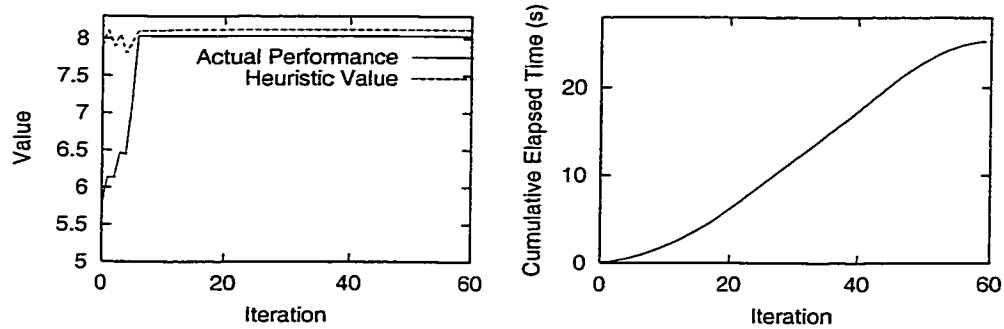


Figure 4.19: COFFEE domain (6 variables, 64 states). The non-homogeneous partitioning algorithm found the optimal policy after 6 splits, totaling 10 blocks in the partition. The optimal value is 8.11727.

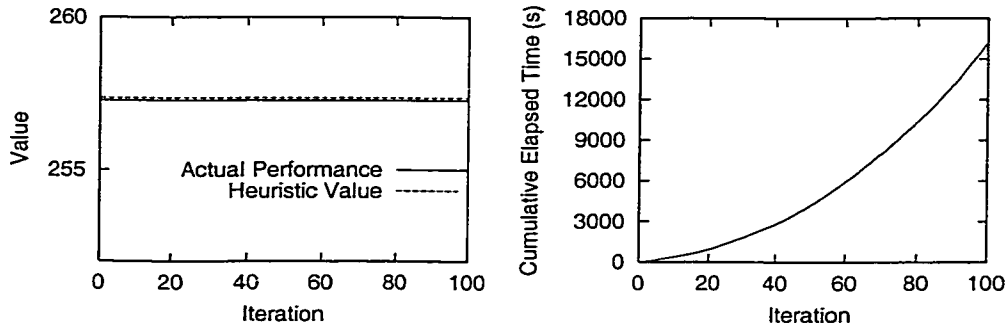


Figure 4.20: LINEAR domain (14 variables, 16,384 states). The optimal policy of the aggregate MDP from the reward partition yields the optimal value, *i.e.*, the optimal policy was obtained without any split. The optimal value is 257.356.

is not able to find the optimal policy with partition size less than or equal to 113, however, the policy at the end of 100th iteration performs 10 times better than the initial policy from the aggregate MDP induced by the reward partition. The size of the ADD representation for the optimal value function is also quite large in the case of the FACTORY domain (Figure 4.22). After 100 splits, totaling 126 blocks, the policy from the aggregated MDP has a value of 24.8 which is 87% of the optimal value.

We also experimented on the LINEAR domain (14 variables, 16,384 states). The optimal policy of the aggregate MDP from the reward partition yields the optimal value, *i.e.*, the optimal policy was obtained without any split. It takes 23.28 seconds for 1 iteration. When the algorithm finds out that every refinement  $P'$  of the current partition  $P$  yields  $\|V_{P'}^* - V_P^*\| = 0$ , then it knows the current optimal policy from the aggregated MDP is indeed optimal. We ran the algorithm for 100 iterations to confirm the slow increase of running time between the consecutive iterations. Meanwhile, our implementation of SPUDD, which



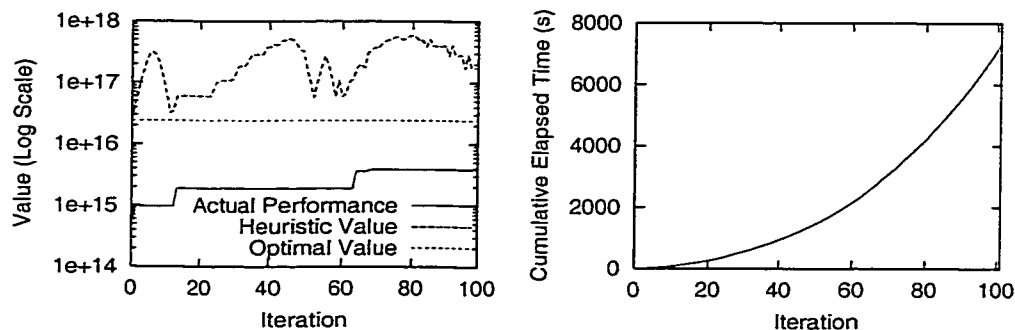


Figure 4.21: EXPON domain (12 variables, 4096 states). After 100 splits (113 blocks), the non-homogeneous partitioning algorithm yields an approximately optimal policy. The optimal value is  $2.4 \times 10^{16}$ .

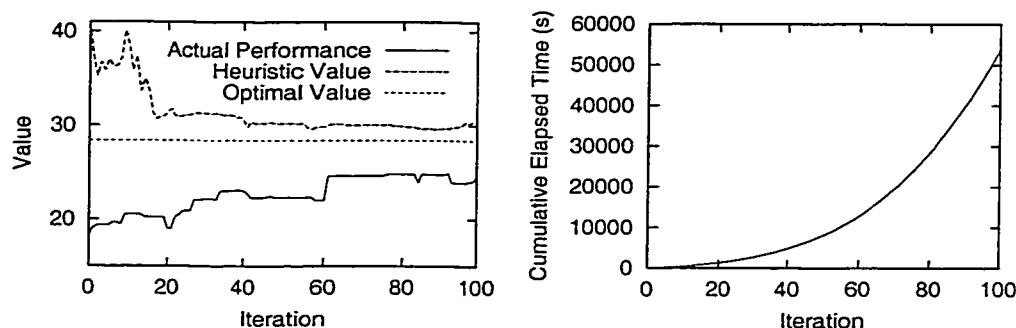


Figure 4.22: FACTORY domain (17 variables, 131,072 states). After 100 splits (126 blocks), the non-homogeneous partitioning algorithm yields an approximately optimal policy. The optimal value is 28.4.

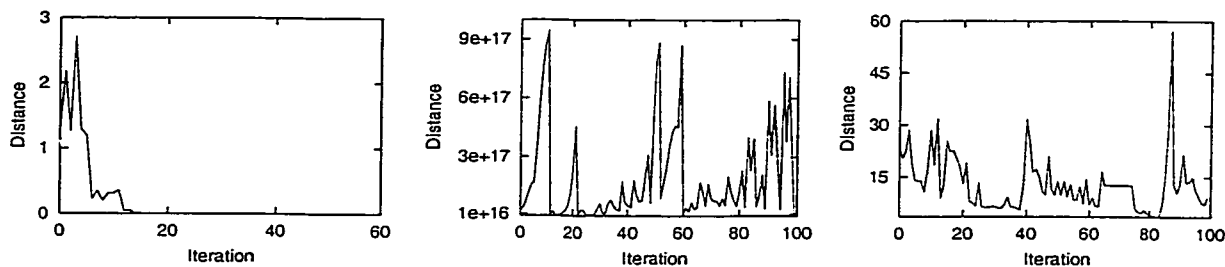


Figure 4.23: Distance plots between two value functions of successive aggregate MDPs ( $\|V_P^* - V_{P'}^*\|$  in Figure 4.16). From left to right: COFFEE, EXPON, FACTORY.

is not fully optimized, takes 43.58 seconds and produces 15 terminal nodes.

Figure 4.23 shows the distances between two value functions of aggregate MDPs in consecutive iterations for each domain. We excluded the LINEAR domain since the distances were zero from the onset. The distance graphs show how the heuristic function change over

time. Such a distance is not a good measure as a stopping criterion, but it reveals useful information on the policy from a non-homogeneous partition. Note that in the COFFEE domain, the distance decreases sharply after the actual performance of the aggregate MDP policy reaches the optimal. However, the distances after that are not zero, which implies the partition is still non-homogeneous. We do not observe such behavior in the EXPON and FACTORY domain since the algorithm was not able to find the optimal policy.

These experimental results are collected from the non-homogeneous aggregation algorithm that examines every block-fluent pair before each refinement. The ILOG CPLEX 6.5 allows us to efficiently compute a new solution when the new problem is slightly modified. Despite such optimization, constructing LP for every block-fluent pair is an exhaustive task due to a large number of block-fluent pairs. The work by Munos and Moore [86] provides us with some ideas for how we might select a good block-fluent pair without examining every pair. Although their work concerns refining the discretization of continuous MDPs, their sophisticated heuristic for deciding where to split can be applied to our problem as well. They introduce two measures that indicate which parts of the discretized state space are most important to allocate additional computational resources for gathering more data, and these measures naturally apply to splitting blocks in FMDPs:

- The *influence* of a state  $s$  on another state  $s'$  is a measure of how much the state  $s$  contributes to the value function of the state  $s'$ . It is calculated by

$$I(s, s') = \sum_{s'' \in S_P} \gamma T_P(s, \pi_P^*(s), s'') I(s'', s') + \begin{cases} 1 & \text{if } s = s' \\ 0 & \text{if } s \neq s' \end{cases}$$

where  $S_P$  and  $T_P$  are the state space and the transition probabilities of  $M_P$ , the MDP constructed from non-homogeneous partition  $P$ , and  $\pi_P^*$  is the optimal policy of  $M_P$ .

- The *variance* of a state  $s$  is a measure of how uncertain we are about the value function. It is calculated by

$$\sigma^2(s) = e(s) + \gamma^2 \sum_{s' \in S_P} T_P(s, \pi_P^*(s), s') \sigma^2(s')$$

where

$$e(s) = \sum_{s' \in S_P} T(s, \pi_P^*(s), s') [\gamma V_P(s') - V_P(s) + R_P(s)]^2.$$

Note that the states used in the above definitions are indeed blocks. The overall measure for splitting a block is defined by

$$Stdev\_Inf(s) = \sigma(s) \sum_{s' \in S_d} I(s, s')$$

where  $S_d$  is the set of states obtained as follows: let a pessimistic optimal policy be a policy which assumes that the actual transition probabilities are unfavorable to the decision making agent, and the optimistic optimal policy be a policy which assumes the contrary.  $S_d$  is the set of states in which the actions taken by the pessimistic optimal policy and the optimistic optimal policy differ, using the techniques in bounded-parameter MDPs (Givan *et al.* [44]) and MDPs with imprecise parameters (White and Eldeib [108]).

Formally, let  $T_{\downarrow}(s, a, s')$  be the minimum among the individual transition probabilities for originating from a state in the block represented by  $s$  and arriving at a state in the block represented by  $s'$  when executing action  $a$ . Let  $T_{\uparrow}(s, a, s')$  be the maximum among the individual transition probabilities of originating from a state in the block represented by  $s$  and arriving at a state in the block represented by  $s'$  when executing action  $a$ .  $T_{\downarrow}(s, a, s')$  and  $T_{\uparrow}(s, a, s')$  are obtained during the calculation of  $T_P(s, a, s')$  (See Figure 4.17). The pessimistic optimal policy is calculated from the pessimistic optimal value function, which is obtained by the iteration

$$V_{\text{pes}}^{(n+1)}(s) = \max_a \left[ R(s) + \gamma \min_{T(s,a,\cdot)} \sum_{s'} T(s, a, s') V_{\text{pes}}^{(n)}(s') \right]$$

where the minimization is done by choosing probabilities  $T(s, a, s')$  that are between the minimum and the maximum of the individual transition probabilities:

$$T_{\downarrow}(s, a, s') \leq T(s, a, s') \leq T_{\uparrow}(s, a, s') \forall s',$$

$$\sum_{s'} T(s, a, s') = 1.$$

The optimistic optimal policy is calculated by doing the contrary:

$$V_{\text{opt}}^{(n+1)}(s) = \max_a \left[ R(s) + \gamma \max_{T(s,a,\cdot)} \sum_{s'} T(s, a, s') V_{\text{opt}}^{(n)}(s') \right].$$

At each iteration, we select the block which has the largest *Stdev\_Inf* and examine splitting the block with respect to each fluent. We select the best fluent by constructing the MDP and comparing the value functions as the previous version of the algorithm. This

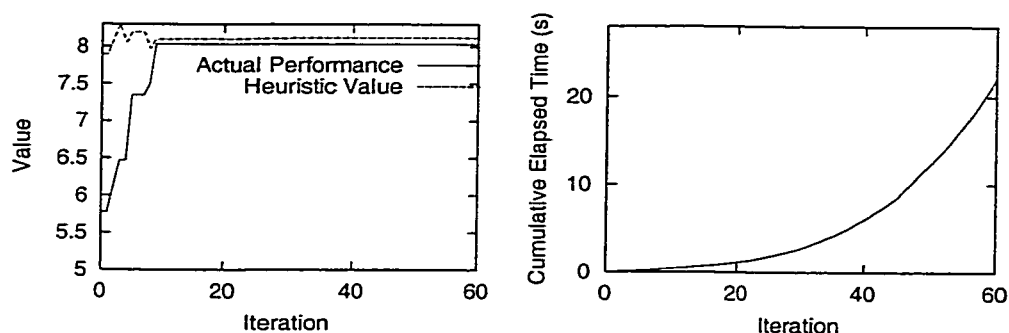


Figure 4.24: Munos and Moore's heuristic on COFFEE domain (6 variables, 64 states). The optimal value is 8.11727.

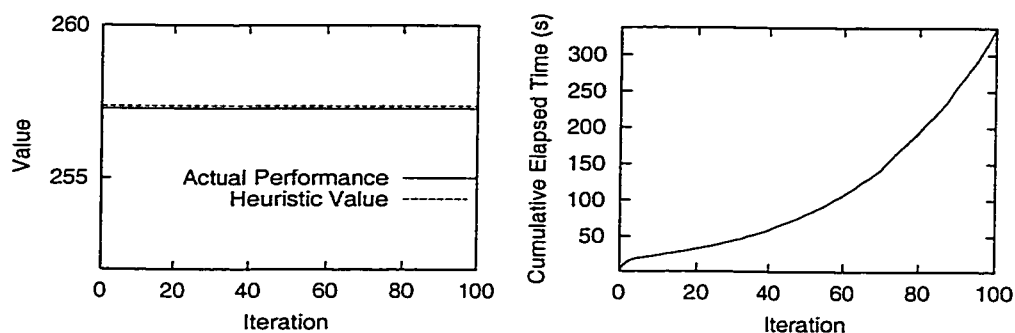


Figure 4.25: Munos and Moore's heuristic on LINEAR domain (14 variables, 16,384 states). The optimal value is 257.356.

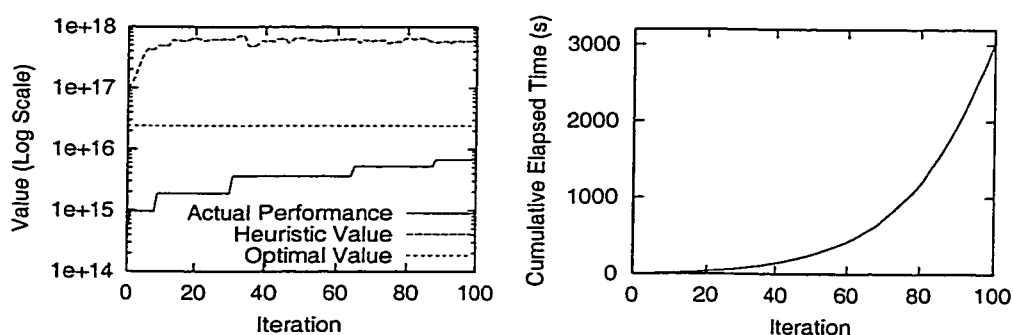


Figure 4.26: Munos and Moore's heuristic on EXPON domain (12 variables, 4096 states). The optimal value is  $2.4 \times 10^{16}$ .

strategy gives us a significant speed up, but shows some loss in performance. We show the results in Figure 4.24, Figure 4.25, Figure 4.26, and Figure 4.27.

As the basis for comparison of the heuristics we have shown, we also show the experimental results from random partitioning method. At each iteration, this method randomly

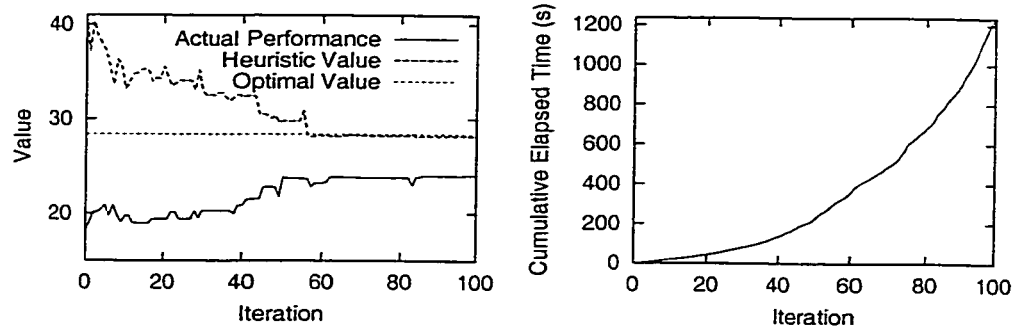


Figure 4.27: Munos and Moore’s heuristic on FACTORY domain (17 variables, 131,072 states). The optimal value is 28.4.

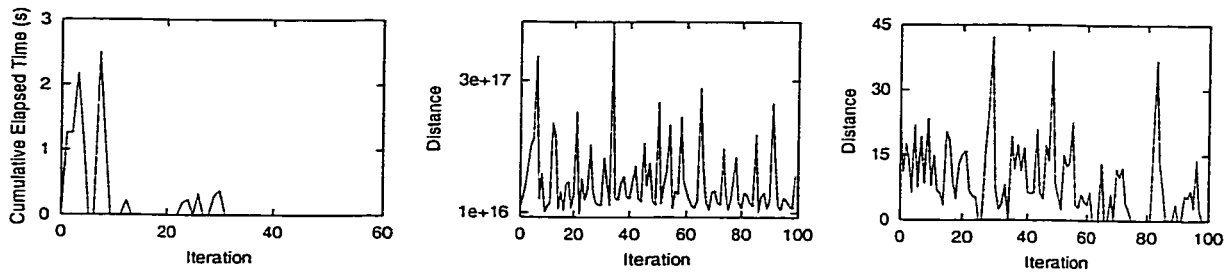


Figure 4.28: Distance plots between two value functions of successive aggregate MDPs ( $\|V_{P^*} - V_P^*\|$  in Figure 4.16). From left to right: COFFEE, EXPON, FACTORY.

	EXPON		FACTORY	
	Performance	Time	Performance	Time
Best	16.0 %	7333 s	87.3 %	55402 s
Munos & Moore	28.0 %	3169 s	84.6 %	1260 s
Random	7.9 %	38 s	69.5 %	65 s

Table 4.3: Summary of results after 100 iterations on EXPON and FACTORY.

selects the next refinement. Figure 4.29 shows the performance of random partitioning method on the same domains. Table 4.3 summarizes the results of non-homogeneous aggregation methods.

We also compared the performance of non-homogeneous partitioning algorithm to APRICODD [101]. Table 4.4 and Table 4.5 summarize the comparison of the two algorithms on the larger domains, EXPON and FACTORY. By trial and error, we tuned the pruning parameter of the APRICODD algorithm so that the size of the approximate value function from the APRICODD algorithm is comparable to that of the non-homogeneous partition at the end of 100 iterations. We used “sliding-tolerance” pruning technique with different parameters. We allow two to three times more nodes in the decision diagrams than the number

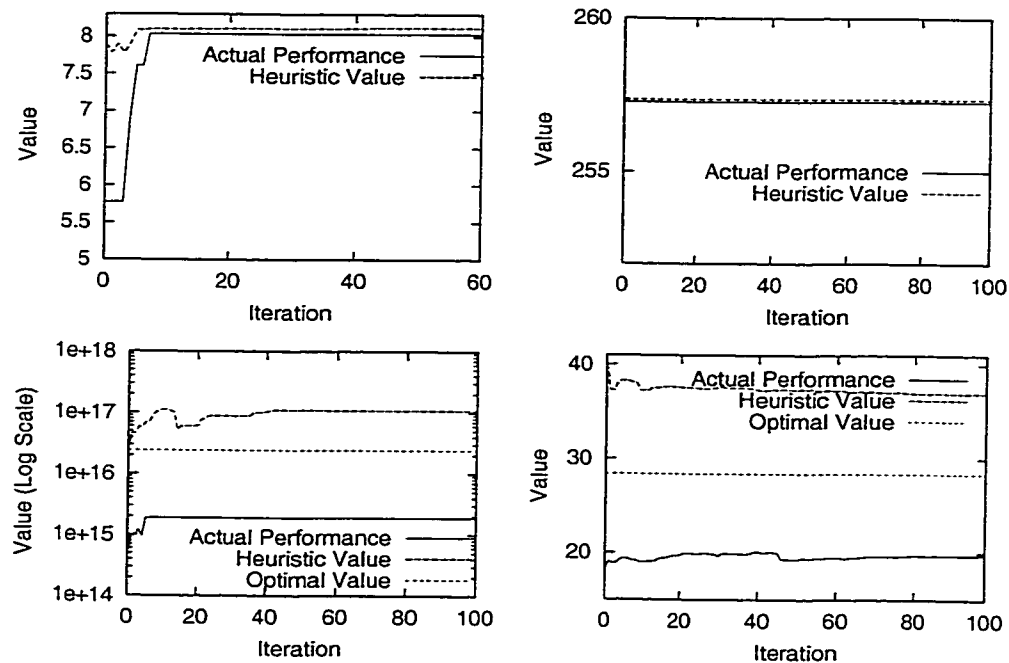


Figure 4.29: Random partitioning on COFFEE (upper-left), LINEAR (upper-right), EXPON (lower-left) and FACTORY (lower-right).

of blocks in the non-homogeneous partition. Often, increasing the pruning parameter so that we get smaller value functions from APRICODD results in non-converging behavior. In fact, APRICODD on the EXPON domain with pruning parameter 0.5 does not converge. In this case, we stopped the algorithm when the value functions between consecutive iterations were oscillating (500 iterations). Note that we are still allowing the APRICODD to search in the space of a much richer representation for value functions — an ADD with 246 nodes can represent a much finer partition than a partition with 113 blocks. In terms of the number of blocks, the non-homogeneous partitioning algorithm performs comparable to the APRICODD algorithm. Note also that our implementation of APRICODD is not fully optimized, and that the running times may be significantly greater than those reported by the original authors.

The above experiments show two advantages of using the non-homogeneous aggregation algorithm. First, while in some domains the coarsest homogeneous partition may be quite large, it may not be critical to compute a homogeneous partition to obtain an optimal (or near optimal) policy. We can obtain the optimal value function without computing a homogeneous partition, but the representation of the optimal value function can be large, too. In such domains, structured algorithms such as FMDP minimization, SPUDD or APRICODD

	Perf.	Blocks		Time
Non-homogeneous (Best)	16 %	113		7333 s
Non-homogeneous (Munos & Moore)	28 %	113		3169 s
	Perf.	Nodes	Leaves	Time
APRICODD (0.4)	48 %	320	65	630 s
APRICODD (0.5)	23 %	246	33	73 s

Table 4.4: Comparison of the non-homogeneous partitioning algorithm and the APRICODD algorithm on EXPON domain. The number inside the parentheses is the pruning parameter determined by “sliding-tolerance” pruning. “Perf” is the ratio between the performance of the optimal policy and that of the approximate policy assuming uniform starting distribution on states. We also present the number of nodes and leaves (terminal nodes) in the ADD representation of the approximate value function from the APRICODD.

	Perf.	Blocks		Time
Non-homogeneous (Best)	87 %	126		55402 s
Non-homogeneous (Munos & Moore)	85 %	126		1260 s
	Perf.	Nodes	Leaves	Time
APRICODD (0.2)	67 %	342	73	920 s
APRICODD (0.3)	26 %	252	65	893 s

Table 4.5: Comparison of the non-homogeneous partitioning algorithm and the APRICODD algorithm on the FACTORY domain.

may incur a large cost in representing an optimal or near optimal value function, whereas the non-homogeneous aggregation algorithm does not necessarily face such a problem. Second, the algorithm provides a new approach to finding an approximately optimal policy, which allows us to specify the desired size of the policy *a priori*.

#### 4.2.2 Related Work

The algorithm by Bertsekas and Castañon [11] is perhaps the closest work to ours in the literature. However, their aggregation algorithm assumes that the states are explicitly enumerable, and works directly on approximating the current estimate to the true value function. In contrast to our algorithm which generates refinements at each iteration, their algorithm disregards the partition from the previous iteration. Once we have structured methods that implement the core parts of their algorithm, applying their algorithm to FMDPs seems promising. In the area of reinforcement learning, G-learning by Chapman and Kaelbling [30] and U-Tree algorithm by McCallum [79] use decision trees to aggregate the states. Since these algorithms assume no prior knowledge of the transition probabilities and the rewards, they first collect the samples of probabilities and rewards and do statistical

test to determine whether we should differentiate the parts of the state space. In contrast, our framework has prior knowledge of exact probabilities and rewards, but examining every one of them for each state is prohibitive due to the huge state space.

The use of an averaged model in reinforcement learning is called the indirect method, meaning that we first collect the samples of the transition probabilities and rewards to construct the model of the domain and then solve the domain using the model (Bradtke and Barto [23], Kearns and Singh [65], Boyan [19]). Since indirect methods also assume no initial knowledge of the domain, they assume explicitly enumerable number of states since it requires a number of samples for each state to construct a reliable model.

The algorithm by Horsch and Poole [55] proposes a similar strategy for solving multi-stage influence diagrams. They present an anytime algorithm that refines a tree-structured optimal policy for the influence diagram at each iteration. The anytime behavior of the algorithm is credited to the fact that it does exact computation at each iteration. Our algorithm avoids the time-consuming exact computation by constructing the aggregate MDP, the policy from which *generally* improves throughout the iterations since the bound on the performance gets tighter.

### 4.3 Decomposition Algorithms

Another technique for solving FMDPs is *decomposition* (Boutilier *et al.* [13], Dean and Lin [36], Singh and Cohn [97])<sup>3</sup>. The main idea is to divide the original MDP into small pieces and solve them separately. For example, consider a disaster relief scenario where the airplanes have to be dispatched to a number of disaster sites to supply relief packages. This problem may be formulated as an MDP with a large state space, but such an MDP may be intractable to solve. Using decomposition, we focus on individual sub-problems for each site, such as deciding how we allocate airplanes and relief packages for each site without considering other sites. We refer these sub-problems as sub-MDPs when formulated as MDPs. Note that in general the solutions to sub-problems does not solve the original MDP since they are obtained without some of the constraints — the number of airplanes or relief packages is limited and we may not be able to address all the disaster sites at certain times. The key to the success of decomposition techniques lies in determining how we decompose the original problem into sub-problems and how we efficiently combine the sub-solution to

---

<sup>3</sup>The material presented here is based on the work done in collaboration with Nicolas Meuleau, Milos Hauskrecht, Leonid Peshkin, Thomas Dean, Leslie Kaelbling and Craig Boutilier, and presented in Meuleau *et al.* [81]



obtain a solution to the original problem.

Each sub-MDP represents a sub-task to perform. A number of different tasks must be addressed and actions consist of assigning various resources at different times to each of these sub-tasks. We assume that these tasks are *additive utility independent* (Keeney and Raiffa [66]): the utility of achieving any collection of sub-tasks is the sum of rewards associated with each sub-task. This assumption is quite natural in the disaster relief domain which we later provide the precise definition — each disaster site has a (different) reward of being relieved, and the goal is to maximize the sum of rewards.

Among various planning problems especially suitable for decomposition are resource allocation problems. Similar to FA-FMDPs, these problems typically have extremely large action spaces and an assignment of resources to one task has no effect on any other tasks. In the disaster relief domain, the action is the allocation of relief package and airplane to each site. The action space is the cross product of all possible allocations of packages and airplanes. Also, allocating more airplanes and relief packages to site  $i$  does not affect the status of site  $j$  ( $j \neq i$ ), given that we do not deallocate any airplane or relief package for site  $j$ . This means that each task can be viewed as an independent subprocess whose rewards and transitions are independent of the others, given a fixed resource assignment policy.

Such an independence does not generally make it easy to find a global optimal policy. Resources are usually constrained, so the allocation of resources to one task at a given point in time restricts the actions available for others at every point in time. Thus the resource allocation problem is still a complex optimization problem. For the disaster relief domain, the airplanes and the relief packages are limited, so we have to be careful when sacrificing a site — although we may want to allocate more resources to a site with higher reward by deallocating some resources from a site with lower reward, we may end up not relieving either site. However, if there are absolutely no resource constraints, the processes are completely independent. They can be solved individually, and an optimal global solution is determined by the concurrent execution of the optimal local policies. With resource constraints, however, optimal solutions to individual sub-MDPs can be easily computed, but combining them to calculate the policy for the whole problem is non-trivial since these individual sub-solutions are calculated with the assumption that the all resources are exclusively available to the sub-task.

In this section, we present heuristic algorithms for solving stochastic resource allocation problems. These algorithms exploit the fact that these problems have independent sub-problems given an infinite amount of resource, and heuristically combine optimal policies for sub-MDPs to obtain approximately optimal policy for the whole problem. We assume

a special form of FA-FMDP suitable for modeling these problems, which we define as a Markov task set (MTS):

**Definition 4.3.1 (Markov Task Set)** *A Markov task set (MTS) of  $n$  tasks is defined as  $M = \{\bar{X}, \bar{A}, \bar{T}, \bar{R}, C, L, G\}$  where*

- $\bar{X} = [X_1, \dots, X_n]$  is the set of state fluents where each  $X_i$  is the set of primitive states of Markov task  $i$ .
- $\bar{A} = [A_1, \dots, A_n]$  is the set of action fluents where each  $A_i$  is a set of integers from 0 to some limit, describing the allocation of an amount of resource to task  $i$ .
- $\bar{T} = [T_1, \dots, T_n]$  is the set of transition probability distributions where each  $T_i$  is defined as

$$T_i(x_{i,t}, a_i, x_{i,t+1}) = P(x_{i,t+1} | x_{i,t}, a_i)$$

where  $x_{i,t}$  and  $x_{i,t+1}$  denote the state of task  $i$  at time  $t$  and  $t + 1$ , respectively. Note that the transition probability for task  $i$  is independent from those for all other tasks.

- $\bar{R} = [R_1, \dots, R_n]$  is the set of reward functions where each  $R_i(t) : X_{i,t} \times A_{i,t} \times X_{i,t} \times t \rightarrow \mathfrak{R}$  specifies the reward conditional on the starting state, executed action, and resulting state at each time  $t$ .
- $C$  is the cost for using a single unit of resource.
- $G$  is the amount of resource available.

Note that, in our formulation, we assume that the resource allocation problem is a goal oriented problem under finite-horizon. Since the reward depends on time step  $t$ , it is natural to consider the finite horizon optimization. We simply take the time horizon  $h$  of the problem to be the latest time step where reward for one of the tasks is not 0. An MTS can be converted to an MDP in which the state space consists of the cross product of the individual state spaces of the tasks and the available resources, and the action space is the cross product of individual action spaces specifying resource assignments. The rewards are determined by summing the individual rewards and subtracting action costs. The transition probabilities are given by multiplying the individual transition probabilities.

To solve an MTS, we should find an optimal non-stationary policy  $\bar{\pi}_t^* = [\pi_{1,t}^*, \dots, \pi_{n,t}^*]$  for each time step  $t$  between 0 and  $h - 1$ , where  $\pi_{i,t}^*$  denotes an optimal resource allocation

to task  $i$  at time step  $t$  which maximizes

$$E\left[\sum_{t=0}^{h-1} \sum_{i=1}^n R_i(x_{i,t}, \pi_{i,t}^*(x_{i,t}), x_{i,t+1}, t) - C\pi_{i,t}^*(x_{i,t}) \mid \bar{\pi}^*\right]$$

subject to

$$\sum_{t=0}^{h-1} \sum_{i=1}^n \pi_{i,t}^*(x_{i,t}) \leq G.$$

It is natural that the MDP induced from an MTS is very large and hence, finding an optimal policy is a very hard problem even for a small MTS. The major difficulty comes from that fact that the decision to allocate a resource to a task influences the availability of that resource for other tasks. Thus, a local policy for each task should take into account the state of each of the other tasks.

We introduce an approximation strategy for the MTS, which we call Markov task decomposition (MTD). MTD solves MTS by independently solving individual tasks and combine the local policies  $\pi_{i,t}^*$ . In summary, MTD runs in two phases. First, in the *off-line phase*, the individual value functions are calculated for each task. Second, in the *on-line phase*, these value functions are used to determine the action to be executed given the current state of all tasks.

In the off-line phase, MTD computes the component value functions  $V_i(x_i, t, m)$  where  $x_i$  is the state of task  $i$ ,  $t$  is the time step between 0 and  $h$ , and  $m$  is the number of resources remaining:

$$V_i(x_i, t, m) = \max_{a \leq m} \sum_{x'_i \in X_i} T_i(x_i, a, x'_i) [R_i(x_i, a, x'_i, t) - V_i(x'_i, t + 1, m - a)] - Ca$$

with  $V_i(x_i, h, m) = 0$  for any  $x_i$  and  $m$ . These component value functions ignore all other tasks. Note that it may be suboptimal to spend all of the remaining resources at the last stage ( $t = h - 1$ ).

In the on-line phase, MTD determines the answer to the following question: given the current state  $\vec{x} = [x_1, \dots, x_n]$  and the amount of the remaining resource  $m$ , and the time step  $t$ , what action  $\vec{a} = [a_1, \dots, a_n]$  is (close to) optimal? The collection of precomputed component value functions  $V_i$  is used as the basis for a heuristic. Figure 4.30 shows how MTD in the on-line phase uses  $V_i$  at each time step. Allocating  $m_i$  to each task  $i$  generally requires specific domain knowledge. Note that even if we know the optimal allocation  $m_i$  at time step  $t$ ,  $\sum_i V_i(x_i, t, m_i)$  is a lower bound on  $V^*$  since it is the value we would achieve if we made the allocation at time step  $t$  and never made any change in the later steps.

1. For each  $i$ , using  $V_i(s_i, t, m_i)$  computed in the off-line phase, heuristically assign resources  $m_i$  to task  $i$  such that  $\sum_i m_i \leq m$  where  $m$  is the amount of resource available at the current time step.

2. Given  $m_i$  computed from step 1, compute  $a_i$  from the following equation:

$$a_i = \arg \max_{a \leq m_i} \sum_{x'_i \in X_i} T_i(x_i, a, x'_i) [R_i(x_i, a, x'_i, t) + V_i(x'_i, t + 1, m_i - a)] - Ca.$$

3. Execute action  $\vec{a} = [a_1, \dots, a_n]$  obtained from step 2 and compute remaining amount of resource  $m' = m - \sum_{a_i} a_i$ .

Figure 4.30: The MTD algorithm in the on-line phase.

In the disaster relief domain which we primarily focused on, the resources are of two types: airplanes and relief packages. Each site forms an individual task. The status of a state is either relieved or unrelieved, *i.e.*,  $X_i = \{d, u\}$ . Each site  $i$  has a *window of delivery* specified as

$$[b_i, e_i]$$

and the reward  $r_i$  is received when the site is relieved during the window:

$$R_i(x_i, a_i, x'_i, t) = \begin{cases} r_i & \text{if } b_i \leq t \leq e_i \text{ and } x_i = u \text{ and } x'_i = d. \\ 0 & \text{otherwise.} \end{cases}$$

Supplying a relief package relieves site  $i$  with probability  $p_i$ , and the noisy-or model is assumed for supplying more than one relief package:

$$T_i(x_i, a_i, x'_i) = \begin{cases} 0 & \text{if } x_i = d \text{ and } x'_i = u. \\ 1 & \text{if } x_i = d \text{ and } x'_i = d. \\ (1 - p_i)^{a_i} & \text{if } x_i = u \text{ and } x'_i = u. \\ 1 - (1 - p_i)^{a_i} & \text{if } x_i = u \text{ and } x'_i = d. \end{cases}$$

The off-line phase of the MTD algorithm for the disaster relief domain exploits the following observations:

- We can show that  $V_i(u, t, m)$  increases monotonically with  $m$  until a point  $m^*$  at which point it remains constant:  $m^*$  is the point at which the marginal utility of resource

allocation becomes zero, and where the marginal utility of resource use is negative. This means that we only need to compute  $V_i(u, t, m)$  for  $m \leq m^*$ .

- For each site  $i$ , it is non-optimal to supply a relief package outside the window  $[b_i, e_i]$ .

More specifically, we compute the component value functions for each site  $i$ ,  $m \leq m^*$ , and  $t \in [b_i, e_i]$  using dynamic programming:

$$V_i(u, t, m) = \max_{0 \leq a \leq m} [(1 - (1 - p_i)^a)R_i - Ca + (1 - p_i)^a V_i(u, t + 1, m - a)]$$

where  $V_i(u, e_i + 1, m) = 0$  and  $V_i(u, t, 0) = 0$  for all  $t$  and  $m$ .

The on-line phase of the MTD algorithm adopts a greedy strategy for allocating relief packages. Define

$$\Delta V_i(u, t, m) \equiv V_i(u, t, m + 1) - V_i(u, t, m)$$

which specifies the marginal utility of assigning an additional relief package to site  $i$ , given that  $m$  relief packages are assigned to the site already. We assign relief packages one by one to the site that has the highest value of  $\Delta V_i(u, t, m)$  given its current assignment of relief packages. This strategy is equivalent to greedily maximizing  $\sum_i V_i$ . However, the fact that  $V_i$  is monotonically non-decreasing assures that this strategy always finds the maximum of  $\sum_i V_i$ . Let  $m_i^\dagger$  denote the number of relief packages allocated to site  $i$  by the above strategy. The tasks are completely decoupled and the optimal policy  $\pi_t$  is defined as

$$\begin{aligned} \pi_{i,t}(u) &\equiv \arg \max_a [(1 - (1 - p_i)^a)R_i - Ca + (1 - p_i)^a V_i(u, t + 1, m_i^\dagger - a)], \\ \pi_{i,t}(d) &\equiv 0, \end{aligned} \tag{4.7}$$

where  $\pi_{i,t}$  is the  $i$ -th component of  $\pi_t$ , *i.e.*, action to be executed for site  $i$ .

We can make the domain more complicated by introducing constraints on airplanes that carry the relief packages. There are a limited number of airplanes available ( $U$ ) and each airplane can carry a limited number of relief packages per campaign ( $K$ ). The on-line phase of MTD satisfies these constraints in the following manner: An action  $\vec{a} = [a_1, \dots, a_n]$  is determined for the current stage via MTD ignoring the airplane constraints. We assign  $u_i$  airplanes to site  $i$ , sufficient to carry  $a_i$  relief packages. This assignment may be infeasible if  $\sum u_i > U$  thus violating the airplane constraint. We use a greedy reallocation strategy to satisfy the constraint  $\sum u_i \leq U$ .

We have to select which site receives fewer airplanes, hence fewer relief packages. To this end, we define the negative change in the value by deallocating one airplane from site

$i$ :

$$\begin{aligned} \blacktriangle V_i(u, t, m_i^\dagger) &\equiv (1 - (1 - p_i)^{a_i'}) R_i - C a_i' + (1 - p_i)^{a_i'} V_i(u, t + 1, m_i^\dagger - a_i') \\ &\quad - (1 - (1 - p_i)^{a_i}) R_i + C a_i - (1 - p_i)^{a_i} V_i(u, t + 1, m_i^\dagger - a_i) \\ &\quad + \delta V_i \end{aligned}$$

where  $a_i'$  is the resulting number of packages delivered. Note that we can use the relief packages from the deallocated airplane, reallocating to some other site in the later time steps.  $\delta V_i$  defines such increase in the overall value by reallocating the relief packages. The reallocation process is again similar to the on-line phase of MTD without the airplane constraint: Define the marginal utility of adding a relief package to site  $i$  in the later time steps as

$$\Delta' V_i(u, t, m) \equiv (1 - p_i)^{a_i} [V_i(u, t + 1, m + 1 - a_i) - V_i(u, t + 1, m - a_i)].$$

Thus starting from  $m_1^\dagger, \dots, m_n^\dagger$ , we assign  $(a - a_i)$  relief packages one by one to the site that has the highest value of  $\Delta' V_i(u, t, m)$ .  $\delta V_i$  is the overall change in the value in the end of the process: Let

$$\Delta'_k V_i(u, t, m) \equiv (1 - p_i)^{a_i} [V_i(u, t + 1, m + k - a_i) - V_i(u, t + 1, m - a_i)],$$

and  $d_j$  be the number of relief packages reallocated to site  $j$  so that  $\sum d_j = a - a_i$ , then  $\delta V_i \equiv \sum_j \Delta'_{d_j} V_j$ . Thus, we keep deallocating airplanes one by one from the target with the largest  $\blacktriangle V_i(u, t, m_i^\dagger)$  until the airplane constraint is met. This process outputs an action specifying the number of relief packages and airplanes to be allocated to each site.

Besides the decomposition technique used by MTD, the algorithm adopts on-line policy to alleviate the need to reason about and store many future contingencies. While this idea has been popular in solving large MDPs (Barto *et al.* [6]), the crucial success of the MTD method relies on the ability to construct good actions heuristically using component value functions.

### 4.3.1 Experiments

Figure 4.31 shows the running times of MTD with varying number of sites, packages and airplanes. For each problem size parameter  $s$ , we generate 30 random instances involving  $100s$  sites,  $1000s$  packages, and  $10s$  airplanes. Each  $p_i$ , the probability of successfully relieving site  $i$ , is sampled uniformly in  $[0, 1]$ . Each  $r_i$ , the reward of relieving site  $i$ , is sampled uniformly in  $[0, 100]$ . The window of delivery  $[b_i, e_i]$  is sampled in the following way: first, the the beginning of the window is sampled uniformly in  $[0, h]$ , where  $h$  is the

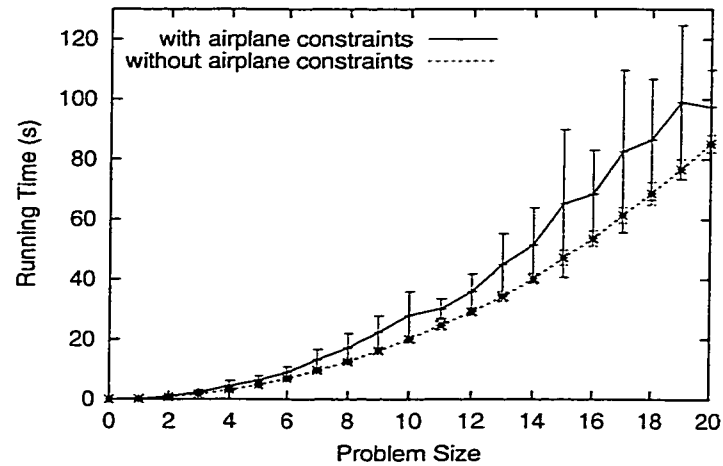


Figure 4.31: The running times of MTD. Each plot is an average running time of 30 randomly generated problems with size parameter  $s$ , so that the problem involves  $100s$  sites,  $1000s$  packages, and  $10s$  airplanes. Note that the running times on problems with airplane constraints are larger and more varied than those without airplane constraints.

horizon of the problem. Second, assuming that the beginning of the window is  $b_i$ , the end of the window is sampled uniformly in  $[b_i, h]$ . The running time for each problem represents the time spent on the off-line phase plus the time spent on one simulation of the online phase. Note that the running time varies from one simulation to another for the same problem. In our experiments, the biggest problem with thousands of sites and tens of thousands of packages is solved in an order of minutes. Introducing the constraint that we have a limited number of available airplanes slightly increases the running time of the online phase. Throughout the problems, we assume that each airplane has a capacity of 4 packages. Note that the variance of the running times is quite high for problems with airplane constraints. This is due to the resource deallocation and reallocation routines in the on-line phase of MTD.

We find that using dynamic programming without decomposition is computationally intractable. The state space is the cross product of the status of each site and the number of available packages. The action space is all possible allocations of packages to sites. Even excluding some of the states that are unreachable, the running time for a problem with 5 sites and 50 packages takes hours in general.

We compare the policy obtained from MTD to the optimal policy. Since calculating the optimal policy without decomposition is intractable, we test on small problems only. In addition, we compare the performance of the policy from MTD with two other less

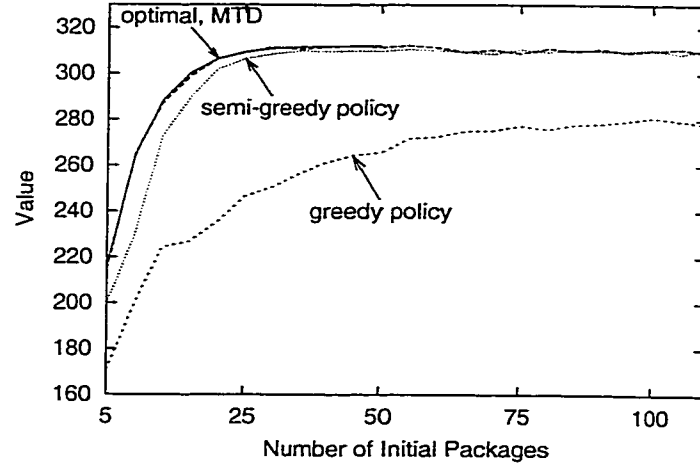


Figure 4.32: Performances of the policies on a randomly generated 5-site problem without airplane constraints.

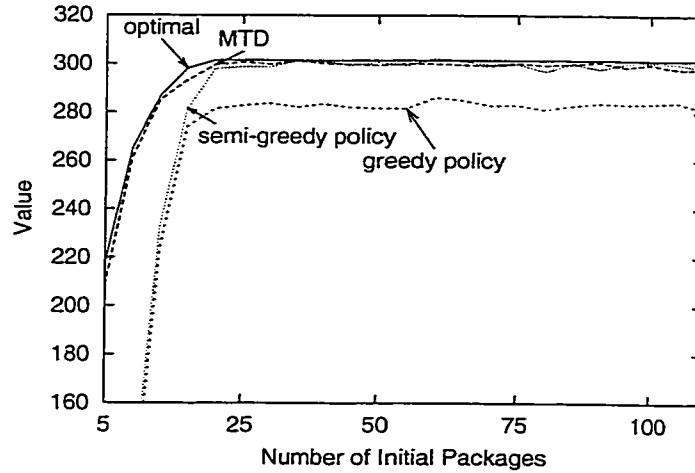


Figure 4.33: Performances of the policies on a randomly generated 7-site problem with airplane constraints.

sophisticated policies:

- **Greedy policy** : We disregard the component value functions computed during the off-line phase and try to maximize the immediate reward. In problems with airplane constraints, the site with a higher reward has a higher priority in allocating the airplanes.
- **Semi-greedy policy** : The resource allocation is done by Equation 4.7, but disregard the constraint that  $\sum_i m_i^\dagger \leq G$ . If the allocation attempt exceeds the available



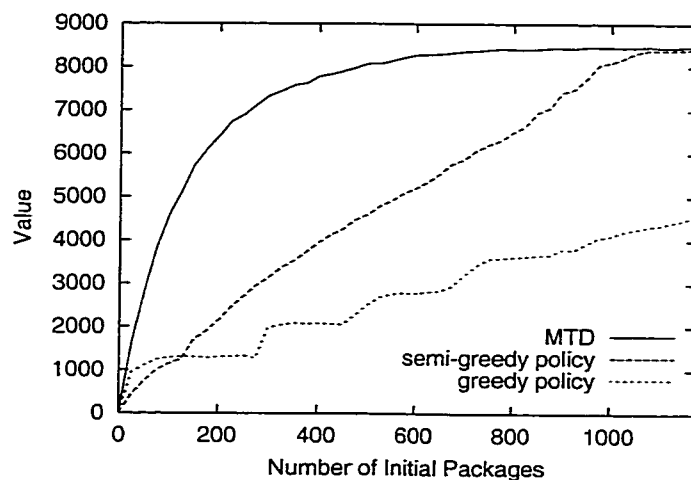


Figure 4.34: Performances of the policies on a randomly generated 200-site problem.

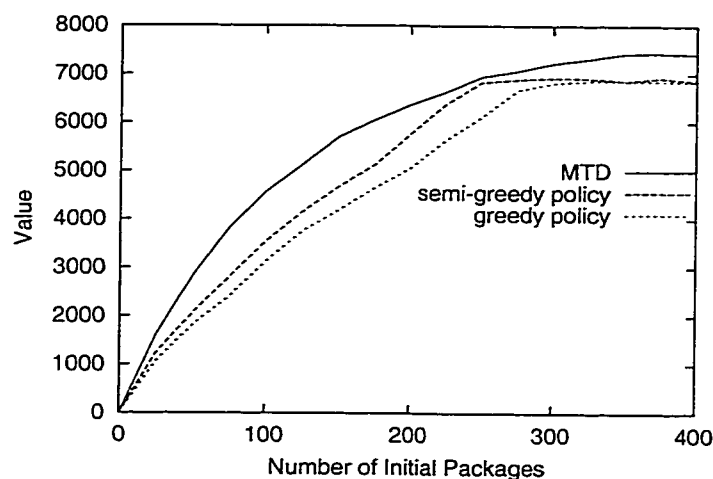


Figure 4.35: Performances of the policies on a randomly generated 200-site problem with 20 airplanes.

resource, the site with a lower index receives a higher priority, so that site 1 receives the highest priority and site  $n$  receives the lowest priority.

Figure 4.32 shows the performance of these policies as the number of initial packages increases in a randomly generated 5-site problem without airplane constraints. Figure 4.33 shows the performance of these policies as the number of initial packages increases in a randomly generated 7-site problem with 3 airplanes each capable of carrying 2 packages. The performance of MTD is closer to optimal compared to the greedy policy and the semi-greedy policy.

Figure 4.34 compares the policy from MTD to the greedy and semi-greedy policies in a large domain without airplane constraints. Figure 4.35 compares the policy from MTD to the greedy and semi-greedy policies in the same domain with 20 airplanes each capable of carrying 4 packages at a time. With or without airplane constraints, the behavior of the policy from MTD is most interesting when we have relatively few initial packages. The greedy and semi-greedy policies are far from the optimal, whereas the policy from MTD is very close to the optimal. In fact, when we have unbounded number of initial packages, Equation 4.7 with  $m_i^\dagger$  replaced by  $\infty$  is provably optimal, which makes the on-line phase of MTD trivial.

### 4.3.2 Related Work

The work by Dean and Lin [36] is aimed at solving MDPs with large state spaces (but not factored representations such as FMDPs) by decomposing the original MDP into sub-MDPs. Their method is different from ours in the fact that their sub-MDPs are sequential — the state spaces of sub-MDPs constitute a partition of the original state, and only one sub-MDP is *active* at each time step in the sense that the state space of the active sub-MDP contains the current state. The method presented in this section can be seen as a parallel decomposition method compared to their work. The MTD is most effective when the domain problems can be identified as a set of parallel sub-problems with limited interactions. The series of work by Boyen and Koller [20, 21, 22] provide theoretical analyses and applications of parallel decomposition in the area of temporal reasoning. Their work is concerned with computing approximation to the probability distribution on state spaces of DBNs (Forbes *et al.* [41]) and the error bound for the approximated probability distribution is derived using Kullback-Liebler distance measure from information theory (Cover and Thomas [31]). Extension of their work to derive the performance bound on MTD requires finding an adequate distance measure, which is yet to be discovered.

There have been a number of algorithms for combining policies from sub-MDPs without violating the constraints. A reinforcement learning algorithm by Singh and Cohn [97] uses the lower bound and the upper bound of the value function to resolve conflicts when computing the optimal policy. The lower bound and the upper bound will eventually converge to the same value, but there is no bound on the running time. The method by Boutilier *et al.* [13] extracts the most likely path of states taken by the local optimal policy for each sub-MDP, and does a heuristic search to find a globally optimal policy. Calculating

the most likely path allows us to limit the search space and to assume the domain is deterministic. Thus, the local optimal policy becomes a partially ordered plan with a causal structure, and merging two local policies is done by maximizing one of the two local value functions while maintaining the causal structures in both policies. Yang *et al.* [112] provides algorithms for merging plans for sub-goals with restricted interaction in deterministic planning domains. Kearns *et al.* [64] presents a general-purpose on-line policy algorithm for large domains and the theoretical analyses behind the algorithm.

The above algorithms are more general than MTD in the sense that MTD is concerned with a task with only two types of resource constraints, namely the relief packages and the airplanes. MTD can be seen as a special case of these algorithms, optimized to solve the disaster relief domains that are explained in this section. In a closer setting, Castanón [29] and Yost and Washburn [113] investigate the decomposition of large resource allocation problems posed as POMDPs. The key technique that allows decomposition is relaxing the resource constraints. This is similar to our strategy, since MTD assumes unbounded amount of resource in the off-line phase. We suspect that if we reformulate our disaster relief domain as a true MDP, in the sense that the domain does not have the window of delivery constraint and symbolic actions, their technique is equivalent to ours, except that MTD exploits the full observability of the domain and constructs component value functions for each site.

#### 4.4 Model-Based Reinforcement Learning Techniques

Most of the current FMDP algorithms assume the decision tree for representing the stationary optimal policy: non-terminal nodes represent important aspects (or features) of the current state and terminal nodes determine which action to execute given those features are present in the current state (Boutilier *et al.* [16]). Note that by restricting the size of the policy, thereby limiting the features used to condition actions, we also reduce our ability to discriminate among states. What this means is that the underlying dynamics, which is in part determined by the actions selected according to the agent’s policy, may become *non-Markovian*. Within the space of all policies that have the same capability with respect to discriminating among states, the best policy may require remembering features from states in the past time steps, which implies that the optimal policy may be history dependent. This observation is the primary motivation for the work described in this section.<sup>4</sup>

One way to represent the history is to use memory of past states. We could of course enhance the agent’s ability to discriminate among states by having it remember features

<sup>4</sup>The material presented here is based on Kim *et al.* [68]

from previous states. However, this simply transforms the problem into an equivalent problem with an enlarged set of features exacerbating the curse of dimensionality in the process. An alternative approach that has been pursued in solving POMDPs is to use PFSA policies (Definition 3.2.1 of this thesis, Hansen [48], Meuleau *et al.* [82, 83] for more details). The advantage of this approach is that such controllers can *remember* features for an indefinite length of time without the overhead of remembering all of the intervening features. Search is carried out in the space of policies that have a finite amount of memory.

In this section, we adopt the approach of searching directly in the space of policies as opposed to searching in the space of value functions, finding the optimal value function or a good approximation thereof and constructing a policy from this value function. Finding the best PFSA policy is a hard problem and we step aside from some of the computational difficulties by using a stochastic greedy search algorithm based on Baird and Moore’s VAPS algorithm [5] and its predecessor REINFORCE algorithm (Williams [111]) to find a locally optimal PFSA policy. In particular, the VAPS algorithm generalizes the REINFORCE algorithm so that it covers a variety of reinforcement learning algorithms as a stochastic greedy search algorithm trying to minimize mean squared residuals per trial, where the residuals are defined differently depending on the reinforcement learning algorithm of the interest. We use the mathematical formulations of the VAPS algorithm to present our approach.

Our algorithm is a trial-based, stochastic gradient descent of a general error measure, and hence we can show that it converges to a local optimum with probability 1. The measure we attempt to maximize is the expected cumulative discounted reward for a policy  $\pi$ :

$$V_\pi(\vec{x}_0) \equiv E \left[ \sum_{t=0}^{\infty} \gamma^t R(\vec{x}_t, |a_t) | \vec{x}_0, \pi \right],$$

where (as a reminder)  $\vec{x}_t$  and  $a_t$  is the state of FMDP and the action taken at time  $t$ , respectively. Assume that the problem is a goal-achievement task — there exists an absorbing goal-state that the system must be in as quick as possible. As soon as the system reaches the goal-state, the system halts and assigns a reward. In this case, we can write down our optimality criterion as minimizing  $C_\pi$  with respect to some error measure  $\epsilon$ , defined by

$$C_\pi \equiv \sum_{T=0}^{\infty} \sum_{h_T \in H_T} P(h_T | \vec{x}_0, \pi) \epsilon(h_T), \quad (4.8)$$

where  $H_T$  is the set of all trajectories that terminate at time  $T$  so that

$$h_T \equiv [\vec{x}_0, v_0, i_0, a_0, r_0, \dots, \vec{x}_t, v_t, i_t, a_t, r_t, \dots, \vec{x}_T, v_T, i_T, a_T, r_T],$$

where  $v_t$  is the vertex visited at time  $t$ .  $i_t$  is the index number of the satisfied boolean function at time  $t$  so that  $f_{v_t, i_t}(\vec{x}_t)$  is true, and  $\epsilon(h_T)$  denotes the total error associated with trajectory  $h_T$ . We assume the total error  $\epsilon$  is additive in the sense that

$$\epsilon(h_T) \equiv \sum_{t=0}^T e(h_T[0, t]),$$

where  $e(h_T[0, t])$  is an instantaneous error associated with sequence prefix

$$h_T[0, t] \equiv [\vec{x}_0, v_0, i_0, a_0, r_0, \dots, \vec{x}_t, v_t, i_t, a_t, r_t].$$

We can define  $e$  in different ways to obtain different algorithms. For example, we can obtain Q-learning [107] by instantiating  $e$  as

$$e(h_T[0, t]) \equiv \frac{1}{2} [r_{t-1} + \gamma \max_a Q(\vec{x}_t, a) - Q(\vec{x}_{t-1}, a_{t-1})]. \quad (4.9)$$

In this work, we use TD(1) error (Sutton and Barto [103]) defined by

$$e(h_T[0, t]) \equiv -\gamma^t r_t.$$

In this case, we note that our objective function (Equation 4.8) is exactly  $-V_\pi$ , therefore we try to minimize  $C_\pi$ .

To illustrate the stochastic gradient descent algorithm on  $C_\pi$ , we derive the gradient  $\nabla C_\pi$  with respect to the parameters of the PFSA policy. From Definition 3.2.1, the parameter space is  $\{\Psi, \Delta_I, \Delta_T\}$ . Without loss of generality, let  $\omega$  be any parameter of the PFSA policy. Then we have

$$\begin{aligned} \frac{\partial C_\pi}{\partial \omega} = \sum_{T=0}^{\infty} \sum_{h_T \in H_T} & \left[ P(h_T | \vec{x}_0, \pi) \frac{\partial \epsilon(h_T)}{\partial \omega} \right. \\ & \left. + \epsilon(h_T) \frac{\partial P(h_T | \vec{x}_0, \pi)}{\partial \omega} \right]. \end{aligned} \quad (4.10)$$

Note that for our choice of  $e$ , the partial derivative of  $\epsilon(h_T)$  with regard to  $\omega$  is always 0. If we chose Q-learning algorithm by following Equation 4.9 so that the  $Q$  function is represented by a set of parameters  $\Phi$ , we should also calculate the partial derivatives with respect to parameters  $\phi \in \Phi$ . In this case,  $\partial \epsilon(h_T) / \partial \phi$  would not be 0.

We now derive the partial derivative of  $P(h_T | \vec{x}_0, \pi)$  with regard to  $\omega$ . Since

$$\begin{aligned} P(h_T | \vec{x}_0, \pi) = & P(\vec{x}_0) \delta_I(v_0) \psi(v_0, a_0) \\ & \prod_{t=1}^T \left[ T(\vec{x}_{t-1}, a_{t-1}, \vec{x}_t) \delta_T(v_{t-1}, f_{v_{t-1}, k_t}(\vec{x}_t), v_t) \psi(v_t, a_t) \right], \end{aligned}$$

where  $k_t$  is selected so that  $f_{v_{t-1}, k_t}(\bar{x}_t)$  is true, we can rewrite Equation 4.10 as

$$\begin{aligned} \frac{\partial C_\pi}{\partial \omega} = & \sum_{T=0}^{\infty} \sum_{h_T \in H_T} P(h_T | \bar{x}_0, \pi) \left[ \epsilon(h_T) \sum_{t=0}^T \frac{\partial \ln \psi(v_t, a_t)}{\partial \omega} \right. \\ & + \epsilon(h_T) \sum_{t=1}^T \frac{\partial \ln \delta_T(v_{t-1}, f_{v_{t-1}, k_t}(\bar{x}_t), v_t)}{\partial \omega} \\ & \left. + \epsilon(h_T) \frac{\partial \ln \delta_I(v_0)}{\partial \omega} \right]. \end{aligned}$$

The above equation implies that the gradient is the mean of

$$\begin{aligned} g_\omega(h_T) = & \epsilon(h_T) \sum_{t=0}^T \frac{\partial \ln \psi(v_t, a_t)}{\partial \omega} \\ & + \epsilon(h_T) \sum_{t=1}^T \frac{\partial \ln \delta_T(v_{t-1}, f_{v_{t-1}, k_t}(\bar{x}_t), v_t)}{\partial \omega} \\ & + \epsilon(h_T) \frac{\partial \ln \delta_I(v_0)}{\partial \omega} \end{aligned} \quad (4.11)$$

with regard to the distribution  $P(h_T | \bar{x}_0, \pi)$ . Hence, stochastic gradient descent of the error is done by repeatedly sampling a trajectory  $h_T$  and evaluating  $g_\omega(h_T)$  and updating the parameters of the PFSA policy by adding the negative of the gradient,  $-\nabla C_\pi$ .

Note that above derivation using natural logarithms assumes that we have non-zero probabilities for all the parameters of the PFSA policy. We can enforce this condition by using a *Boltzmann distribution* for the parameters of the PFSA policy

$$\begin{aligned} \psi(v, a) &\equiv \frac{e^{q\psi(v, a)/\theta}}{\sum_{a' \in A} e^{q\psi(v, a')/\theta}} \\ \delta_T(v, f_{v, k}(\bar{x}), v') &\equiv \frac{e^{q\delta_T(v, f_{v, k}(\bar{x}), v')/\theta}}{\sum_{v'' \in \mathcal{V}} e^{q\delta_T(v, f_{v, k}(\bar{x}), v'')/\theta}} \\ \delta_I(v) &\equiv \frac{e^{q\delta_I(v)/\theta}}{\sum_{v' \in \mathcal{V}} e^{q\delta_I(v')/\theta}} \end{aligned}$$

where  $\theta$  is the temperature that governs how sharply the probability mass is concentrated around the maximum of  $q$  values. Smaller  $\theta$  makes the probability mass more sharply concentrated. Without loss of generality, we assume in this work  $\theta = 1$  since we can make the probability mass arbitrarily more concentrated by increasing  $q$ . To obtain the gradient of  $C_\pi$  with respect to the  $q$ 's, we first calculate the gradient with respect to the  $\psi$ 's and the

$\delta$ 's:

$$\begin{aligned} g_{\psi(v,a)}(h_T) &= \frac{N_{v,a}(h_T)}{\psi(v,a)} \sum_{t=0}^T \gamma^t r_t \\ g_{\delta_T(v,f_{v,k}(\bar{x}),v')} &= \frac{N_{v,f_{v,k}(\bar{x}),v'}(h_T)}{\delta_T(v,f_{v,k}(\bar{x}),v')} \sum_{t=0}^T \gamma^t r_t \\ g_{\delta_I(v)} &= \frac{N_v(h_T)}{\delta_I(v)} \sum_{t=0}^T \gamma^t r_t, \end{aligned}$$

where  $N_{v,a}(h_T)$  is the number of times that action  $a$  is executed at vertex  $v$  in  $h_T$ ,  $N_{v,f_{v,k}(\bar{x}),v'}$  is the number of times that transition happens from vertex  $v$  to vertex  $v'$  while  $f_{v,k}(\bar{x})$  is true in  $h_T$ , and finally,  $N_v(h_T)$  is the number of times that vertex  $v$  is selected at time 0 in  $h_T$ . Note that  $N_v(h_T)$  is either 0 or 1, but we use this notation for the sake of uniformity. Observing that the gradient of the Boltzmann distribution

$$p_i \equiv \frac{e^{q_i}}{\sum_j e^{q_j}}$$

is calculated as

$$\frac{\partial p_i}{\partial q_j} = \begin{cases} p_i - p_i^2 & \text{if } i = j, \\ -p_i p_j & \text{if } i \neq j, \end{cases}$$

we use the chain rule to obtain the gradient with respect to  $q$ 's in the PFSA policy:

$$\begin{aligned} g_{q_{\psi(v,a)}}(h_T) &= g_{\psi(v,a)}(h_T)[\psi(v,a) - \psi(v,a)^2] \\ &\quad - \sum_{a' \neq a} g_{\psi(v,a')}(h_T)[\psi(v,a)\psi(v,a')] \\ g_{q_{\delta_T(v,f_{v,k}(\bar{x}),v')}}(h_T) &= g_{\delta_T(v,f_{v,k}(\bar{x}),v')}(h_T)[\delta_T(v,f_{v,k}(\bar{x}),v') - \delta_T(v,f_{v,k}(\bar{x}),v')^2] \\ &\quad - \sum_{v'' \neq v'} g_{\delta_T(v,f_{v,k}(\bar{x}),v'')}(h_T)[\delta_T(v,f_{v,k}(\bar{x}),v')\delta_T(v,f_{v,k}(\bar{x}),v'')] \\ g_{q_{\delta_I(v)}}(h_T) &= g_{\delta_I(v)}(h_T)[\delta_I(v) - \delta_I(v)^2] \\ &\quad - \sum_{v' \neq v} g_{\delta_I(v')}(h_T)[\delta_I(v)\delta_I(v')]. \end{aligned}$$

#### 4.4.1 Experiments

Our implementation assumes that all the boolean functions that label transitions take only one of the domain fluents as input. Thus, the formal definition of PFSA policies we focus on is given as follows:

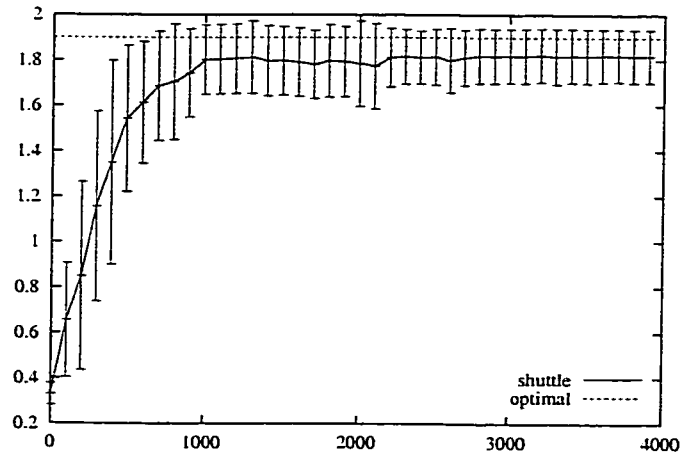


Figure 4.36: Performance on the modified shuttle problem.

**Definition 4.4.1 (Unary PFSA Policy)** A unary PFSA policy  $\pi$  on FMDP  $M = \{\bar{X}, A, T, I, R\}$  is represented by  $(\mathcal{V}, \mathcal{E}, \Psi, \Delta_I, \Delta_T)$  such that

- $\mathcal{V}$  is the set of vertices in the graph.
- $\mathcal{E}$  is the set of directed edges in the graph.
- $\Psi = \{\psi(v, a) | v \in \mathcal{V}, a \in A\}$  such that  $\psi(v, a)$  is the probability of choosing action  $a$  in vertex  $v$ .
- $\Delta_I = \{\delta_I(v) | v \in \mathcal{V}\}$  such that  $\delta_I(v)$  is the probability of initial vertex being  $v$ .
- $\Delta_T = \{\delta_T([v, i], x_i, [v', i']) | v, v' \in \mathcal{V}, i, i' \in \{1, \dots, n\}, x_i \in \Omega_{X_i}\}$  such that  $\delta_T([v, i], x_i, [v', i'])$  is the joint probability of moving from vertex  $v$  and focusing on the fluent  $X_i$  to vertex  $v'$  and focusing on the fluent  $X_{i'}$ , after observing  $x_i$ . Formally, letting  $V_t$  and  $I_t$  be the random variables that respectively represent the vertex and the focused fluent index at time  $t$ ,

$$\delta_T([v, i], x_i, [v', i']) = P(V_{t+1} = v', I_{t+1} = i' | V_t = v, I_t = i, X_{i,t+1} = x_i)$$

for all time steps  $t$ .

Thus, our implementation generates all possible unary boolean functions for transitions between vertices, and selects the best one by increasing the appropriate  $\delta_T$ .

Figure 4.36 presents the learning curve of the unary PFSA policy on a modified version of a small benchmark problem for POMDP algorithms, called the shuttle problem. The



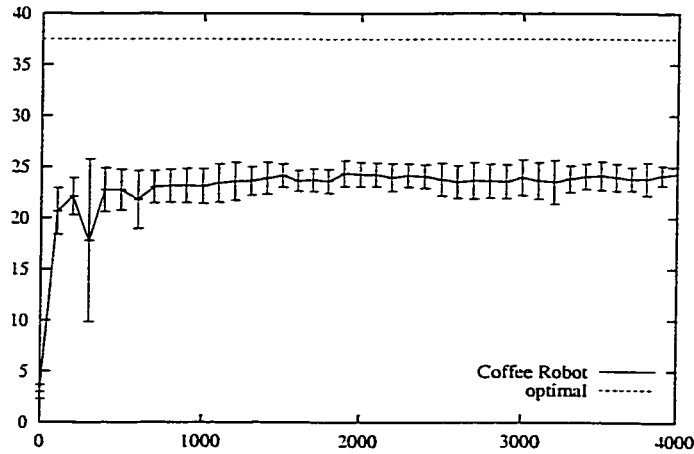


Figure 4.37: Performance on the Coffee Robot domain.

original shuttle problem (Cassandra [28]) is given as a POMDP, where the task is driving a shuttle back and forth between a loading site and an unloading site to deliver a package. The status of whether the shuttle is loaded or not is hidden. For our experiments, we modified the problem so that everything is observable. The problem is composed of 3 fluents, that represent the location of the shuttle, the status of loaded or not, and the arrival of the package at the loading site, respectively. A reward of 1 is given every time the shuttle arrives at the loading site with empty cargo and returns to the unloading site with the package. With the discount rate of 0.9, the optimal performance is around 1.9. The graph shows the average of 10 independent experiments each with 4000 gradient updates on a unary PFSA policy with 2 vertices.

Figure 4.37 shows the performance of the algorithm on the toy coffee robot problem. It is an extended version of the simple problem in Figure 3.1. When explicitly enumerated (*i.e.* flattened out), the problem has 400 states. By using a unary PFSA policy with 2 vertices, we were able to find a locally optimal policy within minutes. The optimal performance is 37.49. The graph shows the average of 10 independent experiments each with 4000 gradient updates. On average, the learned unary PFSA policy had performance around 25. We expected to learn a gradually better policy as we added more vertices, but we ended up stuck at local optima with same performance. The result is to be expected given that we only search in the space of unary PFSA policies and the algorithm looks for local optima; however, the result is in contrast to what we observed in Meuleau *et al.* [83] where increasing the size of the unary PFSA policies allowed for gradual improvements. We suspect that there are large discrete jumps in the performance as the expressiveness of the policy passes

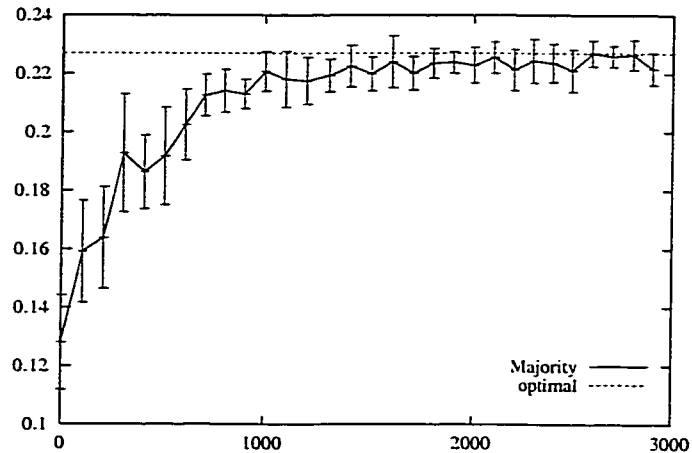


Figure 4.38: Performance on the Majority domain with 5 bits.

some threshold, or the stochastic greedy search algorithm is not good enough to pass a certain local optimum. Extending the expressiveness of the policy by searching in the space of binary PFSA policies was too slow due to the combinatorial explosion in the number of parameters of the policies.

Figure 4.38 shows the performance on the majority domain shown in Figure 3.4 and Figure 3.5. The task was to determine the whether the majority of 5 bits are set or not. When explicitly enumerated, the domain has 5632 states. Using 1-bit memory (2 state FSA), the stochastic gradient descent algorithm converged to the optimum. If we were to use a history-independent policy represented as a decision tree, the most compact optimal policy will have  $2^5$  leaves. Again, the performance graph shows the average of 10 independent experiments each with 3000 gradient updates.

#### 4.4.2 Related Work

The work described in this chapter borrows from two main threads of research besides the work on factored representations. The first thread follows from the idea that it is possible, even desirable in some cases, to search in the space of policies instead of in the space of value functions which is the basis for much of the work involving dynamic programming. The second thread is that it is possible to use gradient methods to search in the space of policies or the space of value functions if the parameterized form of these spaces is smooth and differentiable.

Singh *et al.* [98] investigate theoretical issues and algorithmic methods for learning without state estimation in POMDPs. They provide a new reinforcement learning algorithm

that searches for a stochastic stationary policy rather than a deterministic stationary policy in non-Markov decision processes without state estimation.

Baird and Moore [5] introduce the VAPS family of algorithms that allow searching in the space of policies or in the space of value functions or in some combination of the two. Baird and Moore's work generalizes on and extends the work of Williams [111].

Hansen [48] describes algorithms for solving POMDPs by searching in the policy space of finite memory controllers. Meuleau *et al.* [83] combines the idea of searching in an appropriately parameterized policy space with the idea of using gradient based methods for reinforcement learning. Our work employs an extension of the VAPS methodology to FMDPs.

Bacchus *et al.* [2] present a method for solving non-Markov decision processes by converting them to an FMDP by introducing fluents that capture important aspects of past history. Hence, an optimal stationary policy of this FMDP becomes a history-dependent policy. Using this method to solve FMDPs can be done by, first, converting the given FMDP to a more compact representation using a non-Markov decision process, and then solving the resulting non-Markov decision process using the Bacchus *et al.*'s method. The existence of an efficient algorithm for conversion is yet to be found.

## Chapter 5

# Structured Linear Algebra

In this chapter, we step back from the techniques that aim specifically at exploiting the structure in FMDPs and provide a framework for representing and manipulating vectors and matrices of very large dimension. These techniques take their motivation from the existing FMDP algorithms, and will in turn provide us with new techniques for solving FMDPs, but our immediate goal is to generalize some of what we have done so far. Our inspiration comes from the fact that classical MDP algorithms are mostly based on linear algebra operators for vectors and matrices whose dimensions are roughly of the size of the state space. For factored representations such as FMDPs, we need a way to compactly store vectors and matrices of very large dimension, in the sense that the dimension is exponential in the description length of the FMDP, and linear algebra operators that manipulate these vectors and matrices. We call such a compact data structure for vectors and matrices a *structured representation* in contrast with the simpler methods that allocate storage for every component of vectors and matrices.

Consider calculating vector  $V$  that results from a matrix-vector multiplication,

$$V = AU \tag{5.1}$$

in which  $A$  is an  $m \times m$  matrix and  $U$  and  $V$  are vectors of dimension  $m$ . The matrix  $A$  might be the adjacency matrix for a graph representing a database relation or the state-transition matrix for a planning problem and the vector  $U$  might be the specification of vertices corresponding to an adjacency query or the initial-state distribution.

Suppose that our matrices and vectors correspond to functions and relations on a domain  $\Omega_{\bar{z}}$ . Assume further that we can describe each element of  $\Omega_{\bar{z}}$  in terms of a set of  $n$  features of the domain defined by  $Z_1, \dots, Z_n$  where each feature  $Z_i$  can take on values from  $\Omega_{Z_i}$  and  $\Omega_{\bar{z}} = \prod_{i=1}^n \Omega_{Z_i}$ .

We refer to  $\vec{Z} = [Z_1, \dots, Z_n]$  as the set of *index variables* for  $\Omega_{\vec{z}}$ . Let  $\vec{z} = [z_1, \dots, z_n]$  be an element of  $\Omega_{\vec{z}}$  and define  $I(\vec{z})$  to be the integer index for  $\vec{z}$ . The integer index  $I(\vec{z})$  maps an assignment of values to index variables  $\vec{z}$  to a row number or a column number in the matrices and vectors.

Consider representing a function  $f : \Omega_{\vec{z}} \times \Omega_{\vec{z}} \rightarrow \mathfrak{R}$  as a matrix  $A = \{A_{ij}\}$  and suppose that there exists a partition  $P$  of  $\Omega_{\vec{z}} \times \Omega_{\vec{z}}$  such that for each block  $B \in P$  there exists  $A_B \in \mathfrak{R}$  such that, for all  $(\vec{z}, \vec{z}') \in B$ , if  $I(\vec{z}) = i$  and  $I(\vec{z}') = j$  then  $A_{ij} = f(\vec{z}, \vec{z}') = A_B$ .

Suppose that  $|P|$ , the number of blocks in  $P$ , is small, say polynomial in  $m$ , and it is possible to represent each block as a compact, easily computable function of the index variables  $[Z_1, \dots, Z_n]$ , then at least we can represent  $A$  compactly and compute the value of any  $A_{ij}$  efficiently. This is the type of regularity in the domain that we are concerned with in this chapter. In particular, we are interested in answering the following questions:

- How would this type of regularity help in terms of writing down matrix  $A$ , vectors  $U$  and  $V$ , and in terms of calculating multiplication  $AU$  or solving for  $U$  in  $AU = V$  given  $A$  and  $V$ ?
- Are there interesting cases of matrices and vectors for which small partitions of the sort described above exist and such that can we find them and exploit their regularities? For example, are there cases in which we can elicit the compact representations and requisite numbers from experts?
- Since we can carry out calculations such as  $AU$ , can we compute other linear algebra operators such as the multiplication of a transposed matrix and a vector  $A^T U$  and the power of a matrix  $A^k$ ?

In the following, we address the general problem of defining elementary matrix and vector operators involving very large dimensional vector spaces, and we draw on examples that arise in solving FMDPs.

## 5.1 Exploiting the Regularities in Matrices and Vectors

Consider representing a function  $f : \Omega \rightarrow \mathfrak{R}$  as a vector  $U$  and assume that there exists a partition  $P$  of  $\Omega_{\vec{z}} = \prod_i \Omega_{Z_i}$  such that for each block  $B \in P$

$$\exists U_B \in \mathfrak{R} \text{ such that } \forall \vec{z} \in B, U_{I(\vec{z})} = f(\vec{z}) = U_B \quad (5.2)$$

where we define  $U_B$  to be the value common to any index  $I(\vec{z})$  such that  $\vec{z} \in B$ . Assume further that matrix  $A = \{A_{ij}\}$  represents a function  $g : \Omega_{\vec{z}} \times \Omega_{\vec{z}} \rightarrow \mathfrak{R}$  so that each pair of blocks  $(B_1, B_2) \in P \times P$  satisfies

$$\exists A_{B_1 B_2} \in \mathfrak{R} \text{ such that } \forall \vec{z}_1 \in B_1, \forall \vec{z}_2 \in B_2, A_{I(\vec{z}_1)I(\vec{z}_2)} = g(\vec{z}_1, \vec{z}_2) = A_{B_1 B_2} \quad (5.3)$$

where we define  $A_{B_1 B_2}$  to be the value common to any pair of indices  $(I(\vec{z}_1), I(\vec{z}_2))$  such that  $(\vec{z}_1, \vec{z}_2) \in (B_1, B_2)$ . Under these assumptions we can calculate matrix-vector multiplication  $V = AU$  as follows

$$\forall B \in P, V_B = \sum_{B' \in P} |B'| A_{BB'} U_{B'}$$

so that  $V_i = V_B$  for all  $i = I(\vec{z})$  such that  $\vec{z} \in B$ . The running time in this case is  $O(|P|^2)$  which is acceptable if  $|P|$  is small. In some sense, the size of the partition provides a bound on the intrinsic complexity of the problem defined in the domain.

It will turn out that other elementary linear algebra operators such as scalar and dot products, sums and differences, can also be performed in time polynomial in  $|P|$ . In some cases, we can even achieve polynomial performance for more complicated operators. For example, we can calculate

$$A^k U$$

given  $A, U$  and an integer  $k$  in  $O(k|P|^2)$  time. We can also solve for  $U$  in

$$AU = V$$

given  $A$  and  $V$  using an iterative scheme in which each iteration takes  $O(|P|^2)$  time and the rate of convergence is independent of the dimension size of the matrix  $|\Omega_{\vec{z}}|^2$  or the size of the partition  $|P|$ . The idea is that we carve up the domain finely enough to ensure that the operands,  $A$  and  $U$  in this case, are *uniform* on  $P$  in the sense of satisfying Equation 5.2 and Equation 5.3. If  $A$  satisfies Equation 5.3 for the partition  $P'$ , and  $V$  satisfies Equation 5.2 for a different partition  $P''$ , then, in the worst case, the coarsest uniform partition will be of size  $O(|P'| |P''|)$ .

Unfortunately, not all systems of linear equations can be easily solved. For example, given our previously stated initial motivation for this work, an equation that is of particular interest to us is the vector version of the Bellman equation for a FMDP  $M = \{\vec{X}, A, T, I, R\}$  defined for every  $\vec{x} \in \Omega_{\vec{X}}$ :

$$V_{I(\vec{x})}^* = \max_{a \in A} \left[ R_{I(\vec{x})}^{(a)} + \gamma \sum_{\vec{x}' \in \Omega_{\vec{X}}} T_{I(\vec{x})I(\vec{x}')}^{(a)} V_{I(\vec{x}')}^* \right]$$

in which the set of fluents,  $\bar{X}$ , becomes the set of index variables in our framework.  $V^*$  is now the vector representing the optimal value function, the scalar  $\gamma$  is the discount rate,

$$R^{(a)} = \{R_i^{(a)} | R(\bar{x}, a) \text{ where } I(\bar{x}) = i\}$$

is the vector representing the reward function with respect to action  $a$ , and

$$T^{(a)} = \{T_{ij}^{(a)} | T(\bar{x}, a, \bar{x}') \text{ where } I(\bar{x}) = i \text{ and } I(\bar{x}') = j\}$$

is the transition probability matrix with respect to action  $a$ . Consider using the value iteration algorithm (introduced in Figure 2.4) which is an iterative method for solving the Bellman equation:

$$V_{I(\bar{x})}^{(n+1)} = \max_{a \in A} \left[ R_{I(\bar{x})}^{(a)} + \gamma \sum_{\bar{x}' \in \Omega_{\bar{x}}} T_{I(\bar{x})I(\bar{x}')}^{(a)} V_{I(\bar{x}')}^{(n)} \right].$$

The above iteration can be seen as first computing the  $Q$ -function vectors for every  $\bar{x} \in \Omega_{\bar{x}}$  and  $a \in A$ ,

$$Q_{I(\bar{x})}^{(n+1)}(a) = \left[ R_{I(\bar{x})}^{(a)} + \gamma \sum_{\bar{x}' \in \Omega_{\bar{x}}} T_{I(\bar{x})I(\bar{x}')}^{(a)} V_{I(\bar{x}')}^{(n)} \right]$$

and then combining all of the  $Q^{(n+1)}$  to obtain  $V^{(n+1)}$ :

$$\begin{aligned} V_{I(\bar{x})}^{(n+1)} &= \max_a \{ Q_{I(\bar{x})}^{(n+1)}(a) \} \\ &= \lim_{k \rightarrow \infty} \sqrt[k]{ \sum_a Q_{I(\bar{x})}^{(n+1)}(a)^k } \end{aligned}$$

Once again,  $Q^{(n+1)}(a)$  for a fixed action  $a$ ,  $V^{(n+1)}$  and  $V^{(n)}$  can be seen as vectors. Assume that vector  $Q^{(n+1)}(a)$  satisfies Equation 5.2 for partition  $P_a^{(n+1)}$ . Since the computation of the max operator involves  $Q^{(n+1)}(a)$  for all  $a \in A$ , the coarsest uniform partition of  $V^{(n+1)}$  will be of size  $O(\prod_a |P_a^{(n+1)}|)$ . This implies that in the worst case,  $n$ -th iteration takes time proportional to the product of the sizes of the representations for  $R^{(a)}$  and  $T^{(a)}$  for all action  $a$  so that it is exponential in the number of actions.

Simplifying somewhat, if we have an equation involving  $h$  distinct terms (matrices and vectors) each of which satisfies some variant of Equation 5.2 or Equation 5.3 for some partition  $P_i$ , then, in the worst case, solving the equation will require  $O(\prod_{i=1}^h |P_i|)$  time and space. To avoid this potential combinatorial explosion, we can take advantage of cases where the  $P_i$  agree on blocks or perform approximations that involve combining blocks in partitions whose associated values are approximately the same and then doing additional

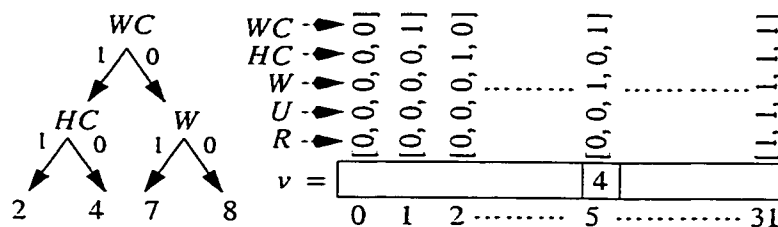


Figure 5.1: A vector represented as a decision tree.

bookkeeping to keep careful track of the resulting errors. One particular technique in the context of FMDPs can be found in Dearden and Boutilier [38].

In the remainder of this chapter, we describe procedures implementing elementary operators on matrices and vectors, applications using these elementary operators to solve problems posed in terms of linear algebra, and the use of techniques to avoid potential combinatorial explosion by trading space for accuracy.

## 5.2 Representing Matrices and Vectors

To illustrate our main points, we borrow the representation for a toy robot planning problem (see Figure 3.1) presented in Boutilier *et al.* [16]. Let  $R, U, W, HC, WC$  be our index variables, which are previously referred as fluents in the FMDP. Recall that these fluents represent, respectively, the weather outside being rainy, the robot having an umbrella, the robot being wet, the robot holding coffee, and the robot's boss wanting coffee. Let  $Z = [R, U, W, HC, WC]$  be a vector of variables ranging over  $\Omega_Z = \{0, 1\}^5$ .

In this thesis, we use decision trees and algebraic decision diagrams (ADDs) as our basic data structure for encoding functions that represent matrices and vectors. There are other methods for compactly representing functions such as rule-based methods that are often used in artificial intelligence to achieve compact representations (Poole [90]). The representation of linear algebra operators on matrices and vectors in ADDs are well studied in the area of VLSI design (Bahar *et al.* [3], Somenzi [100]). In the following we only describe representations and algorithms for the linear algebra operators of matrices and vectors in decision trees.

As an example, we can represent the vector  $V = (V_1, \dots, V_{32})$  corresponding to the reward function for a particular action in the robot planning problem as shown in Figure 5.1. In this example,  $I(0, 0, 1, 0, 1) = 5$  and  $V_5 = 4$ . Note that each terminal node in the decision tree represents a *set of indices* for the vector. This basic idea of representing the values corresponding to sets of indices is important for compactly representing large vectors and



matrices.

The techniques that we consider in this work compactly represent sets of indices whose corresponding entries in a matrix or vector have the same value. Two indices are said to be *equivalent with respect to value* if their corresponding entries are the same. This equivalence relation induces a partition on the set of all indices. Ideally, we would only want to allocate an amount of space polynomial in the number of index variables for each block in this partition. This amount of space would have to suffice for both the number of distinct values of the entry and for the size of the representation for the block. Fortunately, in some cases, we can represent large blocks of indices quite compactly. We may interpret the formula  $WC \wedge \overline{HC}$  as representing the set of all indices in which  $WC$  is assigned 1 and  $HC$  is assigned 0. Note that in Figure 5.1 we have achieved economy of representation by exploiting independence involving the index variables. For example, if  $WC$  is 1, then the value of the entry in  $V$  is independent of the assignment to  $W$ .

We can represent large matrices in a similar manner. The transition probabilities for the action of our robot going outside for a cup of coffee can be specified as a *two-stage temporal Bayesian network* (Dean and Kanazawa [35]) as shown in Figure 3.1. This network provides a compact encoding of the  $2^5 \times 2^5$  transition probability matrix. Each of the conditional probability tables (CPTs), one for each index variable, is encoded as a decision tree as suggested in Boutilier *et al.* [16]. The probability of ending up in one state having started from another is determined by the product of the values found in the decision trees shown in Figure 5.2, where, if  $i = I(1, 1, 1, 0, 0) = 28$  and  $j = I(1, 1, 0, 0, 0) = 24$ , then  $A_{ij} = \Pr[R_{t+1} \wedge U_{t+1} \wedge W_{t+1} \wedge \overline{HC}_{t+1} \wedge \overline{WC}_{t+1} | R_t \wedge U_t \wedge \overline{W}_t \wedge \overline{HC}_t \wedge \overline{WC}_t] = 0.01$ .

Let  $A = \{A_{ij}\}$  so that  $A_{ij}$  can be represented as a function of the index variables

$$f(Z_1, \dots, Z_n; Z'_1, \dots, Z'_n)$$

where  $i = I(Z_1, \dots, Z_n)$  and  $j = I(Z'_1, \dots, Z'_n)$ . In general, if we wish to represent a linear transformation from  $\mathfrak{R}^{2^n}$  to  $\mathfrak{R}^{2^n}$  as a  $2^n \times 2^n$  matrix, we assume that the corresponding function on  $\Omega_{\vec{z}} \times \Omega_{\vec{z}}$  can be factored as a certain combination of functions defined on much smaller spaces. Specifically, we assume that

$$f(Z_1, \dots, Z_n; Z'_1, \dots, Z'_n) \equiv \odot_k f_k(Z_k; \vec{Y}_k)$$

where  $\odot$  is a binary operator and the set of fluents  $\vec{Y}_k$  is a subset of  $[Z'_1, \dots, Z'_n]$ .<sup>1</sup> In the

<sup>1</sup>To cover more general factored representations of functions, we would have to represent each  $f_k$  in the form  $f_k(Z_k; U_k; W_k)$  where  $U_k \subseteq [Z_1, \dots, Z_n] \setminus Z_k$  and  $W_k \subseteq [Z'_1, \dots, Z'_n]$ .  $U_k$  is a subset of feature variables used for index  $i$  excluding  $Z_k$ , and  $W_k$  is a subset of feature variables used for index  $j$ . Using the less expressive representation simplifies our presentation without losing important details.

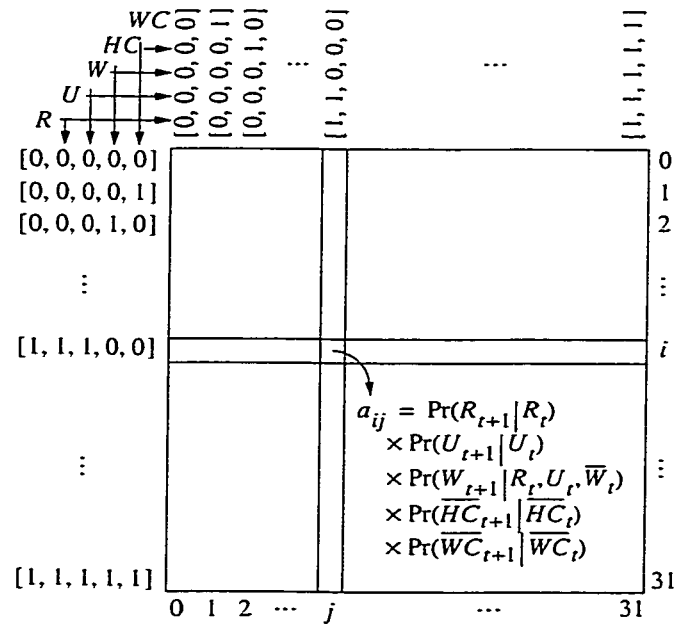


Figure 5.2: Transition matrix for a robot action.

example above,  $\odot$  is scalar multiplication but other operators can be used in other applications. In the case of computing the transitive closure of a large graph represented as an adjacency matrix,  $\odot$  becomes the maximum operator, and in the case of representing additive, multi-attribute value functions,  $\odot$  becomes the summation operator. In the following, we assume that each  $f_k$  is represented as a decision tree.

### 5.3 Elementary Structured Operators

One of the most basic operators on decision trees will be to *graft* one decision tree onto another by attaching a copy of the second to all of the terminal nodes of the first then simplifying the resulting decision tree by removing redundant or useless branches. In terms of partitions, each decision tree represents a partition and grafting one decision tree onto another is equivalent to intersecting the two partitions. The following pseudo-code for the function `TreeAttach` implements this operator minus the steps for simplification.

```

func TreeAttach(upper : terminal, lower : terminal, f : func)
    return f(upper, lower)

```

```

func TreeAttach(upper : terminal, lower : tree, f : func)
  let newresult :=
    foreach child of root(lower) do
      TreeAttach(upper, child, f)
  if children of root(newresult) are the same
    return one of the children of root(newresult)
  else return newresult

func TreeAttach(upper : tree, lower : terminal or tree, f : func)
  let newresult :=
    foreach child of root(upper)
      TreeAttach(child, lower, f)
  if children of root(newresult) are the same
    return one of the children of root(newresult)
  else return newresult

```

Using the function `TreeAttach` we can easily implement vector addition, subtraction, and inner product. Vector addition is illustrated in Figure 5.3 with an example from the robot domain. Our implementation takes as its inspiration the algorithm described in Boutilier *et al.* [16] for performing “structured” value iteration to solve large Markov decision processes. Here we adapt their basic methods for manipulating decision trees representing conditional probability tables to implement many of the basic vector and matrix operators familiar in linear algebra.

The term “structured” refers to the fact that, in order to represent the matrices and vectors economically, we exploit the regularities in the domain as in Boutilier *et al.* Following their lead, we refer to both our elementary vector-vector and matrix-vector operators as *structured operators* and the composite methods based on them as *structured methods*.

```

func StructuredPlus( U : terminal or tree, V : terminal or tree)
  return TreeAttach(U, V, +)

func StructuredSubtract( U : terminal or tree, V : terminal or tree)
  return TreeAttach(U, V, -)

```

Inner product is handled similarly. We describe how the operation is done by using the example in Figure 5.4 in the framework of the coffee robot domain (Figure 3.1). When

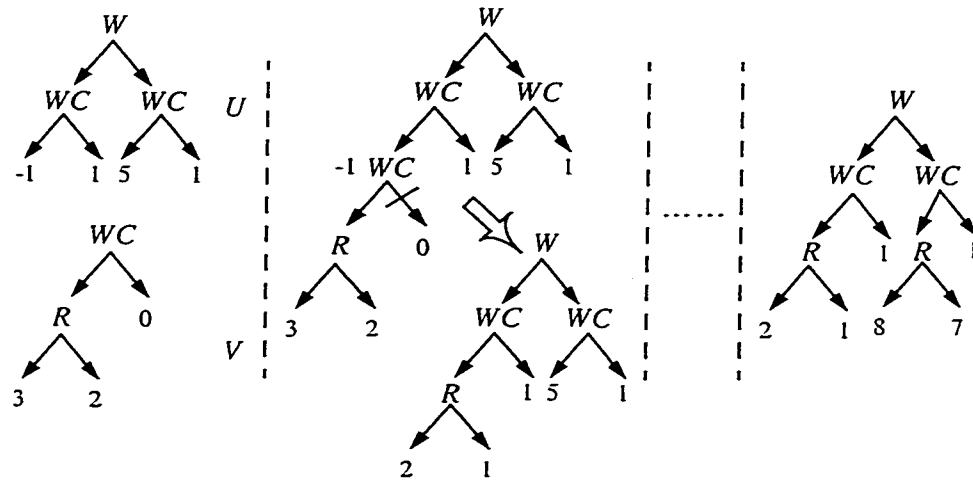


Figure 5.3: Adding two vectors represented as decision trees.

grafting one decision tree to the other, instead of taking the sum as in vector addition, we multiply the item stored in the grafted node to each item in the terminal node of the grafting decision tree: for the example in the figure, we multiply  $-1$  stored in  $WC$  to each item in the terminal node of  $V$ . We keep doing this for each terminal node of  $U$ , until we get the resulting decision tree on the rightmost side.

The final step is to take the sum of the items in the terminal nodes of the decision tree, taking into account that the terminal nodes represent blocks of indices. Since we know the index variables defining the problem, and we know the description of the block corresponding to each terminal node, we can calculate the number of individual indices for each terminal node. In the coffee robot domain example, we had 5 boolean index variables. Thus,  $U^T V = 2^2(-3) + 2^2(-2) + 2^3(1) + 2^2(15) + 2^2(10) + 2^3(1) = 96$ .

We note that given two vectors  $U$  and  $V$  with the number of terminal nodes  $n_U$  and  $n_V$  respectively, the inner product can be done in  $O(n_U n_V)$  time. Shown below is the algorithm for the inner product operator.

```

func StructuredInnerProduct( $\{f_i\}$  : list of tree,  $U, V$  : terminal or tree)
  let result := TreeAttach( $U, V, \times$ )
  foreach terminal leaf in result do
    leaf = leaf  $\times$  Arity(IndexVars(domain) - Path(result, leaf))
  let sum := 0
  foreach terminal leaf in result do
    sum = sum + leaf
  return sum

```

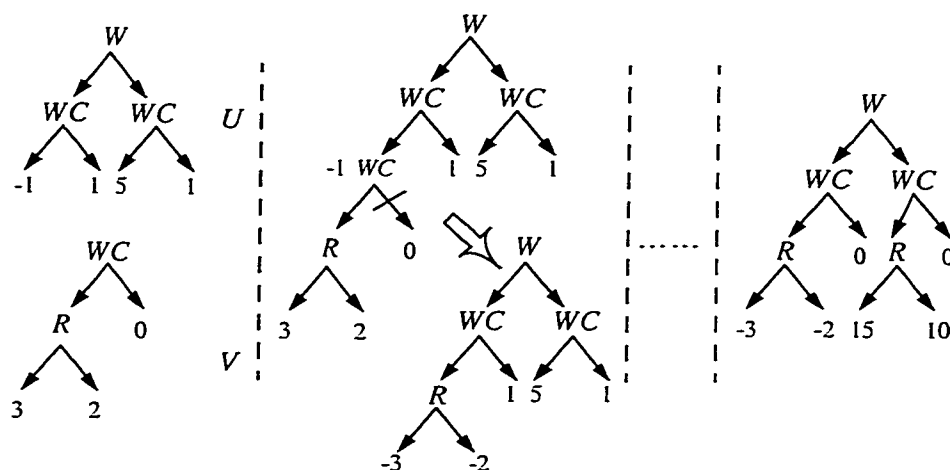


Figure 5.4: Building intermediate decision tree for taking inner product of two vectors represented as decision trees.

Another important structured operator is the multiplication of the transpose of a matrix and a vector. In the case in which the matrix corresponds to the transition matrix for an FMDP and the vector to a reward or value function, this operator is used inside the Bellman backup (Puterman [93]). Figures 5.5 and 5.6 illustrate some of the steps in the application of the following procedure with the vector  $A$  as shown in Figure 5.5 and the matrix as described in Figures 3.1 and 5.2.

```

StructuredTMultiply({ $f_i$ } : list of tree,  $V$  : tree)
  let result := EmptyTree()
  foreach index variable  $Z_k$  in  $V$  do
    foreach terminal leaf in result do
      leaf := TreeAttach(leaf,  $f_k$ , Append)
    result := Simplify(result)
  foreach terminal leaf in result do
    leaf := Collapse(leaf,  $V$ )
  return result

```

The first loop of StructuredTMultiply constructs the rightmost decision tree in Figure 5.5. Simplify eliminates terminal nodes that correspond to empty sets of indices and removes redundant branches. These eliminated terminal nodes at the end of paths corresponding to contradictory assignments to the index variables. The last loop in StructuredTMultiply constructs the final output using the intermediate result obtained from the first loop; here is how Collapse calculates the value stored at a terminal node.

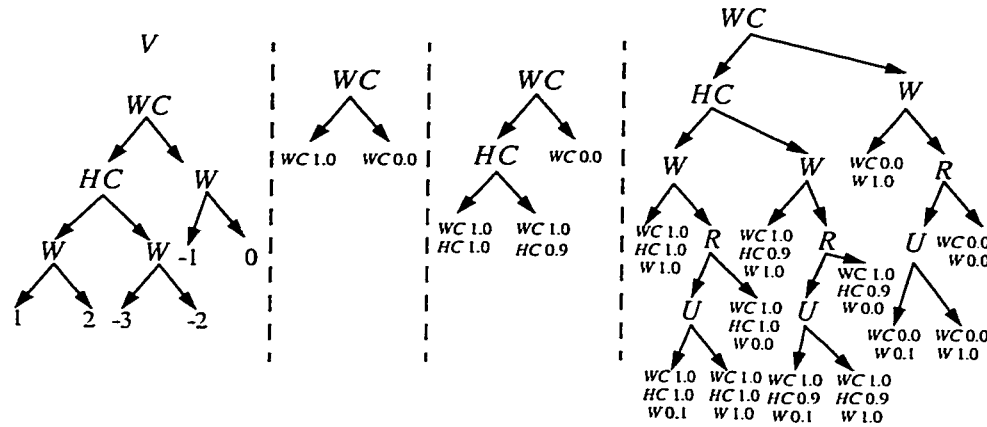


Figure 5.5: Intermediate steps in multiplying the transpose of a matrix with a vector.

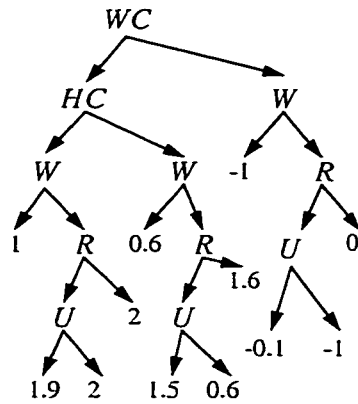


Figure 5.6: Vector resulting from Figure 5.5.

Let  $m \leq n$  so that  $\vec{Z}_V = [Z_{k_1}, \dots, Z_{k_m}]$  be the set of index variables used in the decision tree representation of vector  $V$ . The decision tree induces a partition  $P$  of the indices. Associated with each block  $B \in P$  is an assignment to fluents in  $\vec{Z}_V$ ; let  $z_{k_i, B}$  be the value assigned to  $Z_{k_i}$  by  $B$  and  $V_B$  be the common value assigned to all the indices in block  $B$ . Given the matrix represented as a set of local functions  $\{f_1(Z_1; \vec{Y}_1), \dots, f_n(Z_n; \vec{Y}_n)\}$ , let  $\{f_{k_1}(Z_{k_1}; \vec{Y}_{k_1}), \dots, f_{k_m}(Z_{k_m}; \vec{Y}_{k_m})\}$  be the subset of functions represented as decision trees associated with the  $\vec{Z}_V$ , and without loss of generality let  $\Omega_{Z_{k_i}} = \{1, \dots, |\Omega_{Z_{k_i}}|\}$ . Finally, let vector  $\vec{y}_{k_i, h}$  be the assignment of values to  $\vec{Y}_{k_i}$  determined by the path from the root of the tree to the  $h$ -th terminal node of the intermediate decision tree obtained at the end of the first loop. Just prior to the second loop, the  $h$ -th terminal node contains the following

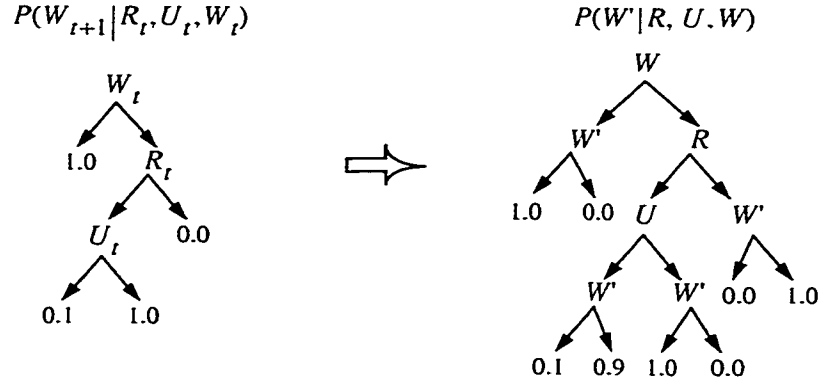


Figure 5.7: Modification of CPT for  $W$  by introducing the index variable at the next time step.

information <sup>2</sup>:

$$\begin{aligned}
 & [f_{k_1}(1; \vec{y}_{k_1,h}), \dots, f_{k_1}(|\Omega_{k_1}|; \vec{y}_{k_1,h})] \\
 & \quad \vdots \\
 & [f_{k_m}(1; \vec{y}_{k_m,h}), \dots, f_{k_m}(|\Omega_{k_m}|; \vec{y}_{k_m,h})].
 \end{aligned}$$

The procedure Collapse takes a terminal node and replaces the above information with a single number:  $\sum_{B \in \mathcal{P}} \prod_{i=1}^m f_{k_i}(z_{k_i,B}; \vec{y}_{k_i,h}) V_B$ .

Finally, we define a structured operator for the multiplication of a matrix and a vector. Given that we are multiplying matrix  $A$  with vector  $V$ , the  $i$ -th component of the vector obtained by multiplying matrix  $A$  and vector  $V$  is given as

$$\begin{aligned}
 (AV)_i &= \sum_j A_{ij} V_j \\
 &= \sum_j f(I^{-1}(i), I^{-1}(j)) V_j \\
 &= \sum_j \prod_k f_k(I^{-1}(i)_k, I^{-1}(j)) V_j.
 \end{aligned} \tag{5.4}$$

The algorithm for carrying out the above equation is another decision tree manipulation algorithm. Again, we explain the algorithm in terms of the FMDP in Figure 3.1 with an example in Figure 5.8. First, assume that the vector  $V$  we are given as input is the leftmost tree in the figure. We want to calculate how the index variables at the next time step will be affected after the vector is multiplied by the transition probability matrix. We set an

<sup>2</sup>Note that for the terminal nodes in Figure 5.5, given that the index variables are binary and the functions are probability distributions, we need only indicate a value for  $Z_k$ , say  $p$ , since the value for  $\bar{Z}_k$  is  $1 - p$ .

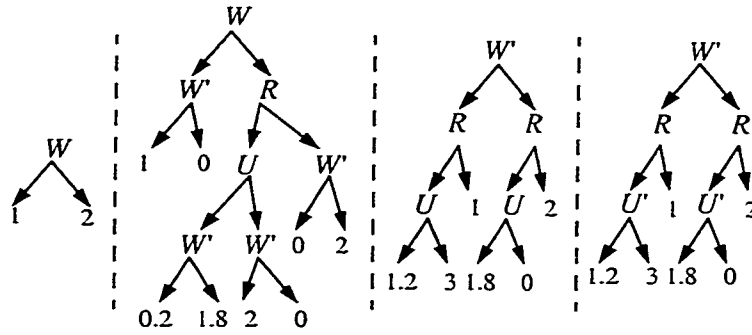


Figure 5.8: Multiplication of the transition probability matrix with a vector (step 1).

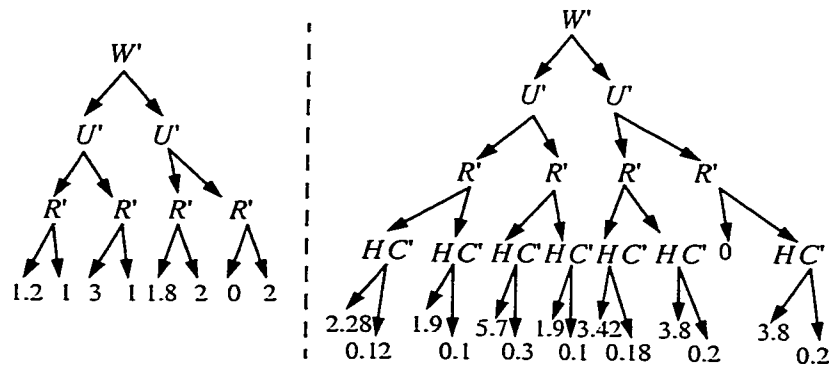


Figure 5.9: Multiplication of the transition probability matrix with a vector (step 2).

ordering for the index variables at the next time step: let's say we have  $[W, U, R, HC, WC]$ . We define the index variables at the next time step as primed variables, so that they are  $[W', U', R', HC', WC']$ . We pick up the first index variable in the ordering,  $W'$ , and attach its modified CPT to each terminal node of the decision tree. The modification of the CPT is done by adding into the decision tree the index variables at the next time step, so that the CPT specifies the probability explicitly in terms of the index variables at the next time step. Figure 5.7 shows the modification process for the index variable  $W'$ . After grafting the modified CPT to each terminal node of the decision tree, we get the decision tree shown in the second column of Figure 5.8. Note that the decision tree includes primed variables as non-terminal nodes. For example, for the intermediate decision tree in the second column, consider the left subtree of  $W$ . From the CPT for  $W'$ , we know that the probability for  $W'$  being true is 1 and that for  $W'$  being false is 0. Thus, when the CPT is grafted, the value at the next time step is either  $1 \times 1 = 1$  when  $W'$  is true and  $1 \times 0 = 0$  when  $W'$  is false. This grafting process corresponds to the product inside of the summation in Equation 5.4.



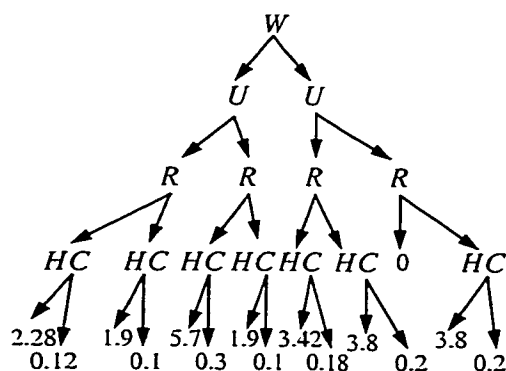


Figure 5.10: Left multiplication of the transition probability matrix with a vector (step 3).

Eventually, since we have to sum over the index variables at the current time step according to Equation 5.4, we will have to add two subtrees of  $W$ . In our case, since no other index variable depends on  $W$ , we can safely add the two subtrees of  $W$  resulting in the tree shown in the third column. However, if some other index variable depends on  $W$ , we cannot simply add subtrees since in the process of adding two subtrees we are essentially taking advantage of the associative rule  $(xy + zy) = (x + z)y$ .  $x$  and  $z$  can be seen as the values stored in left and right subtrees and  $y$  can be seen as the values that are going to be incorporated by the remaining CPTs. If the values that are going to be incorporated do not vary, we can perform the summation early. We then select  $U'$  and repeat the same operation. The result so far is shown in the last column of Figure 5.8.

The left decision tree in Figure 5.9 shows the tree after the CPT for  $R'$  is grafted. Exploiting conditional independence we perform some simplifications. The CPT for  $HC'$  is then grafted and the result is shown in the right side of Figure 5.9. Since the last index variable  $WC'$  does not depend on any other variable and it is persistent, we can skip grafting its CPT.

The final result is shown in Figure 5.10 where we eliminate the primes, since every index variable in the tree is the index variable at the next time step, and the next time step has become the current time step.

The following is a pseudo-code of the algorithm for structured matrix-vector multiplication. Note that the index variables at the next time step in FMDPs correspond to rows, and those at the current time step correspond to columns in general matrices. Hence, we use the term *row index variables* for the primed index variables, and *column index variables* for the original index variables:

```

func StructuredMultiply( $\{f_i\}$  : list of tree,  $V$  : terminal or tree)
  let result :=  $V$ 
  foreach  $k = 1, \dots, n$  do
    let  $f'_k$  be the modified tree of  $f_k$  by introducing the row index variable  $Z'_k$ 
    result := TreeAttach(result,  $f'_k$ ,  $\times$ )
    foreach node  $v$  in result do
      let  $X$  be the variable that  $v$  represents.
      if  $X$  is a row index variable then continue
      if no CPT in  $\{f_{k+1}, \dots, f_n\}$  has a node with variable  $X$  then
        replace  $v$  by StructuredPlus on subtrees of  $v$ .
  return result

```

Using ADDs in FMDPs, Hoey *et al.* [53] and St-Aubin *et al.* [101] construct the complete matrix by finding the ADD representation of  $\prod_{k=1}^n f'_k$ , and then multiply the vector and sum over the column index variables. Constructing the complete matrix using decision trees is prohibitive since the complete decision tree representation tends to explode in the size of the representation. StructuredMultiply using decision trees is an incremental version of their work in the sense that it interleaves multiplying  $f'_k$  and summing over the column index variables, trying to prevent the explosion in the size of the tree.

These structured operators with decision trees take time and space polynomial in the size of the resulting decision tree. When we use ADDs,

## 5.4 Approximating Vectors with Large Representation Size

We mentioned that the intermediate or final values of a calculation involving structured operators could explode such that the size of the representation of the coarsest uniform partition with respect to the terms involved in the calculation could grow to be of a size exponential in the total number of terms. This is not a problem in the case of calculating the value of a component such as  $(AU)_i$  or  $(A^kU)_i$ , but can, for example, lead to problems in solving Equation 5.1.

To avoid such combinatorial explosions, we use approximation methods to make sure that the sizes of intermediate results remain tractable. These methods involve combining blocks in partitions whose associated values are *approximately* the same. Specifically, we apply standard pruning technique similar to those used in learning decision trees (Quinlan [94]). The basic idea when applied to elementary structured operators on decision trees

is that whenever the size of the decision tree surpasses a certain threshold, we replace some of the subtrees by terminal nodes containing the averages. Since some indices are treated the same whose entries are in fact different, these methods introduce errors into calculations. Our methods are heuristic as it is easily shown that finding the smallest representation that achieves a given error is NP-hard (See Hyafil and Rivest [57]. Goldsmith and Sloan [45] investigate the complexity for general propositional formulas). However, by keeping track of bounds on the values of the entries for indices in each block of the partitions, or equivalently, each terminal node of the decision trees, we can report bounds on the error in the final results.

Whenever we deal with numerical calculations that introduce errors, we have to address issues of numerical stability, convergence, and precision. In Section 5.5, we consider several applications of our structured methods accompanied by appropriate numerical analyses. In particular, we show how to make use of traditional methods for analyzing the consequences of floating point underflow and overflow (Wilkinson [110]) to study convergence and precision in the use of structured operators. We also look at some ideas from the machine learning literature and consider how they can be used to analyze the result of applying pruning techniques to tree representations of vectors and matrices.

In preparation for the analyses of Section 5.5, the following two subsections provide some very basic background on modern error analysis and a quick introduction to some useful ideas from machine learning using function approximation.

#### 5.4.1 Error Analysis from Numerical Techniques

A great deal of attention in numerical analysis has been focused on solving systems of linear equations. Since not all real numbers can be represented exactly (or compactly) on computers, researchers in numerical analysis must carefully consider the effect of rounding errors in each algorithm proposed for solving systems of linear equations. The basic theory makes use of the *condition number* of a matrix. Given a matrix norm  $\|\cdot\|$  (we refer to Kincaid and Cheney [69] for a detailed introduction), the condition number of matrix  $A$  is defined as

$$\kappa(A) \equiv \|A\| \|A^{-1}\|.$$

The meaning of this quantity is revealed by the following question:

Given a system of linear equations  $Ax = b$ , suppose that by some perturbation,  $b$  becomes  $\tilde{b}$ . If  $\tilde{x}$  satisfies  $A\tilde{x} = \tilde{b}$ , how much do  $x$  and  $\tilde{x}$  differ in terms of norm  $\|\cdot\|$ ?

Assuming that  $A^{-1}$  exists, we have

$$\begin{aligned}\|x - \tilde{x}\| &= \|A^{-1}b - A^{-1}\tilde{b}\| \leq \|A^{-1}\| \|b - \tilde{b}\| = \|A^{-1}\| \|Ax\| \frac{\|b - \tilde{b}\|}{\|b\|} \\ &\leq \|A^{-1}\| \|A\| \|x\| \frac{\|b - \tilde{b}\|}{\|b\|}\end{aligned}$$

which leads to

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \|A^{-1}\| \|A\| \frac{\|b - \tilde{b}\|}{\|b\|} = \kappa(A) \frac{\|b - \tilde{b}\|}{\|b\|}.$$

From the above inequality, we see that  $\kappa(A)$  serves as an upper bound on the difference between the solutions,  $x$  and  $\tilde{x}$ , with respect to perturbations in the right-hand side of the equation  $Ax = b$ . If  $\kappa(A)$  is large, small perturbations lead to big differences in the solutions. Often, the perturbations are formulated in terms of a matrix  $E$  (particularly for those methods that calculate  $A^{-1}$  directly), so that we are solving

$$(A + E)\tilde{x} = b$$

such that  $E\tilde{x} = b - \tilde{b}$ . The advantage of using a matrix  $E$  is that, in some cases, we can easily denote the error incurred by inaccurate arithmetic operators. For example, when we use the Gaussian elimination method to solve a system of equations, where we compute  $L$  and  $U$  matrices, we set

$$LU \equiv A + E.$$

If every element of  $U$  is in  $[-1, 1]$ , then every element of  $E$  is guaranteed to be between  $[-\frac{1}{2}\epsilon, \frac{1}{2}\epsilon]$ , where  $\epsilon$  denotes the machine precision (Wilkinson [110]). The relative difference in the solutions can be also derived using  $E$ :

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \|I - (A + E)^{-1}A\| \leq \frac{\|E\| \|A^{-1}\|}{1 - \|E\| \|A^{-1}\|}$$

Note that  $\kappa(A)$  denotes a property of the matrix  $A$  which is independent of the accuracy of the arithmetic operators, whereas  $E$  denotes the error incurred during the method, which is dependent on *both*  $\kappa(A)$  and the accuracy of the arithmetic operators. Although the classical use of  $E$  deals with machine precision, we will see in subsequent sections that this basic approach can be extended to analyze the errors that result from some classes of approximate structured operators.

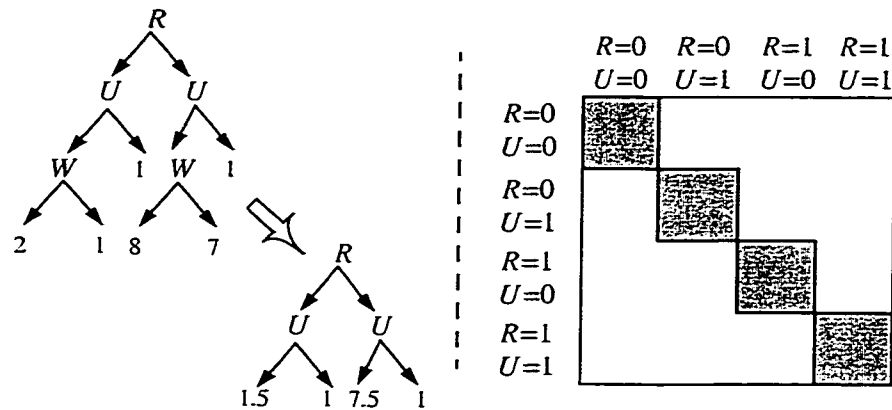


Figure 5.11: An example illustrating the pruning process and the resulting averager matrix. The example follows the domain given in Figure 3.1. The four diagonal blocks are of size  $8 \times 8$  and all of their components are  $1/8$ .

### 5.4.2 Averagers and Stable Approximations

Another way to capture errors that result from certain types of approximation is developed in the work of Gordon [47] and involves the notion of an *averager* (Definition 2.4.1). The most important task in building the connection between the decision tree pruning method and the notion of an averager is to show that the pruning method can be represented as multiplying a matrix  $A$  with the property  $\|A\| \leq 1$ . Figure 5.11 shows a simplified case where the pruning process (replacing certain subtrees by the *averages* of their terminal nodes) can be represented as an averager. Note that in this example, the non-zero elements are gathered along the diagonal. The ordering of the index variables is carefully chosen for the sake of visualization. Although we cannot find in general such a nice diagonalization, we can observe that the ordering of the variables  $R$  and  $U$  does not change the property that  $\|A\| \leq 1$ .

We need a few additional restrictions on the pruning method to fully comply with the definition of an averager and stable approximation (Theorem 2.4.1):

- When replacing a subtree with the average of the terminal nodes, we can take the simple average or the weighted average. Recall that a terminal node represents a set of indices. If we take the weighted average by weighing each value stored at the terminal node by the number of represented indices, the sum of each row of the matrix  $A$  is 1. If we take the simple average, the sum of each row of the matrix  $A$  is less than or equal to 1. In both cases,  $\|A\| \leq 1$ , and thus we obtain a stable approximation.
- Instead of using the classical pruning method that examines every subtree at every

non-terminal node, we prepare a set of index variables that should be eliminated during the pruning process before using any approximate structured method. For example, in Figure 5.11, the classical pruning method eliminates the non-terminal nodes with index variable  $W$  before any other non-terminal nodes since non-terminal nodes with  $W$  are located lower than any other non-terminal nodes. We should obtain the same result if our elimination variable set is  $\{W\}$ . If our elimination variable set was, say,  $\{U\}$ , the decision tree after the pruning should have only  $R$  and  $W$  as its non-terminal nodes. This strategy is similar to Dearden and Boutilier’s [38] strategy of preparing the set of “relevant” variables in approximate abstraction of FMDPs.

This strategy is to make sure that the matrix  $A$  remains the same throughout the structured method. If we relax this restriction, the iterative structured method introduced in subsequent sections may show oscillation instead of convergence, as observed in Boutilier and Dearden [15].

## 5.5 Applications and Analyses

There is a large class of numerical algorithms that are described and implemented in terms of elementary operators on matrices and vectors. With the structured linear algebra operators for matrices and vectors defined in the previous sections, we can extend many of these algorithms to handle large structured matrices and vectors. In this section, we provide examples of these extensions and analyze the resulting algorithms.

We note that some algorithms such as Gaussian elimination cannot easily be implemented using the data structures and basic operators that we develop in this thesis. In particular some operations common in manipulating explicit matrices, such as exchanging columns or rows, are difficult to carry out in tree representations. We do not consider extending some popular algorithms that are known for being efficient and numerically stable, such as  $LU$  decompositions, because of the difficulty of implementing such operations. Instead, we focus on iterative algorithms based on the elementary operators for explicit matrices and vectors that we extended to structured matrices and vectors in Section 5.3.

### 5.5.1 Solving Systems of Linear Equations

Solving systems of linear equations is the most fundamental problem in linear algebra and an often required subroutine in many numerical procedures. Algorithms for solving systems of linear equations are used in a wide range of problems in artificial intelligence, operations

research, economics, and other scientific computing.

There are many algorithms suggested for solving systems of linear equations, such as Gaussian elimination method, matrix decomposition methods, and iterative methods. However, as we have claimed in the earlier part of the section, we concentrate on iterative methods for solving systems of linear equations since they are usually specified in terms of matrix-vector multiplication and vector-vector addition operators, and we have structured operators for doing these operations. We begin with one of the simplest iterative methods used in solving MDPs:

**Policy Evaluation using Bellman Backup** Suppose that we are given an MDP  $M$ , a discount rate  $\gamma$ , and a policy  $\pi$ , and we wish to evaluate the given policy. The value of the policy  $\pi$ , denoted  $V^\pi$ , is defined by the following system of equations represented in matrix form:

$$(I - \gamma T_\pi)V^\pi = R_\pi.$$

where  $I$  is a  $|S| \times |S|$  identity matrix,  $T_\pi$  is the transition probability matrix corresponding to the policy  $\pi$ , and  $R_\pi$  is the reward vector corresponding to the policy  $\pi$ . It can be shown that  $V^\pi$  satisfies

$$V^\pi = R_\pi + \gamma T_\pi V^\pi \tag{5.5}$$

and that  $V^\pi$  is the unique fixed point of the following iterative method

$$V_\pi^{(n+1)} = R_\pi + \gamma T_\pi V_\pi^{(n)}. \tag{5.6}$$

Suppose that  $T_\pi$  and  $R_\pi$  are represented as a 2TBN and a tree, and the matrix and vector operations are defined as in Section 5.3, then the computational cost of each iteration is polynomial in the product of the sizes of  $T_\pi$ ,  $V_\pi^{(n)}$  and  $R_\pi$ . The following pseudo-code is the iterative method of Equation 5.6 using structured linear algebra operators:

```
func StructuredBellmanBackup( $\{f_i\}$  : list of tree,  $R_\pi$  : tree,  $V$  : tree)
  let result := StructuredTMultiply( $\{f_i\}$ ,  $\gamma \cdot V$ )
  return StructuredPlus(result,  $R_\pi$ )
```

Note that this procedure is exactly the Bellman backup with a fixed policy. To multiply a vector represented as a decision tree by a constant, we multiply every terminal item with the constant.

**Richardson method and its acceleration** The Bellman backup with a fixed policy in the previous section is a special case of more general method called the Richardson method used in solving systems of linear equations. We return to the general problem of solving a system of linear equations. Assume that we want to solve systems of linear equations given in a matrix-vector form

$$AV = U.$$

Many iterative methods are defined in terms of a *splitting* of  $A$  such that  $A = M - N$ . Using this splitting, the above equation is rewritten in an equivalent form:

$$MV = NV + U.$$

This equation readily suggests an iterative process, defined by

$$MV^{(n+1)} = NV^{(n)} + U. \quad (5.7)$$

Iterative methods that split the coefficient matrix  $A$  are based on the following fundamental theorem (Kincaid and Cheney [69]):

**Theorem 5.5.1** *If  $\|M^{-1}N\| < 1$  for some subordinate matrix norm, then the sequence produced by Equation 5.7 converges to the solution of  $AV = U$  for any initial vector  $V^{(0)}$ .*

The simplest of the iterative methods is the Richardson method where  $M \equiv I$  and  $N \equiv I - A$  (Kincaid and Cheney [69]). If we know that  $\|I - A\| < 1$ , we can apply the following algorithm for the matrix  $A$  (represented as a set of decision trees  $\{f_i\}$ ) and vector  $V$ . The Max operator that appears in the pseudo-code is maximum of the absolute values of the entries stored in the leaves of the tree.

```

func StructuredRichardson( $\{f_i\}$  : list of tree,  $U$  : tree,  $\epsilon$  : real)
  let  $V_{old} :=$  null vector,  $V_{new} := U$ 
  while Max(|StructuredSubtract( $V_{old}$ ,  $V_{new}$ )|)  $> \epsilon$  do
     $V_{old} := V_{new}$ 
     $V_{new} :=$  StructuredSubtract( $V_{new}$ ,
      StructuredMultiply( $\{f_i\}$ ,  $V_{new}$ ))
     $V_{new} :=$  StructuredPlus( $V_{new}$ ,  $U$ )
  return  $V_{new}$ 

```



In a similar manner, we can implement variants of iterative methods such as Jacobi, Gauss-Seidel, and successive over-relaxation methods using elementary structured operations and some special operators to handle the splittings. We can also implement structured versions of the bi-conjugate gradient descent method, which falls into another category of iterative methods for solving linear equations.

To introduce acceleration techniques, note that manipulating Equation 5.7 yields

$$\|V^{(n+1)} - V^\dagger\| \leq \|M^{-1}N\| \|V^{(n)} - V^\dagger\|,$$

where  $V^\dagger$  is the vector that satisfies  $AV^\dagger = U$ . The above inequality implies that the speed of convergence is determined by  $\|M^{-1}N\|$  since

$$\|V^{(n)} - V^\dagger\| \leq \|M^{-1}N\|^n \|V^{(0)} - V^\dagger\|.$$

It is natural to try to find the best  $M$  and  $N$  such that  $\|M^{-1}N\|$  is minimal. Many acceleration techniques for iterative methods are implemented this way (Press *et al.* [92]).

The following is one of the acceleration techniques called the *extrapolation method*. Observe that Equation 5.7 can be rewritten as follows:

$$V^{(n+1)} = M^{-1}NV^{(n)} + M^{-1}U.$$

We introduce an acceleration parameter  $\lambda \neq 0$  to the above equation to obtain

$$\begin{aligned} V^{(n+1)} &= \lambda(M^{-1}NV^{(n)} + M^{-1}U) + (1 - \lambda)V^{(n)} \\ &= G_\lambda V^{(n)} + \lambda U' \end{aligned}$$

where  $G_\lambda \equiv \lambda M^{-1}N + (1 - \lambda)I$  and  $U' \equiv M^{-1}U$ . If the only information available about the eigenvalues of  $M^{-1}N$  is that they lie in the interval  $[a, b]$ , then the best choice for  $\lambda$  is  $2/(2 - a - b)$  (Kincaid and Cheney [69]). We can calculate the interval  $[a, b]$  as a preprocessing step using structured versions of the power and inverse power methods (see Section 5.5.2) to determine  $\lambda$  for our experiments.

Algorithms for solving a system of linear equations are used in MDPs for a number of purposes. In this work, we concentrated on applying equation-solving techniques to calculate the value function for a particular policy  $\pi$ . The Bellman equation for the value function  $V^\pi$ , which is

$$(I - \gamma T_\pi)V^\pi = R_\pi,$$

can be solved using the Richardson method with  $M \equiv I$  and  $N = \gamma T_\pi$  so that,

$$V_\pi^{(n+1)} = \gamma T_\pi V_\pi^{(n)} + R_\pi,$$

whereas the extrapolation method for calculating  $V^\pi$  is specified by

$$V_\pi^{(n+1)} = \lambda\gamma T_\pi V_\pi^{(n)} + (1 - \lambda)V_\pi^{(n)} + \lambda R_\pi.$$

Note that we can imagine using the extrapolation method to accelerate the value iteration algorithm. Formally, define the mapping

$$L_e V(s) \equiv \max_a [\lambda\gamma \sum_{s'} T(s, a, s') V(s') + (1 - \lambda)V(s) + \lambda R(s, a)].$$

To guarantee convergence of this method, we have to prove that  $L_e$  is a contraction mapping. In other words, we need to show that there exists a constant  $0 \leq \alpha < 1$  such that for any  $V, U \in \mathfrak{R}^{|S|}$ ,  $\|L_e V - L_e U\| \leq \alpha \|V - U\|$ . This is equivalent to showing that the matrix used in the extrapolation method for a fixed policy  $\pi$  defined by

$$G_\lambda^\pi \equiv \lambda\gamma T_\pi + (1 - \lambda)I$$

has the property  $\|G_\lambda^\pi\| < 1$  for any policy  $\pi$ . Without any prior information, we should assume that  $T_\pi$  can be any stochastic matrix, and this assumption implies that the maximum and the minimum eigenvalues of  $T_\pi$  are 1 and -1, respectively. Note that this makes our choice for  $\lambda$  be 1, thus we end up with value iteration. Thus, without knowing the maximum and minimum eigenvalues of  $T_\pi$  for all policy  $\pi$  (no prior information on the eigenvalues is a reasonable assumption), we cannot accelerate the value iteration algorithm by using the extrapolation method. In the subsequent sections, we only consider the case in which the extrapolation method is used as a policy evaluation method.

**Error analysis using rounding error technique** In the following, we carry out the analysis of using the approximate Bellman backup as the iterative method for calculating the value function under fixed policy  $\pi$ . By using approximate operators, which are the structured linear algebra operators with pruning, the Bellman backup with the fixed policy  $\pi$  can be viewed as

$$\tilde{V}_\pi^{(n+1)} = R_\pi + \gamma[T_\pi + E^{(n)}]\tilde{V}_\pi^{(n)} + U \tag{5.8}$$

where matrix  $E$  and vector  $U$  define the errors incurred during matrix-vector multiplication and vector-vector addition, respectively. Since the true solution satisfies Equation 5.5, we have

$$\|\tilde{V}_\pi^{(n+1)} - V^\pi\| \leq \gamma\|T_\pi\|\|\tilde{V}_\pi^{(n)} - V^\pi\| + \|\gamma E^{(n)}\tilde{V}_\pi^{(n)} + U\|$$

which leads to

$$\|\tilde{V}_\pi^{(\infty)} - V^\pi\| \leq \frac{\max_n \|\gamma E^{(n)} \tilde{V}_\pi^{(n)} + U\|}{1 - \gamma \|T_\pi\|}.$$

This inequality indicates that the approximate solution  $\tilde{V}_\pi^{(\infty)}$  is within  $\frac{\max_n \|\gamma E^{(n)} \tilde{V}_\pi^{(n)} + U\|}{1 - \gamma \|T_\pi\|}$  of the exact solution  $V^\pi$ .

Above analysis provides an alternative way to derive the error bound reported in Dearden and Boutilier [38] for evaluating abstract policies for MDPs. Their work constructs an abstract MDP by eliminating irrelevant or weakly relevant fluents in the original FMDP. Their error bound on the value function of an abstract policy, provided in Theorem 5.2 of Dearden and Boutilier, is defined in terms of the error in the reward function due to eliminating some of the fluents ( $U$  in Equation 5.8) and the error in the transition probabilities due to eliminating some of the fluents ( $E$  in Equation 5.8).

**Error analysis using averagers** We analyze the error incurred by using the approximate structured operators in evaluating a policy by the Richardson method, which is equivalent to Bellman backup, and the extrapolation method.

When the restrictions in pruning (Section 5.4.2) are satisfied, we can formulate the approximate Bellman backup as

$$\tilde{V}^{(n+1)} = A[R_\pi + \gamma AT_\pi \tilde{V}^{(n)}],$$

where matrix  $A$  represents the averager induced by pruning after each structured operator. Using the fact that  $V^\pi = R_\pi + \gamma T_\pi V^\pi$ , we have the following inequality:

$$\begin{aligned} \|\tilde{V}^{(n+1)} - AV^\pi\| &= \|\gamma A^2 T_\pi \tilde{V}^{(n)} - \gamma AT_\pi V^\pi\| \\ &= \|\gamma A^2 T_\pi \tilde{V}^{(n)} - \gamma A^3 T_\pi V^\pi + \gamma A^3 T_\pi V^\pi - \gamma AT_\pi V^\pi\| \\ &\leq \|\gamma A^2 T_\pi\| \|\tilde{V}^{(n)} - AV^\pi\| + \|A^2 - I\| \|\gamma AT_\pi V^\pi\|. \end{aligned}$$

We immediately notice that  $\|V^{(n)} - AV^\pi\|$  is bounded for all  $n$  due to the fact that  $\|\gamma A^2 T_\pi\| < 1$ . Thus,

$$\|V^{(\infty)} - AV^\pi\| \leq \frac{\|A^2 - I\| \|\gamma AT_\pi V^\pi\|}{1 - \|\gamma A^2 T_\pi\|}.$$

A similar analysis can be carried out under the extrapolation method. The approximate extrapolation method can be defined as

$$\begin{aligned} \tilde{V}_\pi^{(n+1)} &= A[\lambda \gamma AT_\pi \tilde{V}_\pi^{(n)} + (1 - \lambda) \tilde{V}_\pi^{(n)} + \lambda R_\pi] \\ &= \lambda \gamma A^3 T_\pi \tilde{V}_\pi^{(n)} + (1 - \lambda) A^2 \tilde{V}_\pi^{(n)} + \lambda A R_\pi. \end{aligned}$$

Using the fact that  $V^\pi = \lambda\gamma AT_\pi V^\pi + (1 - \lambda)V^\pi + \lambda R_\pi$ , we have

$$\begin{aligned} \|\tilde{V}_\pi^{(n+1)} - AV^\pi\| &= \|\lambda\gamma A^3 T_\pi \tilde{V}_\pi^{(n)} + (1 - \lambda)A^2 \tilde{V}_\pi^{(n)} - \lambda\gamma AT_\pi V^\pi - (1 - \lambda)AV^\pi\| \\ &\leq \|\lambda\gamma A^3 T_\pi \tilde{V}_\pi^{(n)} + (1 - \lambda)A^2 \tilde{V}_\pi^{(n)} - \lambda\gamma A^3 T_\pi AV^\pi - (1 - \lambda)A^3 V^\pi\| \\ &\quad + \|\lambda\gamma A^3 T_\pi AV^\pi + (1 - \lambda)A^3 V^\pi - \lambda\gamma AT_\pi V^\pi - (1 - \lambda)AV^\pi\| \\ &\leq \|A^2\| \|\lambda\gamma AT_\pi + (1 - \lambda)I\| \|\tilde{V}_\pi^{(n)} - AV^\pi\| \\ &\quad + \|\lambda\gamma A^3 T_\pi AV^\pi + (1 - \lambda)A^3 V^\pi - \lambda\gamma AT_\pi V^\pi - (1 - \lambda)AV^\pi\|. \end{aligned}$$

When  $0 \leq \lambda \leq 1$ , the convergence is guaranteed by

$$\|A^2\| \|\lambda\gamma AT_\pi + (1 - \lambda)I\| \leq \lambda\gamma + 1 - \lambda < 1,$$

since  $\gamma < 1$ . Thus we have,

$$\|\tilde{V}_\pi^{(\infty)} - AV^\pi\| \leq \frac{\|\lambda\gamma A^3 T_\pi AV^\pi + (1 - \lambda)A^3 V^\pi - \lambda\gamma AT_\pi V^\pi - (1 - \lambda)AV^\pi\|}{\|A^2\| \|\lambda\gamma AT_\pi + (1 - \lambda)I\|}.$$

Although the bound does not easily simplify, we can observe that the right hand side gets closer to 0 as  $A$  becomes closer to  $I$ .

**Experiments** As a simple experiment, we compare the performance of the structured Richardson method with the structured version of the acceleration technique, both of which were described in Section 5.5.1. The task was to estimate the value of always executing the action stay in Boutilier and Dearden [15]’s Coffee Robot domain. Table 5.1 and Table 5.2 show the number of iterations and execution time for the Richardson method and the extrapolation method with varying termination condition  $\epsilon$  and discount rate  $\gamma$ . The stopping criteria for the Richardson method is given by

$$\|V^{(n+1)} - V^{(n)}\| \leq \epsilon \frac{1 - \gamma}{2\gamma}$$

and the extrapolation method by

$$\|V^{(n+1)} - V^{(n)}\| \leq \epsilon \frac{1 - [1 - \lambda(1 - \gamma)]}{2[1 - \lambda(1 - \gamma)]}.$$

The latter inequality is derived from the fact that  $\|G_\lambda\| \leq 1 - \lambda(1 - \gamma)$  for our particular choice of  $\lambda$ .

For the Richardson method, one matrix-vector multiplication, one vector-vector addition, and one scalar-vector multiplication are performed at each iteration. For the extrapolation method, one matrix-vector multiplication, two vector-vector additions, and three scalar-vector multiplications are performed per iteration. This explains why the extrapolation method takes slightly longer running time per iteration than the Richardson method.

$\epsilon$	$\gamma$	Richardson method		Extrapolation method	
		Iterations	Run time (sec)	Iterations	Run time (sec)
0.1	0.8	31	37.02	11	13.48
0.1	0.9	72	87.97	23	29.63
0.1	0.95	160	197.66	49	64.57
0.05	0.8	34	40.80	12	14.76
0.05	0.9	78	95.43	25	32.13
0.05	0.95	174	215.05	53	69.76

Table 5.1: Experimental results with exact operations.

$\epsilon$	$\gamma$	Richardson method		
		# of iterations	Run time (sec)	Rel err
0.1	0.8	30	7.49	0.22
0.1	0.9	71	17.91	0.17
0.1	0.95	158	40.16	0.14
0.05	0.8	34	8.54	0.22
0.05	0.9	77	19.57	0.17
0.05	0.95	171	43.48	0.14
$\epsilon$	$\gamma$	Extrapolation method		
		# of iterations	Run time (sec)	Rel err
0.1	0.8	11	2.89	0.26
0.1	0.9	22	5.99	0.21
0.1	0.95	48	13.39	0.16
0.05	0.8	12	3.17	0.26
0.05	0.9	24	6.61	0.21
0.05	0.95	52	14.54	0.16

Table 5.2: Experimental results with approximate operations. “Rel err” denotes the relative error incurred by approximating the vector. The threshold  $\theta$  used for tree pruning algorithm heuristic was set to 1.0 throughout the experiments on the Richardson method and 5.0 on the extrapolation method.

However, we can see that the information about the largest and smallest eigenvalues allows us to apply the extrapolation method in *structured domains*, which is a clear advantage.

Table 5.2 shows the results for the same set of problems as in Table 5.1 but in this case the results are from the approximate Richardson and extrapolation methods with the decision tree pruning technique. The pruning is done using the classical pruning method, which does not use the elimination variable set: after each exact elementary structured operation, TreePrune takes the (possibly large) decision tree as an input and examines the leaves of the subtree rooted at each internal node. TreePrune goes through each node in a recursively bottom-up fashion. If the variance of the leaves is smaller than some threshold value given as input, the algorithm contracts the whole subtree to a single leaf node containing the

average. Otherwise, it proceeds to the parent node without any changes:

```

func TreePrune( $T$  : tree,  $\theta$  : real)
  let result :=
    foreach children child of  $T$  do
      TreePrune(child,  $\theta$ )
  let average := Average of leaf items in result
  let variance := Variance of leaf items in result
  if variance <  $\theta$  do
    return average
  else return  $T$ 

```

Table 5.2 also lists the relative error for the approximations. The relative error is defined to be

$$\|(V^\pi - \tilde{V}^\pi)/V^\pi\|_\infty$$

where  $V^\pi$  and  $\tilde{V}^\pi$  represent, respectively, the exact solution and the approximate value functions. We can observe that the relative error is not dependent on  $\epsilon$ , rather than  $\gamma$ . This is due to the fact that  $\epsilon$  serves as the stopping condition of the iterative methods. When the iterative methods stop, we know that  $V^{(n)}$  has converged to some solution, but the quality of the solution depends on the aggressiveness of the pruning. The dependence of relative error on  $\gamma$  can also be explained. A larger  $\gamma$  makes a larger change in  $V^{(n)}$  at each iteration. Since we start from a null vector whose components are all zeros, the larger change in  $V^{(n)}$  from the onset makes more resilient to the pruning, thus smaller relative error.

The effectiveness of this pruning technique is sensitive to the threshold. In Table 5.3, we show the number of leaves in the tree representing the value function produced as output and the relative error achieved by this tree. Note that the number of leaves in trees produced as output is sensitive to the threshold. In our experiments, running the approximate Richardson method with  $\theta = 5.0$  produced intermediate decision trees with only one terminal node. This is why we set different threshold values for the approximate Richardson method and the approximate extrapolation method. We suspect that such collapsing behavior is due to the difference in the rate of convergence from the onset. Note that in both methods, we start from a null vector, which has the same value for all components, and move towards the solution, which has different values for some of the components. Since the intermediate vectors from the Richardson method move slower to the solution, the Richardson method requires a smaller pruning threshold to obtain

$\theta$	Richardson method		Extrapolation method	
	Terminals	Rel err	Terminals	Rel err
0.0	30	0.003	30	0.003
0.1	30	0.003	30	0.003
0.5	30	0.157	30	0.087
1.0	6	0.218	30	0.104
2.0	6	0.519	30	0.173
3.0	6	0.519	6	0.261
4.0	1	0.885	6	0.261
5.0	1	0.878	6	0.261

Table 5.3: Experimental results with approximate operations with classical pruning method.  $\theta$  is the threshold used for pruning. “Rel err” denotes the relative error incurred by approximating the vector.  $\epsilon$  and  $\gamma$  were set to 0.1 and 0.8, respectively.

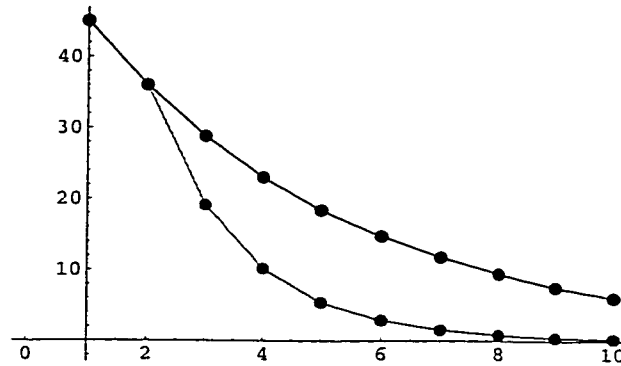


Figure 5.12: The plot of  $\|V^\pi - V^{(n)}\|$  of the Richardson and the extrapolation method at each iteration. The graph showing the slower convergence (upper curve) is the plot from the Richardson method.

(roughly) equivalent pruning as that of the extrapolation method. Figure 5.12 is the plot of  $\|V^\pi - V^{(n)}\|$  for the exact Richardson and the extrapolation methods. Observe that the extrapolation method moves toward the true value function faster than the Richardson method from the second iteration. The second iteration changes the  $V^{(n)}$  so significantly that the variance in the subtree is above the pruning threshold  $\theta$ .

Table 5.4 shows the same experiments with pruning with the various sets of elimination index variables. The Richardson and the extrapolation methods show virtually same performance in terms of the relative error they achieve. However, the comparison in the number of iterations shows that the extrapolation method is faster than the Richardson method. There is not much difference in the number of iterations until all index variables are selected to be eliminated. This behavior can be explained by the fact that the number of iterations is usually dependent on the discount rate  $\gamma$  (Trick and Zin [104]).

Elimination variables	Richardson method			Extrapolation method		
	Iterations	Terminals	Rel err	Iterations	Terminals	Rel err
<i>T</i>	30	6	0.218	11	6	0.217
<i>M</i>	31	20	0.370	11	20	0.370
<i>C</i>	30	15	0.282	11	15	0.282
<i>T, L</i>	30	6	0.218	11	6	0.217
<i>T, L, HC</i>	30	6	0.218	11	6	0.217
<i>T, L, HC, HM</i>	30	2	0.512	11	2	0.513
<i>T, L, HC, HM, M</i>	30	4	0.477	11	4	0.478
All	1	1	0.878	1	1	0.714

Table 5.4: Experimental results with approximate operations with pruning using elimination variables. “Rel err” denotes the relative error incurred by approximating the vector.  $\epsilon$  and  $\gamma$  were set 0.1 and 0.8, respectively.

$\epsilon$	$\gamma$	Richardson method		Extrapolation method	
		Iterations	Run time (sec)	Iterations	Run time (sec)
0.1	0.8	15	1.84	10	1.44
0.1	0.9	32	3.86	20	3.02
0.1	0.95	66	7.52	40	4.49
0.05	0.8	18	2.16	11	1.50
0.05	0.9	38	4.31	23	2.93
0.05	0.95	79	8.64	47	5.48

Table 5.5: Experimental results with ADDs on the FACTORY domain.

Table 5.5 shows the same experiments as Table 5.1 on a large domain using ADDs. The FACTORY domain has 17 binary variables, so the value function vector has 131,072 components. The policy we used was executing the action *polishb*. The CUDD package (Somenzi [100]) already provides a set of linear algebra operators for ADDs, so we used its ADD operators to implement the structured version of the Richardson method and the extrapolation method. Throughout the experiments, the resulting ADD consistently had the total of 29 nodes (8 terminal nodes) for both methods. The ADD experiments were implemented in C++ and the decision tree experiments were implemented in Mathematica. Thus, when comparing the running times in Table 5.5 and Table 5.1, we should take into account that ADD experiments were much faster due to both the effectiveness of ADDs and the optimization of the code.

### 5.5.2 Calculating Eigensystems

The eigensystem (eigenvalues and eigenvectors) of a matrix reveals a lot of useful information about problems described in terms of the matrix. For example, in a Markov chain, the



second largest eigenvalue provides a measure of the *mixing rate* of the process: how fast a random walk converges to the steady state distribution. Also, if none of the eigenvalues is  $-1$ , then the chain is called *ergodic* and is thus known to have several desirable properties.

In this section, we show how to calculate the eigensystem of a large matrix using structured linear algebra operators. We also show its application to compute the singular values of a large matrix.

**Power Method** There are a lot of numerical algorithms for calculating the eigensystem of a matrix. In the following, we illustrate a structured version of the *power method*, which is an iterative method for finding largest eigenvalue and its corresponding eigenvector. The power method is an iterative method defined by

$$V^{(n+1)} = AV^{(n)}$$

given a matrix  $A$ . The power method utilizes the following property:

**Theorem 5.5.2 (Power Method)** *For an arbitrary matrix  $A$  and an initial vector  $V^{(0)}$ , the vector*

$$V^{(n+1)} = AV^{(n)}$$

*aligns with the eigenvector  $U_1$  corresponding to the largest eigenvalue  $\lambda_1$  as  $n \rightarrow \infty$ . Moreover,*

$$r^{(n)} = \frac{\|V^{(n+1)}\|}{\|V^{(n)}\|}$$

*approaches to the eigenvalue  $\lambda_1$  as  $n \rightarrow \infty$ .*

We can implement a structured version of the power method:

```

func StructuredPowerMethod({ $f_i$ } : list of tree ,  $V$  : tree,  $\epsilon$  : real)
  let  $V_{old} := V$ ,  $V_{new} := 0$ ,  $\alpha := 0$ 
  while Max(|StructuredSubtract({ $f_i$ },  $V_{new}$ ,  $V_{old}$ )|) >  $\epsilon$  do
     $V_{old} := V_{new}$ 
     $V_{new} :=$  StructuredMultiply({ $f_i$ },  $V_{old}$ )
     $\alpha := \sqrt{\text{StructuredInnerProduct}(\{f_i\}, V_{new}, V_{new})}$ 
     $V_{new} :=$  StructuredTimes({ $f_i$ },  $1/\alpha$ ,  $V_{new}$ )
  return { $\alpha$ ,  $V_{new}$ }

```

At the termination of the program,  $\alpha$  provides an estimate of the largest eigenvalue and an estimate of the associated eigenvector is stored at `newx` for any trial vector  $b$ . Variants of the power method such as the *inverse power method* to calculate the smallest eigenvalue and its eigenvector, defined by

$$AV^{(n+1)} = V^{(n)},$$

and the *shifted power method* to get the eigenvalue farthest from a specific value  $\mu$  given as the input, defined by

$$V^{(n+1)} = (A - \mu I)V^{(n)},$$

are implementable using structured operations. We can prove the convergence of these method in a similar manner as the proof of Theorem 5.5.2. The use of approximate structured operators for the power method is prohibitive since we do not have guaranteed convergence in general. For the issues on the numerical errors in the classical power method and its variants, refer to Press *et al.* [92] and Jennings and McKeown [58].

The largest eigenvalue and the smallest eigenvalue for the extrapolation method in the previous section were obtained by the structured versions of the power method and the inverse power method.

**Experiments** We summarize some experimental results to illustrate the structured version of power method. The task was to find the dominant eigenvalue and eigenvector pair for a structured stochastic transition matrix. Table 5.6 describes the result for three stochastic transition matrices corresponding to different actions in the coffee robot domain. Since it is known that the dominant eigenvalue of a stochastic transition matrix is 1, we define relative error in the eigenvalue as  $|1 - \lambda|$ , where  $\lambda$  is the eigenvalue calculated by the structured power method. For each pruning threshold  $\theta = 0, 0.0001, 0.0002, 0.0005$ , we get possibly different eigenvectors. Thus, in the last four columns of Table 5.6, we show the inner products between eigenvectors. Note that although the calculated eigenvalues with  $\theta = 0.0005$  is indeed 1, the calculated eigenvectors are far off from the true eigenvectors.

Table 5.7 shows the same experiments using ADDs on larger structured stochastic transition matrices from the FACTORY domain. The matrices are consisted of 17 binary index variables. Since there is no relationship between the number of non-terminal nodes and terminal nodes in ADDs, we report the total number of nodes in the ADDs.

Action	$\theta$	$T$	Terminals	Relative error in eigenvalue	$V \cdot V_\theta$			
					0	0.0001	0.0002	0.0005
stay	0	76.4	40	0.0258	1.0	0.993	0.981	0.282
	0.0001	45.2	12	0.0268		1.0	0.996	0.315
	0.0002	37.4	12	0.0291			1.0	0.326
	0.0005	8.7	1	0.0000				1.0
go1	0	77.3	40	0.0258	1.0	0.984	0.984	0.282
	0.0001	30.2	9	0.0276		1.0	1.0	0.300
	0.0002	26.6	7	0.0299			1.0	0.305
	0.0005	15.1	1	0.0000				1.0
go2	0	75.7	40	0.0258	1.0	0.993	0.981	0.282
	0.0001	44.9	12	0.0268		1.0	0.996	0.315
	0.0002	37.2	12	0.0291			1.0	0.326
	0.0005	8.7	1	0.0000				1.0

Table 5.6: Results of the structured power method on the Coffee Robot domain. For the transition probability matrices corresponding to actions stay, go1, and go2, we calculated the dominant eigenvalue and eigenvector for each action. For the various pruning parameter  $\theta$ , we ran the power method for 20 iterations and show the running time  $T$  in seconds and the number of terminal nodes. Although the matrices for the actions stay and go2 were different, the calculated eigenvalues and eigenvectors were the same.

Action	$T$	Nodes	Relative error in eigenvalue
polisha	0.50	11	0.0
polishb	0.48	11	0.0
glue	4.37	37	0.0

Table 5.7: Results of the structured power method on the FACTORY domain using ADDs. For the transition probability matrices corresponding to actions polisha, polishb, and glue, we calculated the dominant eigenvalue and eigenvector for each action. For the various pruning parameter  $\theta$ , we ran the power method for 20 iterations and show the running time  $T$  in seconds and the number of nodes in the ADDs.

### 5.5.3 Singular Value Decomposition

The singular value decomposition (SVD) of a matrix also reveals useful information about a matrix. SVD is applied in various tasks such as linear regression and principal component analysis (Strang [102], Golub and Van Loan [46]). Especially in AI, the Latent Semantic Analysis (LSA) of a text is basically SVD of a matrix that contains the words of the text (Berry *et al.* [8]). In practical problems, the matrix containing the words can be very large, so we may pursue a structured representation. In this section, we present a structured algorithm for performing SVD of a structured matrix. We start with the basic singular value

decomposition theorem:

**Theorem 5.5.3 (Singular Value Decomposition)** *An arbitrary  $n \times n$  matrix  $A$  can be factored as*

$$A = PDQ$$

where  $P$  is an  $n \times m$  unitary matrix,  $D$  is an  $m \times m$  diagonal matrix, and  $Q$  is an  $m \times n$  unitary matrix.

The non-zero diagonal entries in  $D$  are called *singular values* of matrix  $A$ . They are exactly the square root of the eigenvalues of the matrix  $A^T A$ . This property shows that to calculate singular value decomposition of the matrix  $A$ , we first calculate eigenvalues and eigenvectors of the symmetric matrix  $A^T A$ . We extend the power method described in the previous section to calculate all eigenvalues and eigenvectors of  $A^T A$ .

Since symmetric matrices have nonnegative eigenvalues, let's say  $A^T A$  has eigenvalues  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n \geq 0$ . We denote  $U_i$  as the eigenvector corresponding to the eigenvalue  $\lambda_i$ ,  $i = 1, \dots, n$ . A symmetric matrix has another property (Strang [102]) that there exists a set of orthonormal eigenvectors  $u_i$ 's such that

$$A^T A = \lambda_1 U_1 U_1^T + \dots + \lambda_n U_n U_n^T.$$

The power method introduced in the previous section calculates  $\lambda_1$  and  $U_1$ . Suppose that we re-apply the power method to the matrix defined by

$$A^T A - \lambda_1 U_1 U_1^T.$$

This will yield the eigenvalue  $\lambda_2$  and associated eigenvector  $U_2'$ . If  $\lambda_2 = \lambda_1$ , there is no guarantee that  $U_2'$  is perpendicular to  $U_1$ , we need to find  $U_2$  that is both the eigenvector corresponding to  $\lambda_2$  and perpendicular to  $U_1$ . This can be done by projection:

$$V_2 = U_2' - \frac{U_1^T U_2'}{U_1^T U_1} U_1,$$

$$U_2 = V_2 / \|V_2\|.$$

We can verify that  $A^T A U_2 = \lambda_2 U_2$ . In general, if we know that eigenvalues and eigenvectors  $\{(\lambda_i, U_i) | i = 1, \dots, k\}$ , we can calculate  $\lambda_{k+1}$  and  $U_{k+1}'$  by applying the power method to

$$A^T A - \lambda_1 U_1 U_1^T - \dots - \lambda_k U_k U_k^T. \quad (5.9)$$

From that, we orthonormalize  $U'_{k+1}$  with respect to  $U_1, \dots, U_k$ :

$$\begin{aligned} V_{k+1} &= U'_{k+1} - \frac{U_1^T U'_{k+1}}{U_1^T U_1} U_1 - \dots - \frac{U_k^T U'_{k+1}}{U_k^T U_k} U_k, \\ U_{k+1} &= V_{k+1} / \|V_{k+1}\|. \end{aligned} \quad (5.10)$$

This orthonormalization procedure is also known as Gram-Schmidt process (Strang [102]). By following the above steps iteratively for  $k = \{1, \dots, m\}$ , suppose that we have computed  $m$  non-zero eigenvalue and eigenvector pairs of  $A^T A$ :

$$\{(\lambda_i, U_i) | i = 1, \dots, m\}.$$

We are ready to build matrices  $P, D$  and  $Q$  that form singular value decomposition of  $A$ :

$$D = \text{diag}(\sqrt{\lambda_1}, \dots, \sqrt{\lambda_m}), \quad P = \begin{bmatrix} - & U_1 & - \\ & \vdots & \\ - & U_m & - \end{bmatrix}, \quad Q = \begin{bmatrix} | & & | \\ \frac{1}{\sqrt{\lambda_1}} AU_1 & \dots & \frac{1}{\sqrt{\lambda_n}} AU_m \\ | & & | \end{bmatrix}.$$

Instead of calculating all of  $m$  non-zero singular values, we can compute some ( $m' < m$ ) of them, building  $m' \times m'$  matrix  $D$ ,  $n \times m'$  matrix  $P$  and  $m' \times n$  matrix  $Q$ . This is one way of approximating the matrix  $A$  (Press *et al.* [92]). The following is the structured version of the singular value decomposition algorithm which calculates  $m'$  singular values:

```

func StructuredSVD({ $f_i$ } : list of tree,  $m'$  : integer,  $\epsilon$  : real)
  let eigensystem := {},  $P$  := {},  $D$  := {},  $Q$  := {}
  do  $i = 1, \dots, m'$ 
    let eigenpair := StructuredPowerMethod({ $f_i$ } : list of tree, eigensystem,
      StructuredRandomVector(),  $\epsilon$ )
    eigenpair := StructuredOrthonormalize({ $f_i$ } : list of tree, eigensystem,
      eigenpair)
    eigensystem := Append(eigensystem, eigenpair)
  do  $i = 1, \dots, m'$ 
     $D$  := Append( $D$ ,  $\sqrt{\text{eigensystem}[i][1]}$ )
     $Q$  := Append( $Q$ , eigensystem[i][2])
     $P$  := Append( $P$ , StructuredTimes(domain,  $1/D[i]$ ,
      StructuredMultiply(domain, eigensystem[i][2])))
  return { $P, D, Q$ }

```

Note that we implicitly extended StructuredPowerMethod to calculate the  $(k + 1)$ -th eigenvalue-eigenvector pair in the way shown in Equation 5.9:

```

func StructuredPowerMethod( $\{f_i\}$  : list of tree,
                            $\{\lambda_i\}$  : list of eigenvalues,  $\{U_i\}$  : list of eigenvectors,  $\epsilon$  : real)
  let  $V_{old} := V$ ,  $V_{new} := 0$ ,  $\alpha := 0$ 
  while  $\text{Max}(|\text{StructuredSubtract}(\{f_i\}, V_{new}, V_{old})|) > \epsilon$  do
     $V_{old} := V_{new}$ 
     $V_{new} := \text{StructuredMultiply}(\{f_i\}, V_{old})$ 
     $V_{new} := \text{StructuredTMultiply}(\{f_i\}, V_{new})$ 
    do  $i = 1, \dots, k$ 
      let  $V_{sub} := \text{StructuredTimes}(\lambda_i \text{StructuredInnerProduct}(\{f_i\}, U_i, V_{new}), U_i)$ 
       $V_{new} := \text{StructuredSubtract}(\{f_i\}, V_{new}, V_{sub})$ 
       $\alpha := \sqrt{\text{StructuredInnerProduct}(\{f_i\}, V_{new}, V_{new})}$ 
       $V_{new} := \text{StructuredTimes}(\{f_i\}, 1/\alpha, V_{new})$ 
  return  $\{\alpha, V_{new}\}$ 

```

StructuredOrthonormalize orthonormalizes the  $(k + 1)$ -th eigenvector with respect to those eigenvectors already calculated according to Equation 5.10:

```

func StructuredOrthonormalize( $\{f_i\}$  : list of tree,
                               $\{\lambda_i\}$  : list of eigenvalues,  $\{U_i\}$  : list of eigenvectors,
                               $\lambda_{k+1}$  : eigenvalue,  $U'_{k+1}$  : eigenvector)
  let  $V_{k+1} := U'_{k+1}$ 
  do  $i = 1, \dots, k$ 
    let  $\alpha := \text{StructuredInnerProduct}(\{f_i\}, U_i, U'_{k+1})$ 
    let  $\beta := \text{StructuredInnerProduct}(\{f_i\}, U_i, U_i)$ 
     $V_{k+1} := \text{StructuredSubtract}(\{f_i\}, V_{k+1}, \text{StructuredTimes}(\{f_i\}, \alpha/\beta, U_i))$ 
  let  $\alpha := \sqrt{\text{StructuredInnerProduct}(\{f_i\}, V_{k+1}, V_{k+1})}$ 
  let  $U_{k+1} := \text{StructuredTimes}(\{f_i\}, 1/\alpha, V_{k+1})$ 
  return  $\{\lambda_{k+1}, U_{k+1}\}$ 

```

**Experiments** Table 5.8 shows the results from structured SVD of stochastic matrices. As we observed in the structured power method, introducing a small error into the structured operators has large effects in the final results. Note that by the SVD theorem, the vectors  $U_1$  and  $U_2$  obtained from SVD should be orthogonal so that  $U_1 \cdot U_2 = 0$ , and the vectors  $1/\sqrt{\lambda_1}AU_1$  and  $1/\sqrt{\lambda_2}AU_2$  should also be orthogonal so that  $1/\sqrt{\lambda_1}AU_1 \cdot 1/\sqrt{\lambda_2}AU_2 = 0$ .

Action	$\theta$	$T$	$ U_1 $	$ U_2 $	$\sqrt{\lambda_1}$	$\sqrt{\lambda_2}$	$U_1 \cdot U_2$	$1/\sqrt{\lambda_1}AU_1 \cdot 1/\sqrt{\lambda_2}AU_2$
stay	0	79.9	20	20	1.11	1.06	0.000	0.005
	0.0001	37.0	1	11	1.04	1.04	0.000	-0.111
	0.0002	39.0	1	9	1.00	1.07	0.000	0.000
	0.0005	31.0	1	7	1.00	1.00	0.000	0.000
go1	0	76.6	20	20	1.11	1.06	0.000	-0.005
	0.0001	47.3	1	11	1.04	1.04	0.000	0.107
	0.0002	43.0	1	11	1.00	1.04	0.000	0.000
	0.0005	38.7	1	7	1.00	1.00	0.000	0.000
go2	0	79.3	20	20	1.11	1.06	0.000	-0.005
	0.0001	45.1	1	16	1.04	1.06	0.000	0.070
	0.0002	32.8	1	11	1.00	1.01	0.000	0.000
	0.0005	30.7	1	7	1.00	1.00	0.000	0.000

Table 5.8: Results of the structured SVD on the Coffee Robot domain. For the transition probability matrices corresponding to actions stay, go1, and go2, we calculated the first two dominant singular values,  $\sqrt{\lambda_1}$  and  $\sqrt{\lambda_2}$ , and their associated vectors,  $U_1$  and  $U_2$ .  $|U_1|$  and  $|U_2|$  are the numbers of terminal nodes in  $U_1$  and  $U_2$ , respectively. For varying degree of the pruning parameter  $\theta$ , we run power method for 20 iterations for the matrix  $A^T A$  and show the running time  $T$  in seconds.

Action	$T$	$ U_1 $	$ U_2 $	$\sqrt{\lambda_1}$	$\sqrt{\lambda_2}$	$U_1 \cdot U_2$	$1/\sqrt{\lambda_1}AU_1 \cdot 1/\sqrt{\lambda_2}AU_2$
polisha	10.94	21	21	5.15	4.00	0.000	0.000
polishb	10.00	21	21	5.15	4.00	0.000	0.000
glue	104.29	27	50	3.46	2.45	0.000	0.000

Table 5.9: Results of the structured SVD on the FACTORY domain. For the transition probability matrices corresponding to actions polisha, polishb, and glue, we calculated the first two dominant singular values,  $\sqrt{\lambda_1}$  and  $\sqrt{\lambda_2}$ , and their associated vectors,  $U_1$  and  $U_2$ .  $|U_1|$  and  $|U_2|$  are the numbers of nodes in  $U_1$  and  $U_2$ , respectively. We run power method for 20 iterations for the matrix  $A^T A$  and show the running time  $T$  in seconds.

Although the vectors appear to be orthogonal even with large  $\theta$ , these vectors are not really accurate vectors for singular values because of the error incurred inside the approximate power method.

Table 5.9 shows the results from structured SVD of larger stochastic matrices using ADDs. The linear algebra operators used with ADDs are exact operators. As we can see, the inner products of vectors  $U_1 \cdot U_2$  and  $1/\sqrt{\lambda_1}AU_1 \cdot 1/\sqrt{\lambda_2}AU_2$  are both 0 as desired.

## 5.6 Related Work

Press *et al.* [92] describe a wide range of numerical methods. To the extent that this body of work is concerned with exploiting structure, the emphasis is on sparse matrix methods. For a good introduction to iterative methods for solving systems of linear equations, see Kincaid and Cheney [69]. The seminal work by Wilkinson [110] gives another perspective in analyzing direct methods such as Gaussian elimination in the presence of approximate operators.

Work on MDPs dates back to the early years of operations research, and there are several excellent up-to-date references available that summarize the research carried out in operations research and adaptive control (Puterman [93], Bertsekas [9]). Much of the emphasis in these fields was on the use of iterative, dynamic programming methods for solving discrete problems (or discretized versions of continuous problems); unfortunately, since these methods required summing over the state and actions spaces, they do not scale to many of the problems encountered in practice (Littman *et al.* [75]).

While our initial motivation was to expedite the solution of MDPs, in this chapter we have attempted to push the generality of our methods as far as possible. This exercise serves to increase the class of problems that can be solved using structured methods and to increase the class of methods that can be extended and brought to bear on the problems that motivated us in the first place. One of the most important advantages of our approach over earlier, more specialized approaches is that we can now easily take advantage of a wealth of knowledge from numerical analysis and related disciplines in devising new algorithms.

The contributions of this work include data structures for representing very large matrices and vectors, a set of procedures that operate on these data structures, and a set of analytical methods that enable us to apply a wide range of numerical methods based on linear algebra directly to the solution of combinatorial optimization problems involving very large matrices and vectors.

An introduction of ADDs can be found in Bahar *et al.* [3]. The most popular package for implementing ADDs, which is also the package used in this work, is described in Somenzi [100]. Iterative algorithms for solving FMDPs with ADDs have appeared in Hoey *et al.* [53] and St-Aubin *et al.* [101]. The work in this chapter is concerned with more general problems than finding optimal policies for FMDPs.



## Chapter 6

# Conclusions

The class of FMDPs constitutes a widely used representation for stochastic planning problems. FMDPs provide us with a framework for compactly modeling problems with very large state spaces. We can often achieve an exponential savings in storage compared to the traditional approach of storing state-transition probabilities and rewards in tables that explicitly enumerate states. FMDPs take advantage of certain properties of some domains in order to achieve their economy of representation: the state space must be described in terms of a set of fluents and each fluent is independent of most of the other fluents given the small number of fluents on which it directly depends. However, it is known that such short term independence does not necessarily provide us with the computational leverage we need when we try to achieve a goal or attain high performance in the long term. The regularities that we exploit in compactly representing the domain typically do not directly lead us to an efficient technique for solving FMDPs. This thesis mainly contributes to representing additional sources of regularity and associated algorithmic techniques for exploiting these sources, useful in the sense that they provide us with computational advantage when we design algorithms for solving FMDPs.

In conjunction with Dean and Givan, the work on FA-FMDP minimization presents an aggregation method to obtain an equivalent, minimized model of an FA-FMDP without explicitly enumerating states and actions. The FA-FMDP minimization technique exploits the fact that there are often sets of states that behave the same or nearly so, and that even though the total number of actions is large the set of actions relevant to each particular set of similarly behaving states is small. This work extends the previous work by Dean and Givan on minimization of FMDPs to allow large action spaces represented by factored actions. This work also provides a way to carry out Boutilier *et al.*'s structured value iteration on

FA-FMDPs.

In conjunction with Dean, the work on non-homogeneous aggregation in FMDPs introduces an aggregation method that does not consider equivalence or similarity to the original FMDP. Instead, it exploits the fact that, given any aggregation of the states, the averaged model is often a good approximation the original FMDP. If we analyze the non-homogeneous aggregation method in the framework of  $\epsilon$ -homogeneity introduced by Dean *et al.* [34], the analysis would predict poor performance due to large  $\epsilon$ . However, the theorems and experiments in this thesis demonstrate that the non-homogeneous aggregation method can be effective for some problems.

The work on Markov task decomposition technique with Meuleau, Hauskrecht, Peshkin, Dean, Kaelbling, and Boutilier is concerned with the case in which the domain is loosely coupled. When the domain consists of sub-domains and the dynamics of these sub-domains are highly independent of other sub-domains, we can solve these sub-problems individually. We develop a heuristic method for combining the solutions to sub-problems and calculating an approximate solution to the original problem. The implementation of this technique solves, within minutes, problems with millions of states and actions.

With Dean and Meuleau, the work on model-based reinforcement contributes to extending the policy search algorithm by Williams to solve FMDPs. The work also argues that, at least in some cases, it makes sense to search in the space of policies with memory given that the smallest memoryless optimal policy may be exponential in the size of the representation of the FMDP. The research in MDPs with large state spaces including FMDPs has primarily focused on finding policies without memory. The problem of finding the optimal policy with memory for an FMDP is still a hard problem, and we developed a heuristic search algorithm based on reinforcement learning to search in the space of policies with memory.

In conjunction with Dean and Hazlehurst, the development of a linear algebra framework for very high dimension vector spaces contributes to extending a large class of iterative numerical algorithms for very large vectors and matrices. This work provides data structures and algorithms for carrying out algebraic operations when very large vectors and matrices are presented in factored representations similar to FMDPs. This work also contributes to FMDPs since we can apply a large body of techniques on error analysis in numerical algorithms to FMDP algorithms, as well as access to wide variety of algorithms to analyze the domain such as finding eigenvalues and calculating singular value decomposition. These calculations are crucial for a better understanding of the domain given in a factored representation, and the traditional method of computing these quantities is intractable without

a structured method for handling matrices and vectors with very large dimensions.

## 6.1 Future Work and Open Problems

The latest stochastic planning algorithms including the algorithms described in this thesis are starting to be able to solve large problems. However, we cannot expect an algorithm to perform spectacularly for every possible problem. We have explored various types of regularity that can provide computational leverage when present in a given problem. It would be interesting to see how well the techniques developed in this thesis apply to real-world applications, with all their quirks and real-world complications. Our experiments on the benchmark problems provide valuable lessons. However, real-world applications are likely to exhibit a combination of regularities and require additional simplifications to fit into our representational framework. Armed with the latest techniques, it will be interesting to see what real problems we can tackle now that we couldn't before.

Concerning practical applications, the world wide web is a very fascinating domain. Among recent applications of AI research, building an intelligent agent for the web has drawn a lot of attention. Particularly, formulating the behavior of the agent in a decision theoretic approach and applying conventional planning algorithms has shown some success (Etzioni and Weld [39], Boyan *et al.* [18], McCallum *et al.* [80]). It would be interesting to develop web crawling and searching algorithms using the regularity exploitation techniques described in this thesis. Note that the web is a challenging environment not only because of its large size but because of rich structure and the lack of an obvious compact representation; in order to apply the methods in this thesis, we may have to actually first learn a compact representation or make use of much larger and likely redundant representation for states and actions.

We have seen that using ADDs for FA-FMDP minimization is very effective compared to using decision trees. However, we can speculate that there exist domains in which the ADD representation size of the coarsest homogeneous partition is very large. An extension of this work would be calculating an  $\epsilon$ -homogeneous partition of an FA-FMDP. This extension has a close relationship to APRICODD by St-Aubin *et al.* [101], which is an approximate structured value iteration algorithm using ADDs. Analyzing the effect of variable reordering in ADDs and the performance of approximate minimization would be also an interesting line of research. In our biggest sample domains, we were not able to build the ADD representation for transition probabilities without reordering the variables. As St-Aubin *et al.* indicate, the quality of approximate minimization may not be sensitive to variable reordering which

can also be a huge advantage to approximate minimization using ADDs.

We also found some relationship between the discretization algorithms in continuous MDPs and the non-homogeneous partitioning method. There are some interesting ways to improve the performance of the method. First, as Munos and Moore did, we could split multiple blocks at each iteration instead of splitting just a single block. This would prevent the current method from spending too much time on splitting uninteresting blocks since the method is a greedy local search. We could also observe more robust performance. Second, we could extend the representations of blocks to be more general than simple conjuncts. We could allow ADDs to represent blocks, and impose some restrictions on how a block can be split. Note that this restriction is necessary since we may have exponentially many ways to split a block.

A shortcoming of the work in Markov task decomposition lies is that we lack a theoretical analysis of performance bounds. The work by Boyen and Koller [20, 21, 22] provides a theoretical analysis for a similar decomposition technique in the area of temporal reasoning. The most important task for bridging the Markov task decomposition and the technique by Boyen and Koller is to find an appropriate distance measure that does not increase over time steps under the decomposition. The heuristic used in the reallocation of airplanes may be improved as well.

The improvement of model-based reinforcement learning in FMDPs depends on the advances of policy search reinforcement learning algorithms. Although we showed some success, extending the algorithm so that it scales well is very important. For example, we want better performance as we introduce more complex policies. For the state-of-art discussions on this topic, refer to the dissertation by Peshkin [89].

The work on structured linear algebra also sheds lights on examining existing FMDP algorithms from the view point of classical numerical analysis. A huge body of work on acceleration in numerical analysis could be applied to FMDP algorithms. We could also obtain a structured version of interior point methods to solve linear programming problems, given that these methods use linear algebra operators that allows structural implementation. Translating classical algorithms and identifying the relationship between acceleration and the representation size would be an promising area for future research.

# Bibliography

- [1] Eric Allender and Mitsunori Ogihara. Relationships among PL, #L, and the determinant. *Theoretical Informatics and Applications*, 30:1–21, 1996.
- [2] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Structured solution methods for non-Markovian decision processes. In *Proceedings AAAI-1997*, 1997.
- [3] R. Iris Bahar, Erica Frohm, Charles Gaona, Gary Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on Computer-Aided Design*, 1993.
- [4] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings ICML-95*, 1995.
- [5] Leemon Baird and Andrew Moore. Gradient descent for general reinforcement learning. In *NIPS-99*, 1999.
- [6] Andrew Barto, Steven Bradtke, and Satinder Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.
- [7] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [8] Michael Berry, Susan Dumais, and Gavin O’Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.
- [9] Dimitri Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, 1987.
- [10] Dimitri Bertsekas and John Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

- [11] Dimitri P. Bertsekas and David A. Castañón. Adaptive aggregation for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34(6):589–598, 1989.
- [12] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.
- [13] Craig Boutilier, Ronen I. Brafman, and Christopher Geib. Prioritized goal decomposition of Markov decision processes: Toward a synthesis of classical and decision theoretic planning. In *Proceedings IJCAI-97*, 1997.
- [14] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11, 1999.
- [15] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In *Proceedings ICML-96*, 1996.
- [16] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *Proceedings IJCAI-95*, pages 1104–1111, 1995.
- [17] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in Bayesian networks. In *Proceedings UAI-96*, 1996.
- [18] Justin Boyan, Dayne Freitag, and Thorsten Joachims. A machine learning architecture for optimizing web search engines. In *Proceedings AAAI Workshop on Internet-Based Information Systems*, 1996.
- [19] Justin A. Boyan. Least-squares temporal difference learning. In *Proceedings ICML-99*, 1999.
- [20] Xavier Boyen and Daphne Koller. Approximate learning of dynamic models. In *Proceedings NIPS-98*, 1998.
- [21] Xavier Boyen and Daphne Koller. Tractable inference for complex stochastic processes. In *Proceedings UAI-98*, 1998.
- [22] Xavier Boyen and Daphne Koller. Exploiting the architecture of dynamic systems. In *Proceedings AAAI-99*, 1999.

- [23] Steven J. Bradtke and Andrew G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57, 1996.
- [24] Andrew Browder. *Mathematical Analysis: An Introduction*. Springer, 1996.
- [25] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [26] Anthony Cassandra, Leslie Kaelbling, and James Kurien. Acting under uncertainty: Discrete Bayesian models for mobile robot navigation. In *Proceedings IROS-96*, 1996.
- [27] Anthony Cassandra, Michael Littman, and Nevin Zhang. Incremental pruning: A simple, fast, exact algorithm for partially observable Markov decision processes. In *Proceedings UAI-97*, 1997.
- [28] Anthony Rocco Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Department of Computer Science, Brown University, 1998.
- [29] David A. Castañón. Approximate dynamic programming for sensor management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, volume 2, 1997.
- [30] David Chapman and Leslie Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings IJCAI-91*, 1991.
- [31] Thomas Cover and Joy Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.
- [32] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *Proceedings AAAI-97*, 1997.
- [33] Thomas Dean, Robert Givan, and Kee-Eung Kim. Solving planning problems with large state and action spaces. In *Proceedings AIPS-98*, 1998.
- [34] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *Proceedings UAI-97*, 1997.
- [35] Thomas Dean and Keiji Kanazawa. A Model for Reasoning about Persistence and Causation. *Computational Intelligence*, pages 143–150, 1989.

- [36] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings IJCAI-95*, 1995.
- [37] Thomas Dean and Michael Wellman. *Planning and Control*. Morgan Kaufmann, 1991.
- [38] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 1997.
- [39] Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *Communications of ACM*, 1994.
- [40] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [41] Jeff Forbes, Tim Huang, Keiji Kanazawa, and Stuart Russell. The BATmobile: Towards a Bayesian automated taxi. In *Proceedings IJCAI-95*, 1995.
- [42] B. L. Fox and D. M. Landi. An algorithm for identifying the ergodic subchains and transient states of a stochastic matrix. *Communications of ACM*, 2:619–621, 1968.
- [43] Robert Givan, Sonia Leach, and Thomas Dean. Bounded parameter Markov decision processes. In *Proceedings ECP-97*, 1997.
- [44] Robert Givan, Sonia Leach, and Thomas Dean. Bounded-parameter Markov decision processes. *Artificial Intelligence*, 122:71–109, 2000.
- [45] Judy Goldsmith and Robert H. Sloan. The complexity of model aggregation. In *Proceedings AIPS-2000*, 2000.
- [46] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, Second Edition*. The Johns Hopkins University Press, 1989.
- [47] Geoffrey J. Gordon. Stable function approximation in dynamic programming. Technical Report CMU-CS-103, School of Computer Science, Carnegie Mellon University, 1995.
- [48] Eric Hansen. Solving POMDPs by searching in the policy space. In *Proceedings UAI-98*, 1998.
- [49] Juris Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.



- [50] Milos Hauskrecht. *Planning and Control in Stochastic Domains with Imperfect Information*. PhD thesis, Massachusetts Institute of Technology, 1998.
- [51] Onésimo Hernández-Lerma. *Adaptive Markov Control Processes*. Springer-Verlag, 1989.
- [52] Frederick Hillier and Gerald Lieberman. *Introduction to operations research*. McGraw-Hill, 1990.
- [53] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings UAI-99*, 1999.
- [54] John Hopcroft and Jeffrey Ullman. *Introduction to Automata theory, languages, and computation*. Addison Wesley, 1979.
- [55] Michael Horsch and David Poole. An anytime algorithm for decision making under uncertainty. In *Proceedings of UAI-98*, 1998.
- [56] Ronald Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.
- [57] Laurent Hyafil and Ronald Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.
- [58] Alan Jennings and J. J. McKeown. *Matrix Computation*. John Wiley & Sons, 1992.
- [59] Mark Jerrum and Alistair Sinclair. The Markov chain Monte Carlo method: An approach to approximate counting and integration. In Dorit Hochbaum, editor, *Approximation algorithms for NP-hard problems*. PWS Publishing Company, 1997.
- [60] Leslie Pack Kaelbling, Michael Littman, and Anthony Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.
- [61] Leslie Pack Kaelbling, Michael Littman, and Andrew Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4, 1996.
- [62] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4), 1984.
- [63] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings AAAI-96*, 1996.

- [64] Michael Kearns, Yishay Mansour, and Andrew Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings IJCAI-99*, 1999.
- [65] Michael Kearns and Satinder Singh. Finite-sample convergence rates for Q-learning and indirect algorithms. In *Proceedings NIPS-98*, 1998.
- [66] Ralph Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley, 1976.
- [67] Kee-Eung Kim and Thomas Dean. Solving factored MDPs via non-homogeneous partitioning. In *Proceedings of IJCAI-2001*, 2001.
- [68] Kee-Eung Kim, Thomas Dean, and Nicolas Meuleau. Approximate solutions to factored Markov decision processes via greedy search in the space of finite state controllers. In *Proceedings AIPS-2000*, 2000.
- [69] David Kincaid and Ward Cheney. *Numerical Analysis*. Brooks/Cole Publishing Company, 1991.
- [70] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured MDPs. In *Proceedings IJCAI-99*, 1999.
- [71] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *Proceedings UAI-2000*, 2000.
- [72] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.
- [73] David Lee and Mihalis Yannakakis. On-line minimization of transition systems. In *Proceedings of 24th ACM Symposium on the Theory of Computing*, 1992.
- [74] Michael Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, 1996.
- [75] Michael Littman, Thomas Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings UAI-95*, 1995.
- [76] Michael Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9, 1998.

- [77] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proceedings AAAI-1999*, 1999.
- [78] Sridhar Mahadevan, Nicholas Marchallick, Tapas Das, and Abhijit Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proceedings ICML-97*, 1997.
- [79] Andrew McCallum. Learning to use selective attention and short-term memory in sequential tasks. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB'96)*, 1998.
- [80] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. A machine learning approach to building domain-specific search engines. In *Proceedings IJCAI-99*, 1999.
- [81] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Leslie Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings AAAI-98*, 1998.
- [82] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony Cassandra. Solving POMDPs by searching the space of finite policies. In *Proceedings UAI-99*, 1999.
- [83] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. In *Proceedings UAI-99*, 1999.
- [84] Andrew Moore and Christopher Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21, 1995.
- [85] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [86] Rémi Munos and Andrew Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *Proceedings IJCAI-99*, 1999.
- [87] Manfred Padberg. *Linear Optimization and Extensions*. Springer-Verlag, 1991.

- [88] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3), 1987.
- [89] Leonid Peshkin. *Architectures for Policy Search*. PhD thesis, Brown University, In preparation.
- [90] David Poole. A framework for decision-theoretic planning I: Combining the situation calculus, conditional plans, probability and utility. In *Proceedings of UAI-96*. 1996.
- [91] Malcolm Pradhan and Paul Dagum. Optimal Monte Carlo estimation of belief network inference. In *Proceedings UAI-96*, 1996.
- [92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 1993.
- [93] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.
- [94] J. Ross Quinlan. *C4.5 : Programs for Machine Learning*. Morgan Kaufmann, 1992.
- [95] Ross D. Shachter and Mark A. Peot. Evidential reasoning using likelihood weighting. *Artificial Intelligence*, 1989.
- [96] Satinder Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *NIPS-96*, 1996.
- [97] Satinder Singh and David Cohn. How to dynamically merge Markov decision processes. In *NIPS-98*, 1998.
- [98] Satinder Singh, Tommi Jaakkola, and Michael Jordan. Learning without state-estimation in partially observable Markovian decision processes. In *Proceedings ICML-94*, 1994.
- [99] Satinder Singh and Richard Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227–233, 1994.
- [100] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0*. Department of Electrical and Computer Engineering, University of Colorado at Boulder, 1998.
- [101] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings NIPS-2000*, 2000.

- [102] Gilbert Strang. *Linear Algebra and Its Applications*. Academic Press, 1980.
- [103] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [104] Michael A. Trick and Stanley E. Zin. A linear programming approach to solving stochastic dynamic programs. Technical report, Carnegie Mellon University, 1993.
- [105] John Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.
- [106] John Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.
- [107] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):272–292, 1992.
- [108] Chelsea White and Hany Eldeib. Markov decision processes with imprecise transition probabilities. *Operations Research*, 42(4), 1994.
- [109] David Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4), 1990.
- [110] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, 1963.
- [111] Ronald Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.
- [112] Qiang Yang, Dana Nau, and James Hendler. Merging separately generated plans with restricted interactions. *Computational Intelligence*, 8(2):648–676, 1992.
- [113] Kirk A. Yost and Alan R. Washburn. LP/POMDP marriage: Optimization with imperfect information. *Naval Research Logistics*, 47(8):607–619, 2000.