Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2004-06-14

Solving Large MDPs Quickly with Partitioned Value Iteration

David Wingate
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

Wingate, David, "Solving Large MDPs Quickly with Partitioned Value Iteration" (2004). *All Theses and Dissertations*. 47.
https://scholarsarchive.byu.edu/etd/47

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

SOLVING LARGE MDPS QUICKLY WITH

PARTITIONED VALUE ITERATION

by

David Wingate

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2004

Copyright © 2004 David Wingate

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

David Wingate

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Kevin D. Seppi, Chair
Date	Dan A. Ventura
Date	Robert P. Burton

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of David Wingate in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Kevin D. Seppi
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

SOLVING LARGE MDPS QUICKLY WITH PARTITIONED VALUE ITERATION

David Wingate

Department of Computer Science

Master of Science

Value iteration is not typically considered a viable algorithm for solving large-scale MDPs because it converges too slowly. However, its performance can be dramatically improved by eliminating redundant or useless backups, and by backing up states in the right order. We present several methods designed to help structure value dependency, and present a systematic study of companion prioritization techniques which focus computation in useful regions of the state space. In order to scale to solve ever larger problems, we evaluate all enhancements and methods in the context of parallelizability. Using the enhancements, we discover that in many instances the limiting factor of the algorithms is no longer time, but space. We thus evaluate all metrics and decisions with respect to cache performance. We generate a family of algorithms by combining several of the methods discussed, and present empirical evidence demonstrating that performance can improve by several orders of magnitude for real-world problems, while preserving accuracy and convergence guarantees.

ACKNOWLEDGMENTS

I would like to thank all of the people that have made this thesis possible. To my advisor, Kevin Seppi, for always thinking about things a little bit more abstractly than I do, for an unsinkably cheery disposition, for patiently working with me and for helping to make connections and generate ideas. To James Carroll and Chris Monson, for being good friends, for being smart enough to keep me humble, for always listening to me when I needed to talk through a thorny problem, and for continually offering insights and help. Special thanks also go to Mike Goodrich and Dan Ventura for their patient revisions and cogent suggestions.

I would like to extend special thanks to Todd Peterson. Thank you, Todd, for starting me off on a research track, for believing in me, for going up to bat for me, and for showing me a better way. Thank you most of all for teaching how to "cast my bread upon the waters." You've been an inspiration, a guide, and a good friend.

Thanks most of all to my wonderful wife, Martha, for her boundless love and constant support. I love you!

Contents

1 Introduction **1**

1.1 Motivation of the Thesis . . . 3

1.2 Summary of Contributions . . . 4

1.3 Outline of the Thesis . . . 6

2 Background **9**

2.1 Markov Decision Processes . . . 9

2.2 The Value Function and Policies . . . 11

2.3 Value Iteration . . . 12

2.4 Bellman Error and Vector/Matrix Notation . . . 13

2.5 Convergence and Optimality . . . 14

3 Related Methods **17**

4 Efficient Value Iteration **21**

4.1 Partial Sweeps . . . 22

4.2 Definitions and Notation . . . 24

5 The P-EVA Family of Algorithms **27**

5.1 Motivating and Scaling Partitions . . . 29

5.2 Prioritization Metric Semantics 31
5.3 Selecting Metrics 33
5.4 Backup Order Voting 35

6 Parallelization 39
6.1 Parallel-P-EVA 40
6.2 Assigning Partitions to Processors 43

7 On-Demand Data 47
7.1 Writing to Disk vs. Recomputing 48
7.2 Normal VI and Predictive Caches 49
7.3 Information-Frontier-Only Statistics 51

8 Algorithm Analysis 53
8.1 Analysis Notation 53
8.2 Maximum Difference and Stopping 54
8.3 Convergence 55
8.4 Rates of Convergence 56
8.5 Sufficient Subsets 57
8.6 Convergence and Optimality of Parallel-P-EVA 58

9 Experimental Setup 61
9.1 Discretizing the Space 62
9.2 Test Problems 64
9.2.1 Mountain Car 64
9.2.2 The Pendulum Family 65
9.3 Prioritization and Voting Setup 67
9.4 Parallel-P-EVA Setup 68

9.5 On-Demand Data Setup 69

10 Results **71**

10.1 Priority Metric and Voting Results 71

10.1.1 Positive Results 72

10.1.2 Negative Results 74

10.2 Parallelization Results 75

10.2.1 Positive Results 75

10.2.2 Negative Results 77

10.3 On-Demand Data Results 78

10.3.1 Positive Results 78

10.3.2 Negative Results 79

11 Conclusions and Future Research **81**

11.1 Priority Metrics and Voting 81

11.2 Parallelization 84

11.3 On-Demand Data 86

11.4 Final Conclusions 87

12 Result Graphs **89**

A Additional Multi-Media Materials **115**

List of Figures

5.1 The P-EVA algorithm with voting. 28
5.2 Performance on MCAR. 30
5.3 Different backup orders. 32
5.5 Performance on the topology in Figure 5.4. 34
5.6 A good problem for hybrid metrics. 36
5.7 How backup ordering impact performance. 37

6.1 Pseudocode for the the Parallel-P-EVA algorithm. 41
6.2 The “information frontier” of MCAR. 44
6.3 The assignment of partitions as a function of attractors. 45

9.1 The Kuhn triangulation of a 3d cube. 62
9.2 The MCAR and DAP problems. 65
9.3 The MCAR and SAP value functions 67

12.1 Results for MCAR (wallclock time). 91
12.2 Results for MCAR (number of backups). 92
12.3 Results for SAP (wallclock time). 93
12.4 Results for SAP (number of backups). 94
12.5 Results for DAP (wallclock time). 95
12.6 Results for DAP (number of backups). 96

12.7 Results for Parallel-P-EVA (time vs. attractors). 98

12.8 Results for Parallel-P-EVA (time vs. processors). 99

12.9 Efficiency of Parallel-P-EVA. 100

12.10 Performance of PSVI. 101

12.11 Efficiency of PSVI. 102

12.12 Cache performance for *H1* and *H2* on MCAR. 104

12.13 Cache performance for *H1* and *H2* on SAP. 105

12.14 Cache performance for *H1* and *H2* on DAP. 106

12.15 IFO cache performance for *H1* and *H2* on MCAR. 107

12.16 IFO cache performance for *H1* and *H2* on SAP. 108

12.17 IFO cache performance for *H1* and *H2* on DAP. 109

12.18 Performance as a function of partitions (wallclock). 111

12.19 Performance as a function of partitions (backups). 112

12.20 The MCAR control policy. 113

Chapter 1

Introduction

This thesis systematically explores the idea of minimizing the computational effort needed to compute the value function of a discrete, stationary Markov Decision Process using Value Iteration. The theme of our exploration can be stated generally as "backing up states in the correct order," and to accomplish that, several methods of differing complexity are presented and discussed which structure value dependency and prioritize computation to follow those dependencies.

The goal of this thesis is to solve large-scale, real-world MDPs quickly and accurately. Traditional algorithms, although conceptually simple and easy to implement, are incapable of producing such solutions, because of inefficiencies. These inefficiencies exist both in time (manifested by prohibitively long convergence times), and in space (manifested by excessive resource consumption and poor cache performance). This thesis seeks to remedy both problems, by quantifying inefficiencies, designing solutions to remedy the inefficiencies, and by incorporating the solutions into a holistic, unified architecture.

Central to the thesis is an evaluation of several enhancements which we make to standard Value Iteration (or VI). These enhancements are derived as potential

remedies to observed inefficiencies in the VI algorithm. To avoid redundant or useless work, we evaluate the idea of *prioritization*, which focuses computational effort in regions of the problem which are expected to be maximally productive. To manage the overhead of prioritization, both in terms of time and in terms of space, we evaluate the idea of *partitioning*, which aggregates states together and moves our algorithms up a level of abstraction. To manage the inefficiencies of partitioning, we evaluate the idea of efficient *intra-partition update ordering*, and study a low-cost way to generate good orderings.

But the solution of truly large MDPs demands more. In addition to the basic enhancements mentioned previously, this thesis strives to ensure that the enhancements facilitate effective *parallelization*, which allows the resulting algorithms to scale far beyond single-machine capabilities. This thesis also explores the idea of *on-demand data*, which quantifies the benefits of the enhancements vis-a-vis swapping parts of the problem to disk.

The existence of efficient algorithms which quickly solve large-scale reinforcement learning problems could change the RL research landscape. We anticipate that the results of this thesis will enable new classes of reinforcement learning problems to be solved by commoditizing the solution to the current generation of hardest problems. The ability to solve novel problems can in turn enable new types of applications which use efficient reinforcement learning as a functional engine. And the existence of high-quality solution engines will in turn encourage researchers to strive to solve even more challenging problems that are simply not feasible with current tools.

The results of the thesis are compelling. We demonstrate that the proposed enhancements improve performance by several orders of magnitude, while preserving convergence and optimality guarantees. We demonstrate that the enhancements function together harmoniously, because they improve performance in different ways:

algorithm designers can incorporate one, several, or all of the enhancements. We demonstrate that effective parallel versions of the algorithms are possible. And we demonstrate that one of the key contributions of the thesis, the H2 priority metric, exhibits outstanding cache efficiency.

1.1 Motivation of the Thesis

Value Iteration (or VI) is a robust and well-known method for computing the value function of an MDP, but it does not scale well for large problems. VI is pseudopolynomial in the number of states and actions (Littman, 1996). Other known solution options, such as greedy policy iteration (Mansour and Singh, 1999), and linear programming (Bertsekas, 1995), are similarly polynomial in the size of the representation of the MDP and the discount factor. Although a polynomial time algorithm is generally considered a positive thing, it still represents prohibitive complexity for extremely large problems.

The complexity of these methods can be problematic for anyone who needs to compute value functions quickly. For example, some variable resolution discretization algorithms rely on an accurate value function to guide discretization decisions: they must solve the MDP, then refine, then solve, then refine, etc. (Munos and Moore, 2002; Monson, 2003). Such methods are obviously bound by the time needed to compute the value function. Modified policy iteration (Puterman and Shin, 1978) is similarly bound, because it uses VI to compute the value for a given policy, and thus could benefit from an efficient value iterator. Some researchers may need to tune unknown problem parameters with an experimental cycle: approximate the parameter, solve the MDP, and analyze the result. A cycle lasting even several days is prohibitively long, and limits the size of problems that can be solved. There is

also the common problem in model-based reinforcement learning of finding a balance between planning and execution: ideally, an agent recomputes an optimal policy given any change in its model of the environment. Unfortunately, finding the optimal policy for a given model is non-trivial, so an agent with limited time may not be able to perform such a recomputation.

Two principal observations motivated this work. First, many backups performed by VI can be useless. VI is almost a pessimal algorithm, in the sense that it never leverages any advantage a sparse transition matrix (and/or sparse reward function) may offer. It always iterates over and updates every (s, a) pair, even if such a backup does not (or cannot) change the value function. An intuitive improvement is this: if, on the previous sweep, only a handful of states changed value, why back up the value of *every* state on the next sweep? The only useful backups will be to those states which depend upon states that changed on the previous sweep.

Second, almost all backups are naïvely ordered. For example, ordering the states in an acyclic problem such that the rows in the transition matrix are triangular (corresponding to a topological sort) yields a $O(n)$ solution; but solving the same system in an arbitrary order yields an expected $O(n^2)$ solution time. Additionally, as information backpropagates through a value function estimate, the optimal ordering may change. Dynamically approximating an optimal ordering in an efficient way is one of the central issues we examine.

1.2 Summary of Contributions

This thesis makes seven significant contributions to the field of computer science. This section briefly outlines them, but one of the most significant contributions is the fact that the contributions are all complimentary, and function harmoniously in

a unified architecture.

First, this thesis studies prioritization metrics systematically, comparing and contrasting them to each other. The idea of efficient computation applied to VI is not new, but it has not received a dedicated treatment; most other papers have only presented a metric in isolation, and labeled it as a heuristic to enhance performance. Prioritized Sweeping (Moore and Atkeson, 1993), for instance, uses Bellman error as a priority metric, but we demonstrate that another equally simple metric (the "*H2*" metric) often performs better when used appropriately.

Second, the thesis introduces the idea of *approximate prioritization*, which is accomplished principally through partitioning. This has the benefit of managing the complexity introduced with the priority metrics, which is an issue other researchers have not addressed. Partitioning not only reduces priority queue management overhead, but it also reduces the size of model inverses which are necessary, and naturally facilitates an efficient parallel implementation.

Third, the thesis begins a study of *hybrid* prioritization metrics, which are composed of several atomic metrics. The hybrid metrics (which occur implicitly in the algorithms we present) often yield backup orderings which are superior to the orderings yielded by strict use of a single priority metric.

Fourth, the thesis introduces the idea of "voting," which is a low-cost method to determine an intra-partition update order. The thesis empirically demonstrates that voting can improve algorithm performance tremendously, even with naive algorithms.

Fifth, the thesis introduces the *H2* priority metric. In addition to yielding better backup orderings, and therefore better wallclock performance for our algorithms, we show empirically that the *H2* metric exhibits outstanding cache efficiency. Although the principal focus of the thesis is on temporal efficiency, this insight into spatial efficiency solidifies *H2* as the metric of choice for solving very large MDPs.

Sixth, and somewhat in contrast to most asynchronous VI proofs of convergence (such as Bertsekas, 1982, 1983; Gullapalli and Barto, 1994), the thesis contributes proof that not every state needs to be backed up during each sweep in order to maintain contraction and convergence properties. In fact, some states may never need to be backed up, which can have significant impacts on the amount of effort needed to solve a problem. An important result is the fact that no additional code is necessary to claim this benefit: it is an emergent property of the algorithms.

Seventh, we present an effective parallel version of the algorithm, which demonstrates that all of the improvements and enhancements proposed (such as partitioning, voting, and the priority metrics) work equally well in parallel and serial scenarios. This implies that algorithm designers do not need to trade any of our enhancements for parallelizability, and can therefore fully leverage supercomputing power.

Finally, although not a tangible contribution in the same way as the others, this thesis opens significant new research directions as a result of the insights gained, questions posed, and results obtained. It is anticipated that many of the developments in this thesis will serve as solid stepping stones for even more significant contributions.

1.3 Outline of the Thesis

Chapter 2 ("Background") reviews introductory material on Markov Decision Processes, value functions, policies, convergence and contraction. We also review the normal value iteration algorithm, which is the basic algorithm we enhance throughout the thesis. Chapter 3 ("Related Work") briefly points out related work. We note here that very little work which is directly comparable to our algorithms has been produced, mostly because of differences in the assumption of model availability. Chapter 4 ("Efficient Value Iteration") discusses efficient value iteration and presents

the basic algorithmic enhancements we make to normal value iteration: prioritization, partitioning, and intra-partition backup ordering. This chapter also introduces the key concepts and definitions, and presents the central questions of the thesis.

Chapter 5 ("The P-EVA Family of Algorithms") describes the P-EVA family of algorithms in detail, discussing the motivation of partitioning, the semantics of the different priority metrics, and how "voting" computes a low-cost intra-partition backup ordering. Chapter 6 ("Parallelization") presents a parallel version of P-EVA, named Parallel-P-EVA. This chapter discusses implementation issues, presents some theoretical results, and contributes insights into appropriate domain decompositions. Chapter 7 ("On-Demand Data") explores the cache behavior of the P-EVA algorithms, focusing on the behavior generated by the use of the different priority metrics. This idea is related to spatial efficiency, and is designed to facilitate the solution to truly large problems. Chapter 8 ("Algorithm Analysis") presents theoretical results related to convergence, optimality, rates of convergence, and sufficient subsets.

Chapter 9 ("Experimental Setup") discusses the experimental domain of the thesis. Here we discuss the test suite, the reason that we selected minimum-time optimal control problems, the methods that we use to discretize continuous problems, and the hardware that was used in various experiments. Chapter 10 ("Results") presents the results of all of the experiments, and Chapter 11 ("Conclusions and Future Research") presents conclusions and future research possibilities.

The charts and graphs collected during the course of the thesis are collected in Chapter 12 ("Result Graphs"). Additionally, an on-line appendix is available, which is described at the end of the thesis.

Chapter 2

Background

This chapter presents introductory material on Markov Decision Processes (Section 2.1), the value function of an MDP (Section 2.2), policies (Section 2.2), the basic VI algorithm (Section 2.3), Bellman error and alternative vector/matrix notation (Section 2.4), and common convergence and optimality definitions (Section 2.5). In addition, this chapter introduces most of the notation that will be used throughout the remainder of the thesis. Some of the notation that is specific to our algorithms, and not general to all Markov Decision Processes, is reserved for Section 4.2. We refer the interested reader to Puterman (1994) for a more detailed and rigorous treatment of the foundations of MDPs, their properties, and solution algorithms.

2.1 Markov Decision Processes

A *Markov Decision Process* is a tuple $M = (S, A, Pr, \gamma, R)$ describing a stochastic, evolutionary process. Here $S \subset \mathcal{N}$ is a finite set of states, $A \subset \mathcal{N}$ is a finite set of actions, and $Pr(s'|s, a)$ ($Pr : S \times S \times A \rightarrow \Re$) is a probability mass function describing the probability of transitioning to state s' given that the system is in state s and executes

action a. $R(s, a)$ $(R : S \times A \to \Re)$ is a function describing the reward received by an agent which executes action a in state s. $\gamma \in [0, 1)$ is known as the *discount factor*.

At any given time, the system is in some state $s \in S$. At each decision epoch (for our purposes, this means each timestep), the system can select an arbitrary action $a \in A$ from the set of available actions. Upon executing the action, the system transitions to a random successor state which is selected according to the probability distribution Pr.

Each state and action has a reward associated with it. Since there are several choices of actions at each state, the goal is to determine which action yields the highest expected reward. Some state-action pairs may yield an immediate reward of zero (or may yield negative rewards), but may transition the system into a state where a large positive reward is possible. We therefore want the system to be able to look an arbitrary number of steps into the future when determining how to maximize its reward, but we wish the system to value immediate rewards (rewards that are fewer steps away) more than long-term rewards. To bias the system towards closer rewards, we discount future rewards by a factor of γ for each step it takes to reach them (recall that $\gamma < 1$). In order to determine which actions produce the highest expected discounted reward, we compute the *value function* of the MDP, which is described below. This problem formulation is known as a *discounted infinite-horizon* optimality criteria. Other optimality criteria, such as Blackwell optimality (Blackwell, 1962), finite-horizon reward, total reward, or average reward criteria (Mahadevan, 1996), are also possible. For a thorough survey of optimality criteria, we refer the reader to Littman (1996).

The "Markov" property of the process indicates that the transition probabilities depend only upon the current state, and not upon any previous states . In other words, it doesn't matter *how* the system arrived in the state that it is in – it only

matters that it is *in* that state. Such systems are sometimes known as *memoryless* systems (Kakade, 2003). The "Decision" property indicates that there is a choice of actions in some states. If there were no actions, we would simply have a *Markov Process*.

2.2 The Value Function and Policies

The value function of an MDP M is mathematically defined as the solution to a set of Hamilton-Jacobi-Bellman (HJB) equations. These equations mathematically describe the preceding section. The agent must maximize the expected discounted reward by selecting the best action possible, as shown in Equation 2.1.

$$V(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s'} Pr(s'|s,a)V(s') \right\} \tag{2.1}$$

Clearly, the solution to this set of equations depends directly on the choice of action in each state. This choice is known as a *policy*, which is formally defined as a mapping ($\pi : S \rightarrow A$), and indicates which action an agent will execute in each state. We say that a given policy π *induces* a value function on the MDP, and will only talk about solving this system of equations with respect to a given policy. Under a policy π, Equation 2.1 becomes Equation 2.2.

$$V(s) = R(s,\pi(s)) + \gamma \sum_{s'} Pr(s'|s,\pi(s))V(s') \tag{2.2}$$

If $n = |S|$ is the number of states in the MDP, then Equation 2.1 is a system of n equations and n unknowns. This system may be solved by Gaussian elimination, matrix inversion, Gauss-Jordan elimination, or any other linear systems solution method. However, all of these methods have a time complexity that is $O(n^3)$, which means that solving the value function equations directly is prohibitively expensive. Thus, most algorithms approximate the solution, as discussed in the next section.

Typically, we are interested in computing the *optimal* value function. The optimal value function (denoted V^*) maximizes the expected discounted reward in every state, and is induced by a corresponding *optimal policy* (denoted π^*).

2.3 Value Iteration

As noted, computing an exact solution to the system of equations in Equation 2.1 is prohibitively expensive. Instead, most algorithms opt to approximate the correct value function to within some accuracy ϵ. To accomplish this, an algorithm will generally begin with an arbitrary estimate of the value function, V_0, and execute some sort of algorithm to move the estimate closer to the optimal value function. We will subscript progressive value function estimates with a time index (as in V_t).

Value Iteration (Bellman, 1957) is an algorithm which successively approximates the value function, starting from an arbitrary initial estimate. The Value Iteration (or VI) algorithm simply changes the equality operator in Equation 2.1 to an assignment operator:

$$V_t(s) \leftarrow \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V_{t-1}(s') \right\} \tag{2.3}$$

In VI, the agent iterates (or *sweeps*) over every state, and updates the value of that state according to Equation 2.3. VI can therefore be viewed as generating all one-step optimal policies, then two-step optimal policies, etc., and is generally considered a form of dynamic programming. In the limit, it is guaranteed to generate a policy which is optimal with respect to an infinite horizon. We say that when an algorithm updates the value function estimate for a state s, it has *backed up* the value of state s.

To facilitate compact notation, most researchers consider an algorithm such as VI an atomic operator which operates on an entire value function estimate. The VI

operator is usually written T, and using it, Equation 2.3 may be expressed as:

$$V_t = TV_{t-1} \tag{2.4}$$

The use of such notation, combined with the strict time subscripting, justifies the idea that all of the states are being simultaneously backed up in parallel. This is why operators which do not back up all of the states in a given timestep are known as *asynchronous* operators (Sutton and Barto, 1998). We also note that this form of parallelism is more of a theoretical construct, and should not be confused with actual massively parallel implementations which run on supercomputers or clusters (as discussed in Chapter 6).

2.4 Bellman Error and Vector/Matrix Notation

When an algorithm such as VI backs up a state, the value function estimate for that state will sometimes change. The amount of change that will occur, assuming that a backup were executed, is known as the *Bellman error* of the state:

$$B_t(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V_t(s') \right\} - V_t(s) \tag{2.5}$$

Note that the Bellman error should not be considered a one-step temporal difference (Sutton, 1988): the Bellman error represents the amount of *potential* change to the value function, assuming that a certain state was backed up, as opposed to the *actual* difference between two value function estimates separated by one timestep. This subtle difference will be significant in the context of any algorithm that does not back up every state at each timestep; the algorithms we will introduce are of this type. Peng and Williams (1993) called the Bellman error the *prediction difference*. We will let

$$M_t = \|B_t(s)\|$$

be the largest potential update in the system. This quantity is commonly called the *Bellman error magnitude* (Williams and Baird, 1993).

Equations 2.3 and 2.5 may be rewritten using vector and matrix notation. Following Puterman (1994), let $d \in D$ be a deterministic Markovian *decision rule*, defined as a mapping $d : S \rightarrow A$. Use of decision rules allows us to simplify equations 2.6 and 2.7 by extracting the max operator; we stress, however, that the maximum decision rule can be determined component-wise (if not, we would have to enumerate all possible decision rules). Since reward functions and transition probabilities depend upon the action selected from each state, we will subscript them with a decision rule. We note that a decision rule is simply a slightly more general version of a policy.

Using vector and matrix notation, VI and the Bellman error function can be expressed as:

$$V_t = \max_{d \in D} \{R_d + \gamma \mathbf{P_d} V_{t-1}\} \tag{2.6}$$

$$B_t = \max_{d \in D} \{R_d + \gamma \mathbf{P_d} V_t\} - V_t \tag{2.7}$$

Using the Bellman error function, the VI operator may be equivalently expressed as

$$V_t = TV_{t-1} = V_{t-1} + B_{t-1} \tag{2.8}$$

2.5 Convergence and Optimality

There are several questions which are important when evaluating the utility of a solution algorithm such as VI. The two principal ones deal with convergence and optimality. Since VI successively approximates the solution to a system of equations, the first question is, “will the algorithm eventually converge to a single answer?” The second is, “which answer will the system converge to?” and “Is it the optimal an-

swer?" We will briefly show that VI does in fact converge, and that when it converges, it converges to the optimal policy and therefore to the optimal value function.

A convenient proof of convergence is carried out using max-norm analysis. First, we define what we mean by convergence. For our purposes, a value function estimate has converged if the system has reached a *fixed point*:

$$V_t = V_{t-1} = TV_{t-1}$$

A value function estimate V is a fixed point any time application of the operator T does not change the value function estimate.

We prove convergence for any operator that is a contraction mapping, meaning that it satisifies the following equation:

$$\|Tv - Tu\| \leq \gamma \|v - u\|$$

(where $\|\cdot\|$ is the max norm of a vector field, and v and u are value function estimates). Intuitively, this definition states that the largest difference between two value function estimates shrinks as the T operator is repeatedly applied.

Because the largest difference always shrinks between two value function estimates, it is easy to show that contraction mappings always tend towards a fixed point. We refer the interested reader to Puterman (1994) for details of the proof. It is easy to show that for HJB systems of equations, there is only one fixed point in the space of all possible value function estimates, and that this fixed point corresponds exactly to the optimal value function estimate V^*:

$$V_t = TV_{t+1} \Rightarrow V_t = V^* \quad (2.9)$$

Next, we show that VI is a contraction mapping in max-norm. Recall that there are two important properties of max-norm operators: first, that the max-norm of a

scalar is just that scalar, and that the max-norm of a matrix is the largest row-sum in the matrix. Since all matrices involved here are probability matrices, their max-norm is simply equal to one. This yields the following proof:

Proof.

$$
\begin{aligned}
\|TV_t - TV_{t+1}\| &\leq \gamma\|V_t - V_{t+1}\| \\
\|V_{t+1} - V_{t+2}\| &\leq \gamma\|V_t - V_{t+1}\| \\
\|R_d + \gamma\mathbf{P_d}V_t - R_d - \gamma\mathbf{P_d}V_{t+1}\| &\leq \gamma\|V_t - V_{t+1}\| \\
\|\gamma\mathbf{P_d}\|\|V_t - V_{t+1}\| &\leq \gamma\|V_t - V_{t+1}\| \\
\gamma\|V_t - V_{t+1}\| &\leq \gamma\|V_t - V_{t+1}\|
\end{aligned}
$$

□

The idea of convergence in norm will be essential to the analysis of our algorithms in Chapter 8.

Tangentially, we note that since this is a convergent system, the absolute value of all eigenvalues $\lambda_1 ... \lambda_{n-1}$ of the matrix defined by the value function are strictly less than 1, with the single exception of the dominant eigenvalue λ_0, which is strictly equal to one. The solution vector V corresponds to the dominant eigenvector of the matrix defined by the system.

Chapter 3

Related Methods

This work is about the efficient backpropagation of correct value function estimates. Other researchers who have investigated similar issues of efficiency have produced results that are tangible and compelling, but their algorithms are not directly comparable to ours because they have been developed in the context of on-line, model-free learning. Our algorithms, in contrast, explicitly assume the availability of a complete model.

The difference in these domains is significant and shifts the emphasis of the work. Model-free algorithms do not have the luxury of executing backups to states they have not visited; model-based algorithms, in contrast, can execute backups to any state, in any order. This fact frees us to examine different types of questions. For example, most model-free algorithms must content themselves with backpropagating information along experience traces, but there is no reason to suppose that an experience trace (played in any order) represents an *optimal* sequence of backups. It is a surrogate for what is truly desired, which is the ability to digest the consequences of corrected value function estimates as quickly and thoroughly as possible, throughout the entire problem. Thus, instead of examining questions related to maximizing the

utility of experience traces, this work examines questions related to finding globally optimal backup sequences.

There are three primary classes of methods that researchers have used to accelerate the backpropagation of correct value information. Algorithmically, these methods form a poor basis for comparison, but conceptually, they illustrate several important points.

First is the class of *trace propagation* methods, such as TD(λ) (Sutton, 1988), Q(λ) (Peng and Williams, 1994), SARSA(λ) (Rummery and Niranjan, 1994), Fast Q(λ) (Reynolds, 2002) and Experience Stack Replay (Reynolds, 2002). These methods store a record of past experiences. As value function estimates are corrected, the changes are propagated backwards along the experience trace. Relative to VI, these methods derive enhanced performance partly from backing up states in a principled order (that is, backwards) and by only backing up a subset of all states. The ideas of principled ordering and partial sweeps will be central in this work.

Second, there are *forced generalization* methods, such as Eligibility Traces (Singh and Sutton, 1996), PQ-learning (Zhu and Levinson, 2002) and Propagation-TD (Preux, 2002). These methods attempt to compute the value for a state based on information that was not directly associated with an experience trace. States selected for backup may have been part of a previous experience trace, or may have a geometrical or geodesic relationship to states along the actual trace (this happens implicitly with function approximators, but is forced to happen explicitly in these tabular methods).

Third, there are *prioritized computation* methods, such as Prioritized Sweeping (Moore and Atkeson, 1993) and Queue-DYNA (Peng and Williams, 1993). These methods order the backups in a principled way by constructing priority queues based on Bellman error. The idea of prioritizing backups is also central to our thesis,

but these methods raise many questions that merit further study. It is from these questions that our work springs.

Other researchers have considered extensions to the three basic classes previously enumerated, but the extensions do not match our domain of interest. For example, Andre et al. (1998) proposed a continuous extension to Prioritized Sweeping, and Zhang and Zhang (2001) discuss a method for accelerating the convergence of VI in POMDPs. Policy iteration has traditionally performed much better than VI, but is problematic because it may require an exponential number of sweeps for certain families of MDPs (Littman, 1996). It is also well known that the dual of any MDP can be solved by linear programming. However, Littman et al. (1995) point out that "existing algorithms for solving LPs with provable polynomial-time performance are impractical for most MDPs. Practical algorithms for solving LPs based on the simplex method appear prone to the same sort of worst-case behavior as policy iteration and value iteration." Gordon (1999) provides a thorough survey of other MDP solution techniques, such as state aggregation, interpolated VI, approximate policy iteration, policies without values, etc.

Chapter 4

Efficient Value Iteration

There are three principal methods which we use to improve the efficiency of VI, each of which is discussed in detail in Chapter 5, and which are briefly outlined here. Following the outline, we will discuss some issues related to partial sweeps, and then introduce the definitions and notation specific to our algorithms.

The first method we use to improve efficiency is the prioritization of backups. Essentially, instead of naïvely sweeping over the entire problem, we wish to work our way backwards through the problem. We correct the value function estimate for a state s by backing it up, and then correct the value function estimate for all states which depend upon s. We use Bellman error to characterize how useful any given backup is, and then construct different metrics based on the Bellman error as the priority in a priority queue. In this work, we explore two different prioritization metrics atomically, as well a hybrid metric (each having significantly different semantics), and discuss performance characteristics of all three.

Secondly, we employ the idea of a *partition*. Partitions, which are just sets of states, improve efficiency for three reasons. First, they enable approximate prioritization. Demanding perfect prioritization of every state adds prohibitive overhead,

because once the value for a state has been corrected, each dependent state much be extracted, reprioritized, and reinserted into the queue. For most priority queue algorithms, this results in several $O(\log n)$ operations. If a partition contains mutually dependent states, then backing up states in a high-priority partition is guaranteed to back up the state responsible for the high priority, and automatically backs up other high-priority states, without having to manage the priority queue. Second, partitions enable smaller inverse models to be used. Prioritization at a state level of granularity means that a full model inverse must be stored, because every time a state changes value, all of its dependents must be reprioritized. By moving to a partition level of abstraction, the only dependents that need to be reprioritized (and whose transitions need to be stored) are the dependents in other partitions. Partitions should therefore be constructed to minimize the number of these "cross-partition transitions." This is why our algorithms are structured around priorities between partitions. Third, partitions make *hybrid* prioritization metrics possible: the partition can be prioritized "globally" using one metric, but the states within the partition can be prioritized locally using a different metric.

Since we are interested in performing backups in a correct order, the third and final method we use is the idea of "voting" on an intra-partition backup order. Code and data organization stipulate that the states in a partition must be backed up in some order. Voting is a low-cost, high-yield method which allows states to direct their own backup order.

4.1 Partial Sweeps

None of our algorithms actually have the concept of a full sweep in the same way that VI or policy iteration does. To avoid useless backups, our algorithms implicitly

discard full sweeps in favor of structured, ordered partial sweeps. This deserves some explanation in terms of convergence and complexity.

There are two distinct issues forming the basis for the complexity of VI. The first is the most commonly analyzed: how many sweeps are required for ϵ-convergence? This question has already been answered in detail by Littman (1996). The second, which has been analyzed less frequently, is this: what suffices as a sweep?

Convergence proofs for algorithms which compute the value function of an MDP require an operator T that operates on a value function estimate V_t. This operator must be a contraction mapping, meaning that it must guarantee that:

$$\|Tv - Tu\| \leq \gamma \|v - u\|$$

(where $\|\cdot\|$ is the max norm of the function). In the traditional VI algorithm, the operator is a full sweep over the value function, backing up each state in the usual way (see Equation 2.3). This guarantees that the value function estimate V_t will converge by a factor of at least γ to the optimal value function V^*.

However, there is no stipulation that T operate on every state. This begs the question: what is the minimal number of backups required to ensure suitable contraction? Obviously, the answer depends on the problem. One can easily find examples of sparse matrices where *one* backup is sufficient to force a contraction, but it is also possible to show examples where $|S|$ backups are required (the traditional worst-case bound describing a fully-connected graph). Of course, hardly any large, real-world problems are actually fully-connected graphs.

We can not claim that the algorithms we have developed fundamentally change the complexity (expected or otherwise) of VI. Nor can we claim that they improve performance for all problems; in fact, for some problems, they perform worse due to overhead. But we *do* claim, and demonstrate, that the algorithms remove tremendous

inefficiencies, and function extremely well as general-purpose solutions for practical, large-scale problems.

4.2 Definitions and Notation

This section introduces the specific definitions and notations that we will use to formally describe our algorithms. Before we begin, we note that most of the notation dealing with partitions has been structured around sets. This was done to emphasize the fact that partitions can contain arbitrary states, which may or may not have any relation to each other.

Our algorithms build different prioritization metrics upon the Bellman error function. The first metric we will analyze, *H1*, is equal to the Bellman error itself:

$$H1_t(s) = B_t(s)$$

The second metric is:

$$H2_t(s) = \begin{cases} B_t(s) + V_t(s) & B_t(s) > \epsilon \\ 0 & \text{otherwise} \end{cases}$$

The semantics of both of these metrics will be discussed more fully in Section 5.2. When it is not important which prioritization metric is used, we will use $H_t(s)$ to refer to a generic one.

Let P be a set of partitions which denotes a particular partitioning of the state space, and let $N_p = |P|$ be the number of partitions. Let each $p \in P$ be a set of states. Let $P_s : S \to P$ be the mapping of states to the partitions that contain them. Each P must tessellate the set S by obeying two properties:

$$\bigcup_{p \in P} = S \quad \texttt{and} \quad \forall_{p_1, p_2} p_1 \cap p_2 = \emptyset$$

The *state dependents of a state* is the set of all states who have some probability of transitioning to s, and therefore whose value depend on the value of s. We define it as

$$SDS(s) = \{s' : \exists a Pr(s'|s,a) \neq 0\}$$

The *state dependents of a partition* is the set of all states whose value depends on some state in the partition p. We define it as

$$SDP(p) = \bigcup_{s \in p} SDS(s)$$

The *partition dependents of a state* is the set of partitions which contain a state whose value depends on s. We define it as

$$PDS(s) = \bigcup_{s' \in SDS(s)} P_s(s')$$

The *partition dependents of a partition* is the set of all partitions that contain at least one state that depends on the value of at least one state in p. We define it as

$$PDP(p) = \bigcup_{s \in p} PDS(s)$$

Define the priority between two partitions as

$$HPP_t(p,p') = \max_{s \in p \cap SDP(p')} H_t(s)$$

Note that in general, $HPP_t(p,p') \neq HPP_t(p',p)$. Define the priority of a partition as

$$HP_t(p) = \max_{p'} HPP_t(p,p')$$

To simplify the following discussions, all of the MDPs considered are *positive bounded* (all rewards are positive and finite). Creating a positive bounded MDP can be accomplished by adding a constant C to the reward function; since the value function estimate will be initialized to 0, this ensures that $\forall_t V_t \leq V^*$. This does not

change the resulting policy, and, as Zhang et al. (1999) point out, “the value function of the original [MDP] equals that of the transformed [MDP] minus $C/(1-\gamma)$, where C is the constant added.” This stipulation also simplifies some of the bounds provided in Chapter 8.

Chapter 5

The P-EVA Family of Algorithms

This section introduces the P-EVA family of algorithms. We begin with the the P-EVA ("Partitioned Efficient VAlue iterator") algorithm of Wingate and Seppi (2003) as our base, and generate a family of algorithms by generalizing it according to the techniques we discuss in this thesis. To refer to the family in general, we will use the term "P-EVA," and we will refer to a specific variant within the family by suffixing "P-EVA" with a mnemonic string.

The next subsections discuss partitions, priority metric semantics, when to select which priority metrics, when to select hybrid metrics, and how to compute intra-partition backup orders with voting. Figure 5.1 shows the complete P-EVA algorithm. We note that the notion of a timestep is different between P-EVA and value iteration because in P-EVA, only one state is backed up at every timestep, but in normal VI, *every* state is backed up.

Following is a sketch of the core algorithm, which is common to the entire family:

Initialization: Let $V_0 = 0$. Partitions are processed according to a priority metric, which is defined as the maximum priority of any state within the partition. This implies that $HP_0(p) = \max_{a \in A, s \in p} R(s, a)$.

Initialize by setting $V_0(s) = 0$, $H_0(s) = \max_{a \in A} R(s,a)$, $HP_0(p) = \max_{s \in p, a \in A} R(s,a)$.
Select an initial partition $p = \arg\max_{p'} HP_0(p')$.

Repeat

1. Value iterate over p in the order dictated by voting
   - $\Delta_{max} = 0$
   - Repeat $\forall s \in p$
     - $V_{t+1}(s) \leftarrow \max_{a \in A} \{R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V_t(s')\}$
     - $\Delta_{max} = \max(\Delta_{max}, V_{t+1}(s) - V_t(s))$
     - $t \leftarrow t + 1$
   - until $\Delta_{max} < \epsilon$
2. $HP_t(p) \leftarrow \Delta_{max}$
3. Update partition priority for all dependent partitions
   - For each $p' \in PDP(p)$:
     - $HPP_t(p', p) \leftarrow 0$
     - $h_{max} \leftarrow 0$
     - For each $s' \in (p' \cup SPD(p))$:
       * Update $H_t(s')$
       * $h_{max} = \max(h_{max}, H_t(s'))$
     - $HPP_t(p', p) \leftarrow h_{max}$
     - $HP_t(p') \leftarrow \max_p HPP_t(p', p)$
4. Select the next partition $p = \arg\max_{p'} HP_t(p')$

until $M_t/(1-\gamma) < \epsilon$, or until some other stopping criteria is reached.

Figure 5.1: The P-EVA algorithm with voting.

Repeat: Select the highest-priority partition p from the queue ($p = \arg\max_{p'} HP_t(p')$). Value iterate normally over all $s{\in}p$ in the order dictated by voting. Update the value function for each s in the usual way. Repeat until ϵ-convergence. Recompute the priorities of all states that depended on any state within the partition (this is $SPD(p)$), and recompute the partition priority of each dependent partition (this is $PDP(p)$).

Until: Terminate the algorithm when some sort of stopping criterion is reached. This will be typically be an ϵ-convergence test such as that given in Section 8.2.

5.1 Motivating and Scaling Partitions

The overhead of managing the priority queue is high. Each dependent must be extracted, reprioritized, and reinserted into the queue, resulting in several $O(\log n)$ operations per backup. Figure 5.2 illustrates this overhead empirically. On one problem, although P-EVA with one state per partition backs up the value function far fewer times than normal VI, it takes far longer to solve the problem.

Two observations direct our solution. First, we can accept some back ups that do not occur in strict priority order. Second, any single state (typically) depends on multiple other states; it would be ideal to postpone the reprioritization of a state until multiple dependencies have been backed up. A good principle is to group states together into sets, and to work on the sets, instead of individual states. This accomplishes both goals because it efficiently approximates the backup order induced by the priority metric, and it tends to ensure that multiple dependencies are resolved before moving on.

The specific partitioning used navigates the trade-off between useless backups (there might be states in the partition that did not need to be processed) and priority queue overhead (it is faster to update them anyway, because it takes too long to figure

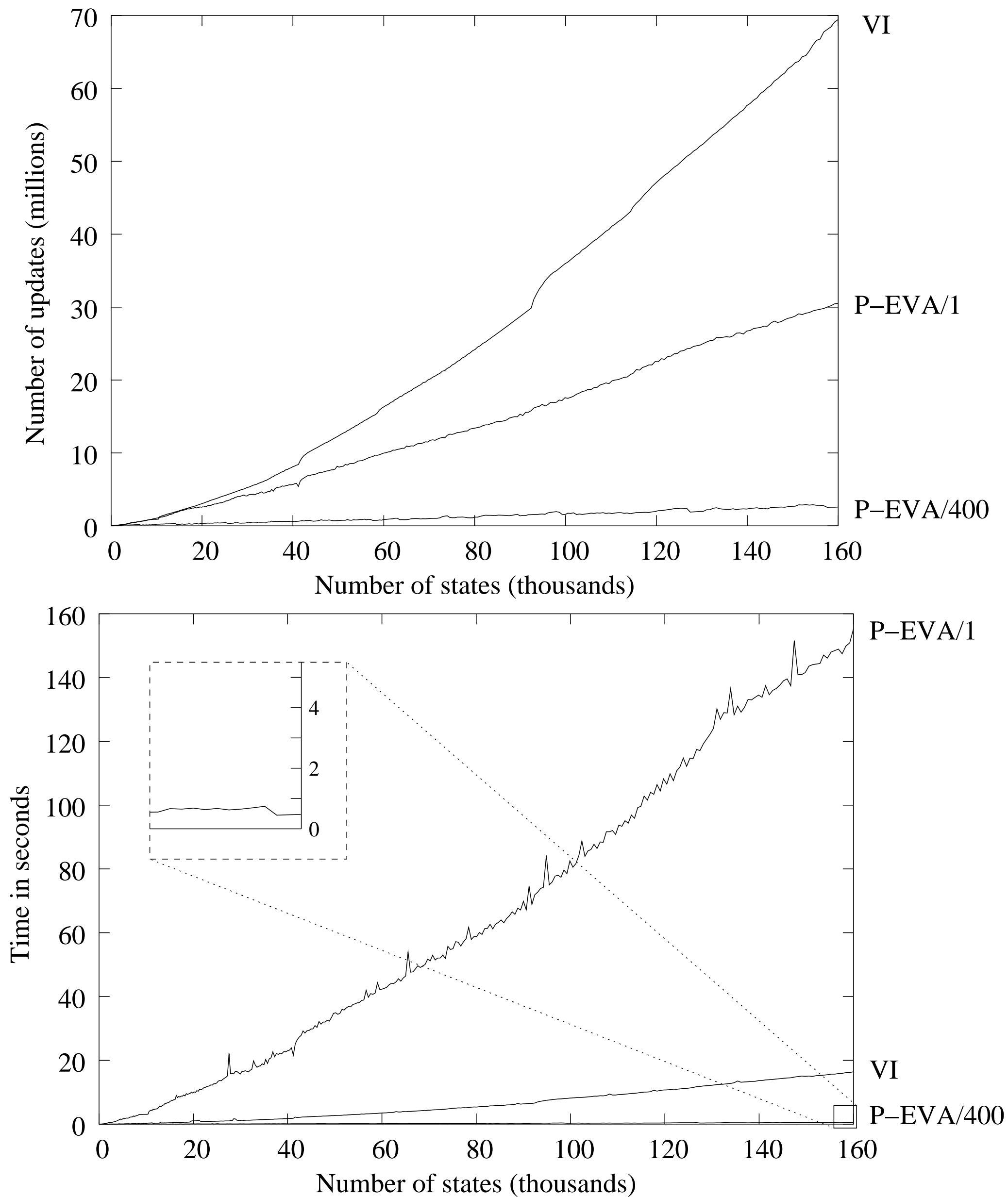

Figure 5.2: Performance on MCAR as a function of problem size. Although P-EVA with one state per partition (P-EVA/1) performs half as many backups as VI, it takes far longer to complete. Priority queue overhead accounts for most of this discrepancy. Adding more states to the partition greatly alleviates the problem: the bottom graph shows that P-EVA with 400 states per partition (P-EVA/400) requires only 0.5 seconds to solve the problem.

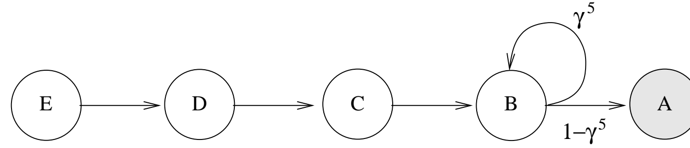

Figure 5.3: An example for demonstrating the different backup orders of *H1* and *H2*.

out which ones are useless). Thus, running P-EVA with a single partition containing all states is equivalent to normal VI, while running it with a single state per partition yields updates in strict priority order (as we shall note later, this is not *quite* true in the single case that a state has a loop to itself).

As shown in Figure 5.2, adding more states to the partitions dramatically improves performance. Both P-EVA/1 and P-EVA/400 perform fewer backups to the value function than normal VI, but because P-EVA/400 eliminates priority queue overhead, the time needed for it to reach a solution drops by three orders of magnitude. The astute reader will note that, counter-intuitively, P-EVA/400 performed fewer backups than P-EVA/1. This behavior is explored in Chapter 5.3.

The partitioning method used in our experiments is described in the Chapter 9.

5.2 Prioritization Metric Semantics

This thesis examines two different prioritization metrics, each of which exhibits very different behavior. We encourage the reader to refer to the on-line appendix to build intuition regarding the different priority metrics and the way that they propagate information.

The *H1* prioritization metric is the most obvious metric, and has been studied before (although not in contrast to other metrics). Using it, P-EVA can be thought of as a greedy reduction in the error of the value function estimate. This has the

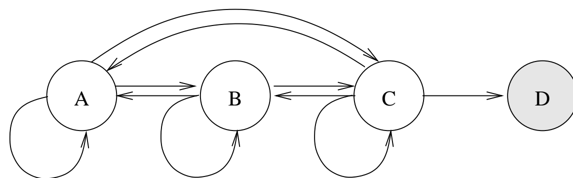

Figure 5.4: An example topology for which P-EVA (with either metric) yields a highly suboptimal backup order, but for which value iteration yields an almost optimal backup order. State D is an absorbing reward state. Only one action is available at each state. Transitions to other states all have equal probability.

tendency to change the estimate quickly throughout the state space, but it also tends to leave large regions only partially converged, and therefore does not necessarily propagate correct *policy* information quickly.

The *H2* metric has a very different effect on computation order. The intuition is this: if there is a value that is more than ϵ away from its optimal value, the value will eventually have to be corrected. Since large values (generated from large rewards, or small loops) have greater influence on the value function than small values, *H2* converges large values before propagating their influence throughout the state space. This tends to ensure that regions are fully converged before anything depending on the region is processed.

Figure 5.3 illustrates the difference. State A is an absorbing goal state with reward R. All other rewards are zero. In state B, the agent transitions to state A with probability $1-\gamma^5$, and back to state B with probability γ^5. Using the *H1* metric, B is backed up first, followed by C, D, then E. State B is then backed up again, followed by C, D, then E. The value of state B will asymptotically approach $R/(1-\gamma^6)$; for each update to B, an "information wave" must be propagated backward along the strand. Using the *H2* metric, however, B is repeatedly backed up until $R/(1-\gamma^6)-V(B) < \epsilon$, at which point a single information wave will propagate along the strand.

5.3 Selecting Metrics

The results in Chapter 10 demonstrate that neither *H1*, *H2*, nor standard VI induce an optimal backup ordering for all MDPs. However, each performs better than the others for some problems. The question of which metric should be used on a new problem naturally arises, but it is difficult to find topological features which accurately predict the performance of each metric. In fact, the best metric is a hybrid of all three, as we shall see.

VI yields a very good backup order any time a problem is close to being fully connected. The obvious corollary is that VI is also very good for any subgraph that is close to fully connected. Value iteration performs poorly any time the problem exhibits strong linearity: this can be due to a large number of strongly connected components, a large graph diameter relative to the number of nodes, or highly sequential dependencies within long loops.

The *H1* metric performs best in graphs which have highly sequential dependencies, which occurs in acyclic graphs and in graphs with long loops. The *H1* metric excels at avoiding useless backups, but tends not to iron out feedback loops completely, meaning that states within such loops must often be processed multiple times.

The advantage of the *H2* metric is more difficult to quantify. *H2* tries to ensure that states have converged before moving on to those states' dependents. Conceptually, this is an appealing idea, but practically it is very difficult to implement without the addition of partitions: it needs some cycles to generate a different order than *H1*, but does poorly with too many cycles. Figure 5.4 illustrates a topology for which *H2* is highly suboptimal, and Figure 5.5 shows performance visually: *H2* selects one state and "spirals" its value upwards, then selects another state and spirals, then a third, and back to the first, in a loop. However, the optimal sequence is to spiral all

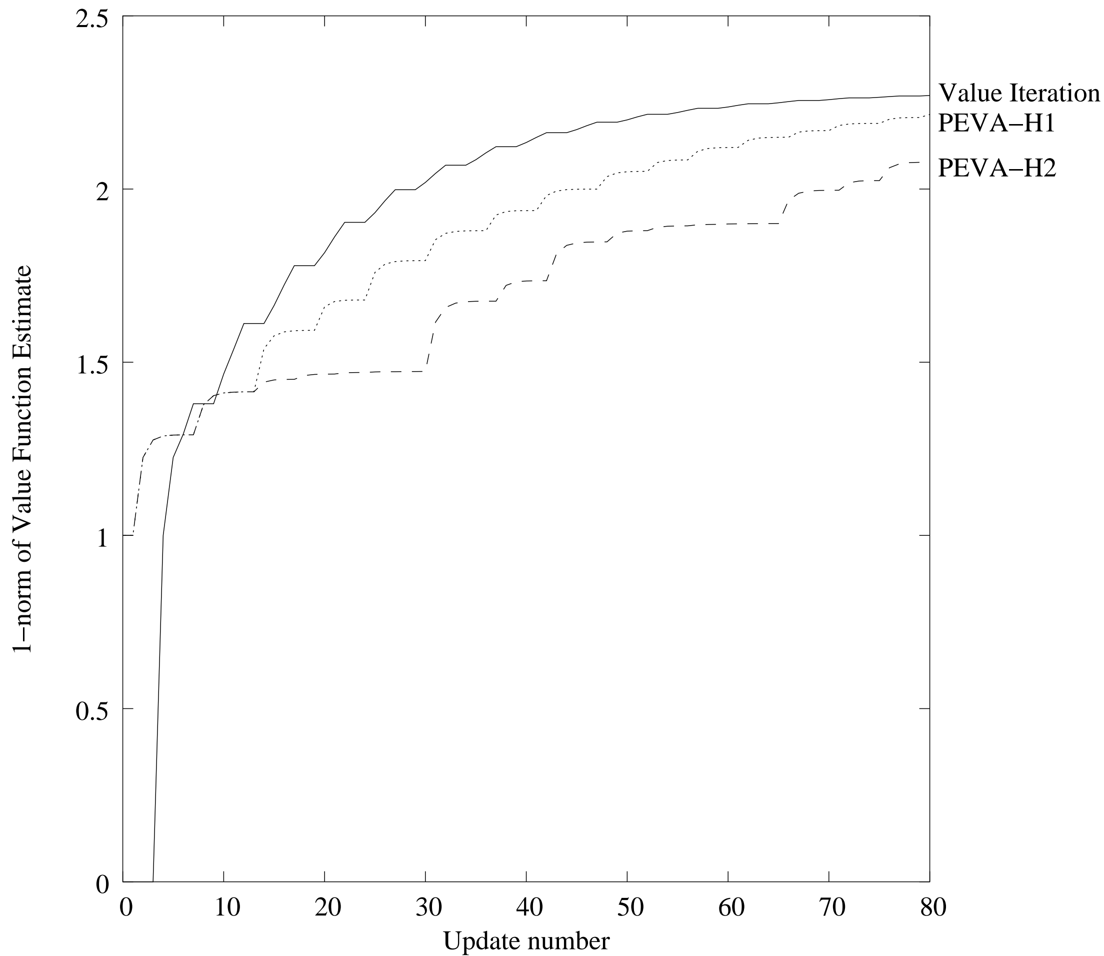

Figure 5.5: Performance against the topology in Figure 5.4. The graph shows the 1-norm of the value function as a function of the update number. The "spiraling" behavior of P-EVA-H2 is clearly shown: P-EVA-H2 selects one state and repeatedly backs it up it until it converges. This implicitly happens with P-EVA-H1 as well, because of Step #1 in the algorithm (see Figure 5.1).

three states upwards in parallel in a round-robin fashion, which implicitly happens with VI.

H2 performs best in a *hybrid* setting, which we will shall illustrate using Figure 5.6. Each cloud represents a cluster of highly interdependent states (perhaps even strongly connected components); clusters are weakly connected to each other. The values of states within each cluster should be converged before moving on to process the next cluster, but within each cluster, standard VI should be employed. By themselves, each metric performs poorly: VI performs useless backups by working on clusters two and three before information has propagated back to them; *H1* has the tendency to prematurely move on to the second and third clusters before the first cluster has converged, and *H2* correctly prioritizes clusters, but functions poorly within each cluster.

The desired hybrid should select a cluster and work on it until convergence, then move on to the next cluster. This is exactly the way that the P-EVA algorithm functions. Either *H1* or *H2* serves as a sort of meta-guide between partitions, but within each partition, normal VI occurs (see Figure 5.1, noting Step #1). This serves to explain why P-EVA performs so well when using many states per partitions, and why it performs so poorly when using just one state per partition. It also provides a theory as to why *H2* performs slightly worse on the double-arm pendulum problem than *H1*: states are highly interdependent, but *partitions* are highly interdependent as well.

5.4 Backup Order Voting

P-EVA must value iterate over all of the states within a partition until the Bellman error drops below ϵ. To optimize this step in the algorithm, we introduce the idea of

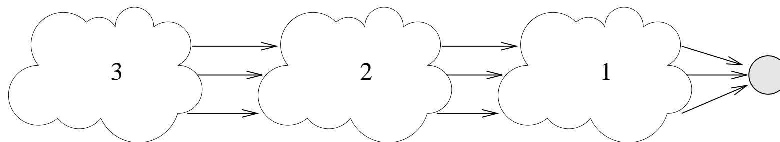

Figure 5.6: An example illustrating when hybrid metrics are close to optimal. Clouds represent clusters of highly interdependent states. The P-EVA algorithm does very well on problems of this sort if partitions correspond to clusters.

backup order voting.

To illustrate, consider the MDP shown in Figure 5.7. The flow of value dependency in this problem is regular, which is a common occurrence in many problems. Assuming that this system was updated from left to right, $O(n^2)$ backups will be required to propagate information from the right-hand side to the left-hand side. If, however, the partition is updated from right to left, only $O(n)$ backups will be required (of course, this difference is significant only when using Gauss-Seidel VI).

In this thesis, all partitions are hypercubes. The regular rectangular structure of the partition implies that states may be backed up simply by iterating systematically over the coordinates in the cube, in the much the same way a program might iterate over a multi-dimensional array. However, there is a choice to be made: in a given dimension, should we iterate from lowest coordinate to highest coordinate, or from highest coordinate to lowest?

Voting is the process of analyzing the transitions interior to the partition, and allowing each state to vote on backup directions for each dimension. This operation is facilitated by the experimental domain of the thesis: since all of the MDPs that we consider are derived from continuous time and continuous state control problems, each state has geometric information associated with it. This geometric information enables voting to be done with minimal overhead. Voting can therefore be used as a surrogate for an intra-partition topological sort (which is not well-defined for cyclic

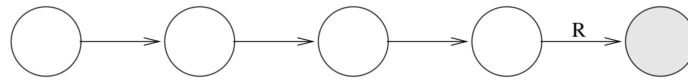

Figure 5.7: An example demonstrating how backup ordering can impact performance.

graphs).

It is important to note that most large problems do not generally exhibit enough dependency regularity to make simple direction-voting schemes helpful *at the global level*. Empirically however, once a problem is decomposed into small subgraphs–through a process like partitioning–voting improves performance substantially.

Admittedly, there are times when voting can hurt performance, but *some* backup order must be selected. Voting is a more principled method than relying on the relatively arbitrary order prescribed by memory organization. Developing a more general method of voting that does not rely on geomtric information is an issue left to future research. Some ideas are briefly discussed in Chapter 11.

Chapter 6

Parallelization

This chapter explores the results of combining partitioning, prioritization, and parallelization into a single VI algorithm. A central point of the chapter is the fact that algorithm designers do not need to trade one enhancement for another. All three are compatible, and even complimentary; the combination of any (or all) improves performance while maintaining convergence and optimality guarantees. We encourage the reader to refer to Wingate and Seppi (to appear 2004) for our most recent research into this subject.

There are practical and theoretical problems with the design of such a composite algorithm. Since other chapters have focused on prioritization and partitioning, the focus here is on the parallelization, and the unique issues resulting from a combination of all three ideas. Here, we answer questions related to scalability and efficiency by contributing insights into the design and implementation issues associated with parallelization. The most significant insights relate to an effective domain decomposition: we note that naive block-decomposition methods are unlikely to be effective, because of the way in which prioritization focuses computation on an "information frontier." We therefore analyze a heuristic decomposition designed to balance par-

allelization and prioritization. Other questions addressed include the following: can the idea of prioritized VI be efficiently implemented in parallel? Can the gains of the parallelization be quantified? How does a parallel, partitioned, and prioritized value iterator compare to a parallel naive value iterator? In this Chapter, we discuss the algorithm and enhancements, and we present experimental results in Chapter 10.

6.1 Parallel-P-EVA

The Parallel-P-EVA algorithm is a parallel version of the P-EVA algorithm. The core ideas remain the same, except that the work is spread out among different processes. This section describes the algorithm in detail.

Before we describe the algorithm, we first note that the synergy between the P-EVA algorithm and the Parallel-P-EVA algorithm is substantial. In fact, we argue that an effective parallel value iterator is possible in part because of the design decisions made and insights gained from the P-EVA algorithm. A partitioned value iteration algorithm naturally enables an efficient parallel value iteration algorithm, because partitions can be assigned to different processors, which allows them to operate on different parts of the problem in parallel. Since partitions are designed to minimize the amount of storage and computation required to prioritize backups, the same partitioning naturally reduces the amount of inter-processor communication in a parallel algorithm.

In the Parallel-P-EVA algorithm, a set of partitions is created (the number of which is greater than the number of processors, as discussed in the next section) and divided among the processors. The processors then executes the algorithm shown in Figure 6.1 asynchronously. Each processor maintains a local priority queue which prioritizes locally assigned partitions. The processor selects the local partition p with

Initialization

1. Partition the state space
2. Assign partitions to processors
3. Coordinate dependencies between processors
4. Construct a priority queue for partitions local to each processor

Repeat (in parallel)

1. Select the highest priority partition p from the local queue
2. Value iterate over the states within p, until the maximum change $< \epsilon$
3. Recompute the priorities of local partitions depending on p
4. Inform foreign processors about new values of states in p
5. Process incoming messages: for each foreign partition f that has changed, recompute the priorities of local partitions that depend on states in f

Until stopping criteria is met

Figure 6.1: Pseudocode for the the Parallel-P-EVA algorithm.

the highest priority and value iterates over the states within it until they converge. The priorities of local partitions depending on p are then updated, and the new values of states within p are communicated to processors that need to be informed. Naturally, there may be some partitions which do not contain any states whose values need to be communicated to a foreign processor.

The processor then processes incoming messages, which are of two types. The first type is "partition update" messages, which contain the values of states in foreign partitions that have changed. The second type is "termination" messages, which will be explained shortly. Of course, if no incoming messages are waiting, the processor may proceed to select another partition, and work on it normally. Once a processor has no more work to do (all local partitions have a priority that is below some threshold), it informs a designated master processor that it is finished, and blocks on incoming messages. This is part of the distributed stopping criteria. Once all processes have reported that they have finished, and that fact has been verified by the master processor (some additional checking to avoid a mild race condition) termination messages are sent to all processes. To avoid starvation, the maximum number of incoming messages that will be processed is equal to the number of processors; once those messages have been processed, the processor returns to working on the local priority queue.

This architecture has the effect of allowing processors to work independently whenever possible, but forcing synchronization when necessary. Of course, care must be taken with certain message passing libraries to avoid deadlock.

In a parallel value iterator, the issue of cross-processor dependencies is important. It is possible that states owned by one processor depend on the value of states owned by another processor. In fact, the *priority* of a local partition will often depend on the value of states owned by foreign processors. This necessitates important steps which

are unnecessary in serial versions. First, in the initialization phase of the algorithm, all dependencies are coordinated between processors. Each processor must inform foreign processors that they wish to be updated when the value of a dependency changes. Of course, the value of a state will only change once a processor has selected a local partition and backed up the states within it. Thus, after processing a partition, a processor must communicate the values of states in the partition to all foreign processors that depend on those states. This additionally implies that processors must maintain some sort of cache of the last known value of a foreign state, which adds some space complexity. Cross-processor transitions must be minimized for several reasons: first, to simplify the coordination phase; second, to minimize the number and size of messages communicated after processing a partition; and third, to minimize the size of the foreign state cache.

6.2 Assigning Partitions to Processors

There are several problematic issues involved in the design and implementation of the Parallel-P-EVA algorithm. These may be divided into two broad categories: first, there are theoretical issues related to optimality, convergence, and appropriate decompositions. Second, there are low-level issues related to stopping, deadlock avoidance, and efficient internal representation. Since low-level issues are largely implementation dependent, and discussion of them would detract from the focus of the thesis, they will not be discussed in detail. The convergence and optimality issues are grouped with similar issues in the P-EVA algorithm, and are discussed in Chapter 8.

The most interesting problems confronting the Parallel-P-EVA algorithm involve appropriate domain decompositions. There are two broad issues: first, how are states allocated to partitions? Second, how are partitions allocated to processors? The issue

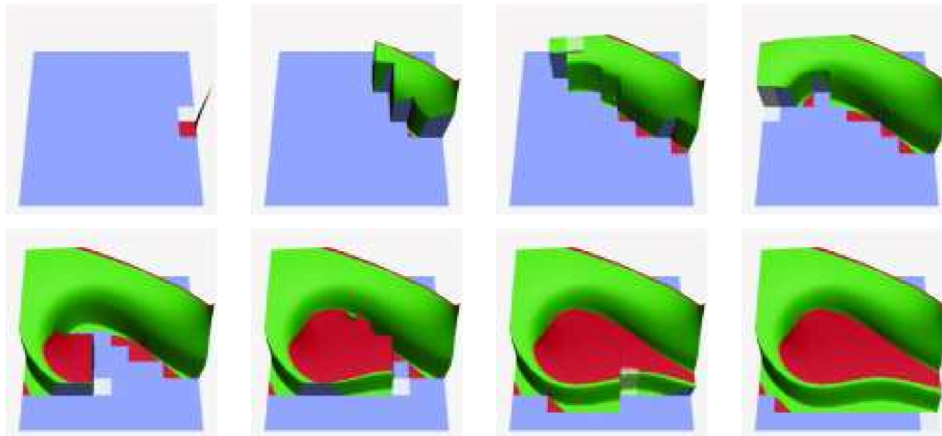

Figure 6.2: Moving from left to right, top to bottom: the "information frontier" of the Mountain Car value function propagates outward from the primary reward. Colors are not relevant to the point of the figure. These images were generated using a slightly modified version of the Mountain Car reward function, as discussed in Chapter 9.

of allocating states to partitions is not treated in this work, for the following reason. The focus of this chapter is exploring the general benefit of parallelization, and not in tuning the many specific design choices involved. The barrier to entry of creating principled partitions is quite high, and an adequate partitioning can be constructed simply by using the geometric coordinates of each state. This is a consequence of our experimental setup. The MDPs are derived from continuous state optimal control problems, and the states have associated coordinates in the original state space, meaning that partitions of highly related states can be generated by gridding the state space. Solving more general MDPs for which geometric information is not available is an important issue that has been left for future research. It is anticipated that existing k-way minimum-cut graph partitioning algorithms, such as recursive spectral bisection (Alpert, 1996) or parallel multilevel partitioning (Karypis and Kumar, 1996) will be useful in partitioning such problems.

The issue of allocating partitions to processors is more interesting, and is one of

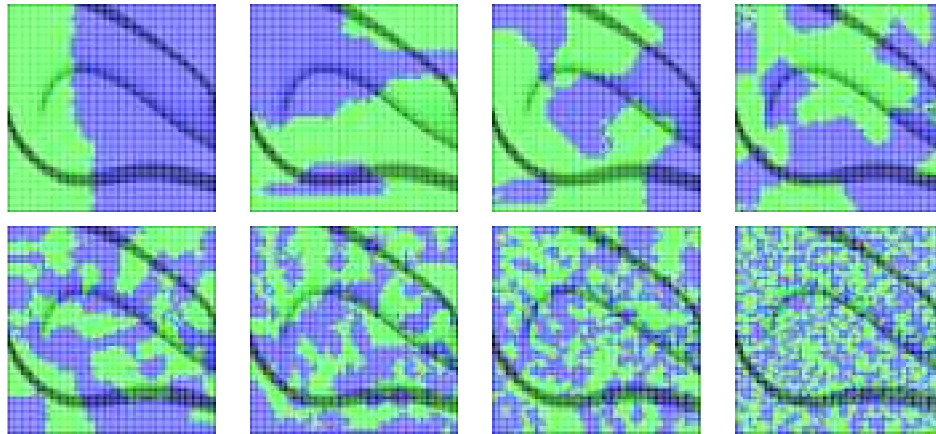

Figure 6.3: The assignment of partitions as a function of attractors. Shown is a top view of the MCAR value function. Blue (dark) and green (light) colors indicate assignment to processor one and two, respectively. Shown are the resulting assignments for (upper-left to lower-right) 1, 5, 10, 20, 50, 100, 200, and 1000 attractors per processor. The more attractors are added, the more random the partition assignment appears to be.

our focuses. The difficulty is finding a balance between parallelism and prioritization, and is best explained through an example. Consider Figure 6.2. Shown are frames in the evolution of the MCAR value function, demonstrating the backpropagation of value information throughout the state space. This value information travels as an "information wave." Formally, this information wave is dV/dt. If desired, the reader may refer to Chapter 9 for a a detailed description of the MCAR problem.

The prioritization metrics are designed to focus computation on the crest of the wave, between the region of not-yet-processed and already-converged. The problem lies in the fact that a traditional block decomposition will not yield good parallelization. As an example, consider solving the MCAR problem with two processors. Assume that the problem was divided into roughly equal blocks named A and B (as shown in the upper-left image of Figure 6.3), and that block A (on the left) was assigned to processor one, and that block B was assigned to assigned to processor

two. This decomposition would perform quite poorly: processor one would start off idle, while processor two would drive the information wave from the primary reward until it exited block B and entered block A. Processor two would then be largely idle, while processor one drove the information wave around the curve, and back into block B. Finally, processor one would be idle, as processor two completed pushing the information wave through the state space.

The foregoing example suggests two ideas. First, it is clear that the ideal scenario is to have all processors work on the information frontier in parallel, but this is difficult to do, since it is not known in advance how the information frontier will progress through the state space. Second, it seems clear that creating more partitions than processors may allow the processors to always be working in parallel (for a variety of technical reasons, we do not consider the *dynamic* allocation of partitions to processors; partitions are allocated once during initialization, although this idea represents a significant direction for future research). This suggests that a fully random allocation of partitions to processors may be a viable solution, but intuitively, it seems that this would create little cohesion between partitions, and would result in prohibitive inter-processor communication.

At one extreme of the decomposition, therefore, is a full block decomposition, where the state space is divided equally among processors in large contiguous regions. At the other extreme is a fully random allocation. To explore this continuum, we introduce the idea of "attractors," which are essentially nodes in a radial basis function. We allocate n attractors per processor, and scatter them randomly throughout the state space. To assign a partition to a processor, the partition's distance to all of the attractors is computed; the partition is assigned to the owner of the nearest attractor. This effectively computes a Voronoi tessellation of the space. Adding more attractors makes the assignment more random, as shown in Figure 6.3.

Chapter 7

On-Demand Data

The algorithmic enhancements we have discussed have all represented improvements in *temporal efficiency*. They improve the running time of the algorithm, but do so at the expense of extra space. One of the most significant results of the thesis (which is discussed in detail in Chapter 10) is the fact that our algorithms are so temporally efficient that the factor limiting the size of problems which can be solved is no longer time, but RAM.

This chapter briefly explores one idea related to *spatial efficiency*, which is "on-demand data." As we scale to solve larger and larger problems, we become unable to store all of the model information and supporting data structures in RAM. Since all of our algorithms operate at the partition level of granularity, and any given machine is only processing one partition at any time, there are large amounts of RAM which are not actively in use. This implies that we may be able to discard some information for inactive partitions, and then recall it (or regenerate it) "on-demand" – that is, when we decide to process the partition. There are two options: recompute the discarded data, or cache it on disk. Section 7.1 explains both ideas in depth.

As problem sizes grow larger and larger, algorithms eventually *must* make use

of some form of an "on-demand" strategy. An important aspect of the behavior of our algorithms, therefore, is whether or not they facilitate good cache performance. Although we will not study the idea of on-demand data as thoroughly as other ideas, we feel that it is important to examine the idea briefly, because it contributes significantly to the overall goals of the thesis. There are several reasons for this. First, it allows us to solve ever larger MDPs, which is one of the goals of the thesis. However, the most significant reason is because *good cache performance for non-predictive caches is only made viable by prioritized partitioning.*

During the course of the many experiments run for this thesis, we noted a very important fact. For almost all problems, the average number of times any given partition is visited is quite low – usually less than ten. Considering that normal VI usually requires hundreds or thousands of sweeps (each of which must touch every partition), this implies that the localization and focus of computational effort through prioritization is what makes effective caching of partition data possible.

Thus, when we evaluate the utility of various enhancements, we will add "cache efficiency" to the list of performance criteria. For example, in most experiments, the *H2* priority metric outperforms the *H1* metric. Sometimes the *H1* priority metric does better, but only in terms of wallclock time. *H2 always* yields better cache efficiency.

7.1 Writing to Disk vs. Recomputing

As mentioned previously, there are two possible variants of "on-demand data." This section explains both in detail.

By far, the largest consumer of RAM is the model information, which consists of several sets (one per action) of transition probabilities for each state. This implies that

two RAM management strategies are possible: first, since all of the model information is derived directly from system dynamics models (as discussed in Chapter 9), it may be possible to discard model information until it is needed to process a partition, and then recompute it on-demand. This option suffers from a lack of generality, because it does not apply to any MDP which is not derived from a system dynamics model. The second option is a more traditional approach, which is to store the data on disk, and read it back on-demand.

This thesis explores the second option, for several reasons. First, as noted, is the issue of generality. Second, several preliminary benchmarks indicate that although prioritized, partitioned value iteration makes the recomputation *more* feasible, recomputing transition information is still too expensive compared to a disk-based approach. On the MCAR problem, for instance, P-EVA computes transitions at the rate of about one transition every 0.0000812 seconds. Assuming that there were about 100 states in a partition, computing all of the transition information for a partition would require about 0.008 seconds. However, this is the average seek time for standard hard disks, which is the primary latency factor for reading data from disk. In this situation, either option is basically equivalent, but if the transitions were more expensive to compute (as is the case for the triple-arm pendulum), or partitions were to consist of several hundred (or even thousand) states, reading them from disk would always yield the best performance.

7.2 Normal VI and Predictive Caches

As a baseline, it is important to consider the cache behavior of normal VI, using a caching variant of on-demand data. Normal VI exhibits pessimal behavior for non-predictive caches, because as it iterates over a problem, it touches each partition

once, and then moves on to the next partition. It never revisits a partition until it has visited every other partition in the problem. Assuming a cache strategy that kicks out the least-recently-used element, any cache smaller than the entire problem will always yield a hit ratio of zero.

It is possible to improve this performance through the use of a predictive cache, since the order in which partitions will be accessed is known a priori. This is very feasible, and could probably be tuned to provide almost optimal cache performance. In addition, data reads could be executed in threads that were concurrent to the main program, to ensure the least amount of blocking possible. However, such a predictive cache will not be explored further, either for normal VI or for any of our algorithms. Such a study would detract from the goal of the thesis, and would complicate the experimental matrix considerably. The many issues surrounding predictive caches represents an excellent research space that we leave for future work.

Instead, we wish to limit our evaluation strictly to non-predictive caches. The reason we discuss the difference between predictive caches and non-predictive caches is because most virtual memory managers employ some version of a non-predictive cache (although technically some VMMs can pull blocks off disk that are spatially related to the requested block, such locality on disk does not necessarily correspond to locality in the *problem* space). Assuming that an algorithm designer allowed the operating system to perform all caching (probably by writing memory blocks to swap space), the performance of our algorithms under such a non-predictive cache would be important.

7.3 Information-Frontier-Only Statistics

The final observation we make relative to on-demand data relates to the *intrinsic* cacheability of a problem.

In the P-EVA algorithm, there are two different times that partitions are needed. The first happens when a partition p is extracted from the priority queue as having the highest priority. p becomes the working partition, and if it is not in cache, it must be retrieved from disk. We term these the "information-frontier-only" partition accesses, for reasons explained below. However, there is a second time that partitions are needed. When we have *finished* processing p, we must recompute the priority of any partition d that depends upon any state in p, meaning that the transition information for certain states in d will be needed. This in turn means that if d is not in cache, it must be read from disk. We term these "auxiliary partition accesses."

Currently, the P-EVA algorithm pulls the entire contents of d out of the cache for each auxiliary access, but this is not strictly necessary. It is possible (and even probable) that not *all* states in d depend upon some state in p, so it may be possible to pull out only the necessary states. Although disk latency may still be a factor, this may be reduce the time needed by the disk read. Or, it may be feasible to recompute the transition information for just the states in d that depend on some state in p, instead of recomputing the entire partition.

The distinction between "information-frontier-only" and "auxiliary" partition accesses is significant for another reason. Although we do not pursue this idea in this thesis, it may be possible to *approximate* the priorities between partitions, instead of recomputing them exactly. Such an approximation may not require all of the transition information in the d partition, which means that an auxiliary partition access may not be necessary.

We bring this up to illustrate that there are two different measures of cache performance: first, there is the cache performance of the algorithms *as they stand*, and second, there is the *intrinsic cacheability* of the problem itself. Of course, both measures must be taken with respect to a given priority metric.

Counting auxiliary partition accesses dramatically changes the cache performance characteristics of our algorithms. For that reason, the results in Chapter 10 report two sets of cache efficiencies, one which includes the auxiliary partition accesses, and the other which does not.

Chapter 8

Algorithm Analysis

In this section, we analyze the properties of the P-EVA algorithms under either priority metric, with partitioning, and in a parallel scenario. We present proofs and equations for the maximum difference and stopping criteria (Section 8.2), convergence (Section 8.3), rates of convergence (Section 8.4), and present some results on sufficient subsets (Section 8.5).

8.1 Analysis Notation

For this section, we adopt the more conventional timestep notation where t is incremented once per sweep. Let $n = |S|$. Let $\|V\|_1 = \sum_{i=0}^{n} |V_i|$ be the 1-norm (or Manhattan norm) of the vector V, which contains the values of all states. Let $E_t = \|V^* - V_t\|_1$ be the true sum error of the value function estimate. All vectors will be column vectors.

As noted in Section 4.1, most of our algorithms execute partial sweeps, as opposed to full sweeps. Suppose that, instead of backing up all of the states in the problem, only a subset p is backed up. To accomplish this notationally, we begin by defining

the *selector matrix* of a set p of states at time t as the n x n diagonal matrix

$$\mathbf{K}^{tp}_{ii} = \begin{cases} 1 & i \in p \\ 0 & \text{otherwise} \end{cases}$$

All other entries in $\mathbf{K}^{tp}$ are 0. Based on Equation 2.8, the partial update is then easily expressed as

$$V_{t+1} = V_t + \mathbf{K}^{tp} B_t \tag{8.1}$$

8.2 Maximum Difference and Stopping

It is easy to characterize the largest difference between a value function estimate and the optimal value function in terms of the Bellman error magnitude. This has already been accomplished by Williams and Baird (1993); similar results can be easily derived by using equation 6.3.7 of Puterman's book (Puterman, 1994):

$$\|V_t - V^*\| \leq M_{t-1}/(1-\gamma) \tag{8.2}$$

The maximum difference provides a natural stopping criteria. The algorithm can stop when $M_t < \epsilon(1-\gamma)$ and will be guaranteed to have an ϵ-optimal policy. A more common bound (for example, Puterman, 1994) is that if $\|V_{t+1} - V_t\| < \epsilon(1-\gamma)/2\gamma$, then $\|V_{t+1} - V^*\| < \epsilon/2$. The slight difference in the two equations can be accounted for by noting that we previously stipulated that all rewards be positive (which allows us to provide a tighter bound by avoiding absolute values), and because of a minor difference in time subscripting.

This stopping criteria is particularly useful because, as explained in Section 8.4, many of the sequences involved in the P-EVA algorithm do not converge in norm.

8.3 Convergence

Theorem 8.3.1. *The sequence of value function estimates* $\{V_t\}$ *generated by performing backups based on a sequence of selector matrices converges uniformly to* V^**, provided that* $\|\mathbf{K}^{tp} B_t\|_1 > 0$.

Proof.

$$\begin{aligned} \|V^* - V_{t+1}\|_1 &= \|V^* - TV_t\|_1 \\ &= \|V^* - V_t - \mathbf{K}^{tp} B_t\|_1 \\ &= \|V^* - V_t\|_1 - \|\mathbf{K}^{tp} B_t\|_1 \\ E_{t+1} &= E_t - \|\mathbf{K}^{tp} B_t\|_1 \end{aligned}$$

Since $\|\mathbf{K}^{tp} B_t\|_1 > 0$, $E_{t+1} < E_t$, implying that the true error of the system is monotonically decreasing. To prove uniform convergence, it must be true that for any $\epsilon > 0$ there exists an N such that $\|V_t - V^*\| < \epsilon$ for all $t > N$. This can be easily established because $V_t < V^*$, but can never overshoot it. Thus at some t, the states responsible for $\|V_t - V^*\|$ will have the only Bellman error and will be backed up. Convergence to V^* is established by noting that when $\|B_t\|_1 = 0$, $TV_t = V_t$, indicating that V_t is the unique fixed point of the value function space. □

Corollary 8.3.1. *P-EVA converges to* V^**, using either the H2 or H1 priority metric.*

Proof. At each time t, and using either priority metric, P-EVA picks some partition p with non-zero Bellman error, which ensures that $\|\mathbf{K}^{tp} B_t\|_1 > 0$. This meets the conditions stated in Theorem 8.3.1, implying that P-EVA converges. □

8.4 Rates of Convergence

The standard contraction mapping definition $\|Tv - Tu\| \leq \gamma\|v - u\|$ implies two useful properties:

$$\|V_{t+2} - V_{t+1}\| \leq \gamma\|V_{t+1} - V_t\| \tag{8.3}$$

$$\|V^* - V_{t+1}\| \leq \gamma\|V^* - V_t\| \tag{8.4}$$

Equation 8.3 can be interpreted to mean that successive value function estimates must draw closer and closer together, while Equation 8.4 can be interpreted to mean that successive value function estimates must draw closer and closer to the optimal value function.

Now, assume that a set is constructed which represents the minimal number of states which need to be backed up in order to force a contraction, in the sense of Equation 8.4. It is then possible to construct examples where (8.4) holds, but where (8.3) does not. This implies that while (8.4) is a necessary condition, (8.3) is merely a sufficient condition.

In fact, this often occurs with the P-EVA algorithm (although this is mostly an artifact of the timestep notation). The increased efficiency of the P-EVA algorithm can be partly explained by this violation of property (8.3). To explain further, let us revert to the definition of a timestep where t is incremented after every update. Recalling that the one-norm of a vector can be viewed as a sort of discrete integral over the vector, an obvious efficiency metric might be

$$\texttt{efficiency}_t = \|V_t\|_1/t$$

(where higher efficiency is better, and noting that this metric is only meaningful in the positive bounded case). This efficiency metric gives top scores to algorithms that create more "value function estimate mass" in fewer updates. According to this

measure of efficiency, increasing the largest difference between two value function estimates helps because it creates situations where a single backup may be able to generate a large amount of value function mass. Conversely, normal VI deliberately makes all differences uniformly smaller and smaller.

We feel that it would be appropriate to point out the following. We adopted an unconventional timestep notation because P-EVA does not perform full sweeps. Because t is incremented after every update, the following is true:

- The sequence $\{V_t\}$ does not contract in either max-norm or a span semi-norm.
- The sequence $\{B_t\}$ does not contract in either max-norm or a span semi-norm.
- The sequence $\{M_t\}$ can increase and stabilize at a non-zero value, but that only implies that the largest error is constant, and not that it is vanishing. Eventually, due to the finiteness of the system, $\{M_t\}$ will eventually go to zero, but a precise characterization of how long $\{M_t\}$ will increase has not been found.

However, remember that convergence is still guaranteed. The fact that the sequences do not converge is an unfortunate consequence of timestep notation. The rate of convergence of P-EVA is therefore difficult to quantify in terms of t. Since counter-examples can be found which indicate that P-EVA performs no better than value iteration, we hypothesize that a lower bound on the rate of convergence is simply the same bound as that of VI. An upper-bound, and a problem-dependent bound, remain to be found.

8.5 Sufficient Subsets

The following theorem partially answers the question posed at the beginning of Section 4.1: "what is the minimal set of back ups needed to force a contraction?" Here,

we show that one sufficient (but not necessarily minimal) set is the set of states with non-zero Bellman error.

Theorem 8.5.1. *Let $G_t = \{s : B_t(s) \neq 0\}$. An operator T which backs up all $s \in G_t$ is equivalent to normal value iteration.*

Proof. Let $\mathbf{K}^{tG}$ be the selector matrix associated with G_t. Then, $B_t = \mathbf{K}^{tG} B_t$, and by (8.1), $V_{t+1} = V_t + \mathbf{K}^{tG} B_t = V_t + B_t$, which is equivalent to normal value iteration. □

A tighter bound on the size of this set may be achieved by noting that $M_t \leq \|V_t - V^*\| \leq M_t/(1-\gamma)$. To contract by a factor of γ, at least one state must change value; the minimum amount of change that could possibly be necessary is $M_t - \gamma M_t$. G_t may therefore be redefined as $G_t = \{s : B_t(s) \geq M_t - \gamma M_t\}$; backing up all $s \in G_t$ will ensure contraction.

We hypothesize that a more precise characterization of the minimal subset will depend on more granular topological features of the transition matrix.

8.6 Convergence and Optimality of Parallel-P-EVA

Convergence of the Parallel-P-EVA is established by appealing to Bertsekas (1982), in which he provides proof that asynchronous value iteration converges without the assistance of a common clock. The scenario described in his paper exactly matches the setup of the Parallel-P-EVA algorithm. We have already shown that the addition of partitioning and prioritization does not affect convergence, because the convergence guarantees are provided for arbitrary backup orderings. The Parallel-P-EVA algorithm simply selects one of many backup orderings, which happens to be principled and efficient.

The optimality of the final solution is also uncompromised. It is well-known that the optimal solution of a value-iteration process is a unique fixed point (Puterman, 1994), and that if the maximum Bellman error

$$\|V_t - V_{t+1}\| = \epsilon$$

then

$$\|V_t - V^*\| \leq \epsilon/(1-\gamma) \tag{8.5}$$

Parallel-P-EVA stops when all processors report that the maximum Bellman error is less than ϵ, which satisfies Equation 8.5. Since the fixed-point is unique, the policy computed is within $\epsilon/(1-\gamma)$ of optimal.

Chapter 9

Experimental Setup

The P-EVA family of algorithms (including Parallel-P-EVA) was validated by running each algorithm against several problems of differing complexity. We will first describe parameters that were general to all of the algorithms that we tested, and will then discuss setups that were specific to each research objective. Section 9.3 discusses the experiments regarding partitioning, priority metrics, and voting. Section 9.4 discusses the experimental setup for parallelization. Section 9.5 discusses the setup for our ideas related to on-demand data. Results of all experiments are discussed in Chapter 10, and are presented in graphical form in Chapter 12.

Success was measured by the amount of time taken, by the number of backups performed, by how accurate the resulting value function was, and in some cases, by the cache hit ratio. We note that all of the selected problems were single-reward systems; multi-reward and continuous-reward problems have been left for future research.

All algorithm variants always used Gauss-Seidel updates. For all experiments, ϵ was set to 0.0001 and γ was set to 0.9.

For each of the test problems, partitioning was done by overlaying the initial discretized state space with another grid. This is a low-cost way to generate sets of

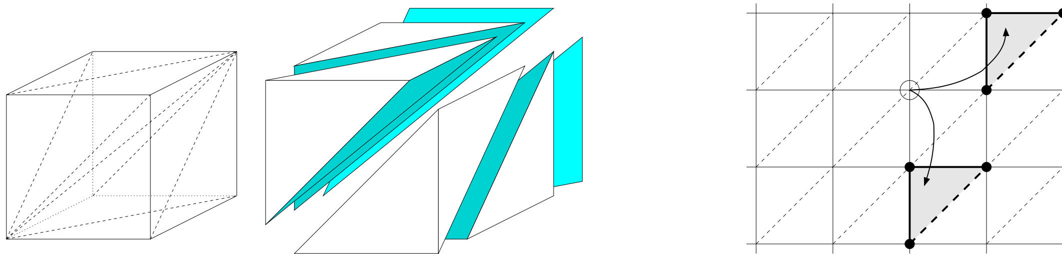

Figure 9.1: On the left, the Kuhn triangulation of a (3d) cube. A d-dimensional hypercube is tessellated (implicitly) into $d!$ simplices. On the right, control of each (s, a) pair is tracked until the resulting state s' enters a new hypercube. Barycentric coordinates relative to the enclosing simplex are computed, and are used to represent probabilistic transitions to vertices.

highly inter-dependent states, especially considering the method used to discretize the state space, and works well because it exploits the locality and continuity of a problem. Various grids were tested; the best was a simple grid with square cells. Chapter 11 discusses more general ways to generate partitions that do not rely on geometric information (an issue which has been left to future research).

Each of the problems tested are continuous time, and involve continuous action and state dimensions. These were discretized as described in the next section, but we note here that this process is tangential to the research focus of this thesis. There are many other methods which could have been used to discretize the problems; naturally, this particular method introduces a bias with respect to the original problem, but since the solution engine simply expects a discrete MDP, the details of where it came from are somewhat irrelevant.

9.1 Discretizing the Space

To discretize the space, we use the same approach described by Munos and Moore (2002), except that no *variable* discretization is used. Instead, the space is discretized once in the initialization phase. We refer the reader to their work for a complete

description of the technique with comprehensive citations on component elements.

The state space is divided into hypercubes by regularly dividing each dimension. A Kuhn triangulation is implemented (implicitly) inside of each hypercube. The hypercubes completely tessellate the space, and the Kuhn triangles completely tessellate each hypercube. The vertices defining the hypercube grid are used as the states in the MDP. The transition matrix is computed by iterating over each vertex s. For each available action a, the system dynamics are integrated using Runge-Kutta and tracked until the resulting state s' enters a new cube. The barycentric coordinates of s' with respect to the enclosing simplex are then computed. The state s can then be said to transition non-deterministically to a vertex in the enclosing simplex with probability equal to the related barycentric coordinate (since barycentric coordinates always sum to one). As Munos and Moore (2002) state, "doing this interpolation is thus mathematically equivalent to probabilistically jumping to a vertex: we approximate a *deterministic* continuous process by a *stochastic* discrete one" (emphasis in original). Figure 9.1 shows two and three dimensional examples of the discretization process.

Approximation of the value function is performed by computing exact values at each of the vertices, and interpolating the value across the interior of each cube. Interpolation is linear within each simplex. Since these problems are continuous time, a slightly different form of the value function equation was used:

$$V_t(s,a) = \int_0^\tau \gamma^t R(s(t),a)dt + \gamma^\tau \sum_{s'} Pr(s'|s(\tau),a) \max_{a' \in A} V_{t-1}(s',a')$$

where τ is the amount of time it took for s' to enter the new cube (or exit the state space), with the convention that $\tau = \infty$ if s' never exited the original hypercube.

Problems with continuous state spaces were selected because the number of states used in the discretization process could be varied at will. The use of Kuhn triangles

was selected as a discretization method because once discretized, each state depends upon exactly $d + 1$ other states. The combination of these two factors allowed us to smoothly vary the size of the problem (thus generating families of highly related MDPs), while maintaining a constant outdegree, and it allowed us to easily generate partitions. In addition, the combination of hypercubes and Kuhn triangles has excellent space and time performance characteristics, which greatly accelerated the experimental cycle.

Once discretized, the model inverse was then computed.

9.2 Test Problems

Four test problems were used to quantify the performance of the P-EVA family of algorithms. These were "Mountain Car" (or "MCAR"), the single-arm pendulum (or "SAP"), the double-arm pendulum (or "DAP"), and the triple-arm pendulum (or "TAP"). The MCAR problem is well-known in the reinforcement learning literature, and is generally considered to be an easy problem. DAP is a canonical engineering problem, but this particular variant is not commonly used in reinforcement learning literature. SAP was introduced for the first time by Wingate and Seppi (2003), and TAP is being introduced to the RL community for the first time in this thesis.

9.2.1 Mountain Car

Mountain car is a two-dimensional control problem, characterized by position and velocity. A small car must rock back and forth until it gains enough momentum to carry itself up to the top of the hill. In order to receive any reward, the car must exit the state space on the right-hand side (positive position), with a velocity close to zero. In order to make results more comparable to the other problems studied,

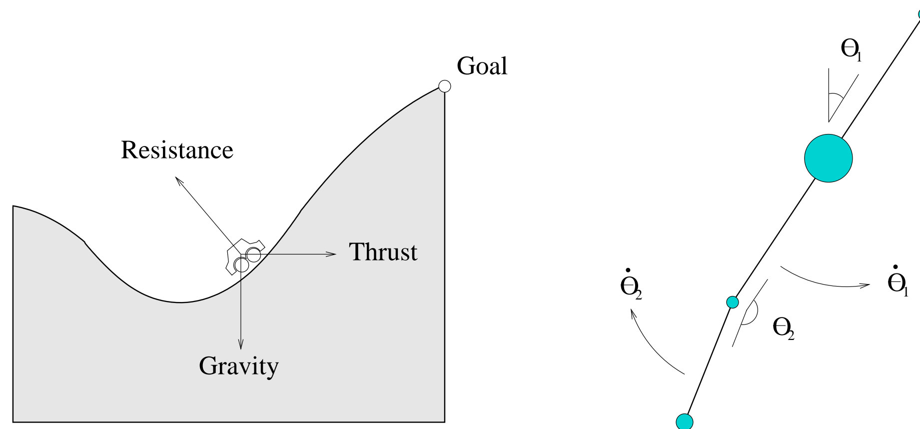

Figure 9.2: The left figure shows the mountain car problem (figure adapted from Munos and Moore, 2002). The car must rock itself back and forth to generate enough momentum to exit the state space. The state space is described by the position and velocity of the car. The right figure shows the double-arm pendulum. The agent must swing the secondary link into the vertical position and keep it there. The state space is described by four variables: θ_1, θ_2, $\dot{\theta_1}$ and $\dot{\theta_2}$. The same dynamics are used for the single-arm pendulum, and similar dynamics are used for the triple-arm pendulum, except that a second free link is added.

the reward function was modified from the traditional gradient reward to be a single-point reward: the agent received a reward only upon exiting the state space with a velocity of zero (plus or minus a small epsilon). This did not substantially change the shape of the resulting value function.

9.2.2 The Pendulum Family

Double-arm pendulum

The double-arm pendulum is a four-dimensional optimal control problem. The agent has a single action available, representing torque applied to a primary link. The second link is free-swinging, which the agent must balance vertically. Since the problem is minimum-time, bang-bang control is sufficient; two actions are selected, representing positive and negative torques. Similar to the mountain car, the agent cannot move

the pendulum from the bottom to the top directly, but must learn to rock it back and forth to generate sufficient momentum. This variant of the double-arm pendulum is different from the easier Acrobot problem, where force is applied at the junction between the two links (Sutton, 1996), and from a horizontal double-arm pendulum (where the main link rotates in the horizontal plane, and the secondary link rotates vertically with respect to the main link). Our version of the double-arm pendulum is the complete swing-up-and-balance problem; other variants only treat the balancing aspect. The two actions are ±10 Newton. Acceptable angular velocities were limited to ±10 radians/s for $\dot{\theta}_1$, and to ±15 radians/s for $\dot{\theta}_2$.

Single-arm pendulum

The single-arm pendulum uses the same dynamics as the double-arm pendulum, but the agent must only learn to balance the main link. Again, the agent must rock the pendulum back and forth until it moves into position. The state space is described by θ_1 and $\dot{\theta}_1$. Although conceptually similar to the mountain car problem, both problems have very different value functions. Thus, the way in which value function information backpropagates through them is also very different. This is shown graphically in Figure 9.3.

Triple-arm pendulum

The triple-arm pendulum uses the same general dynamics as the double-arm pendulum, except that an additional free link is added to the system. The state-space is therefore six-dimensional, being described by θ_1, $\dot{\theta}_1$, θ_2, $\dot{\theta}_2$, θ_3, and $\dot{\theta}_3$. The state space was bounded by clipping $\dot{\theta}_1$ at ±8 radians/s, $\dot{\theta}_2$ at ±10 radians/s, and $\dot{\theta}_3$ at ±10 radians/s. The two actions evaluated were ±10 Newton.

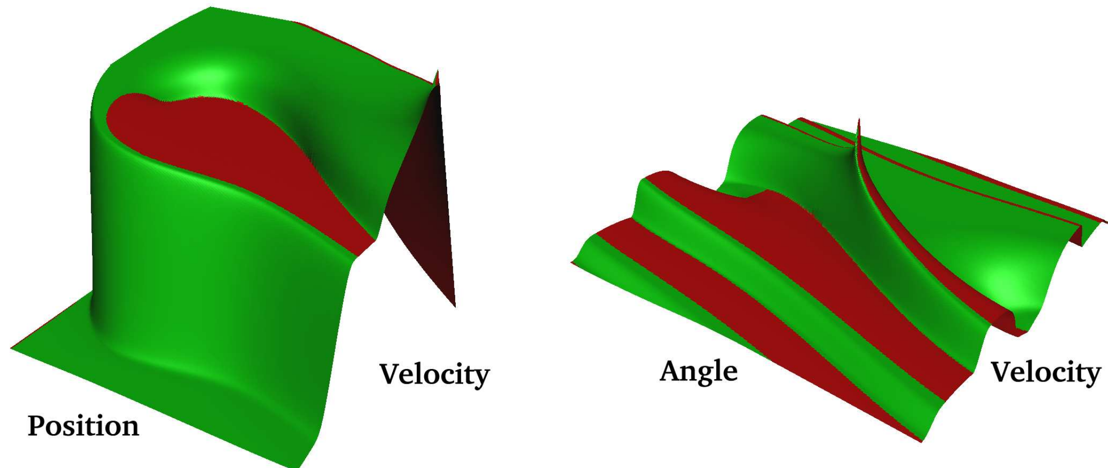

Figure 9.3: On the left, the value function for the Mountain Car problem. On the right, the value function for the Single-Arm Pendulum. Red (dark) and green (light) colors indicate different controls. Information propagates through each problem in very different ways.

9.3 Prioritization and Voting Setup

The bulk of the experimentation was designed to quantify the relative impacts of partitioning, prioritization, and voting. Since partitioning is meaningless without a priority metric, experiments were run in which normal VI was compared to P-EVA using either the *H1* or the *H2* metric. In addition, each variant was tested with and without voting. Two different kinds of experiments were run. Some were designed to quantify the performance gains from prioritization, and some were designed to help find the optimal partition size.

To quantify the impact that voting had on normal VI, the problem was partitioned, and a voted backup order was computed within each partition. Then, for each sweep of VI, the algorithm iterated over all partitions, and updated all states within each partition in the voted direction. Thus, each sweep updated every state once, but in a voted order.

This generated a final experimental matrix of six algorithms: normal VI, normal

VI with voting, P-EVA-*H1*, P-EVA-*H1* with voting, P-EVA-*H2*, and P-EVA-*H2* with voting.

The baseline performance is that of normal VI, without any voting or prioritization. All results were obtained on a 2.8GHz Pentium 4 with 2G of RAM.

9.4 Parallel-P-EVA Setup

Several sets of experiments were run to benchmark Parallel-P-EVA. First, scalability tests of Parallel-P-EVA were run, to determine the approximate efficiency of the algorithm as the number of processors was increased. Secondly, experiments were run comparing Parallel-P-EVA to a parallel version of standard VI (called "PSVI"), in an attempt to quantify the relative gains of prioritization and parallelization. Third, experiments were also run comparing Parallel-P-EVA to the P-EVA algorithm, which is partitioned and prioritized, but not parallelized. Although Parallel-P-EVA is the successor to P-EVA, certain code and data reorganizations necessary for parallelization mean that P-EVA outperforms Parallel-P-EVA when Parallel-P-EVA uses one processor. Finally, experiments were run to evaluate the impact of different partition-to-processor mappings, using the attractor-based system described in Section 6.2. In all other experiments, partitions were allocated to processors randomly.

The efficiencies of the parallel algorithms are also reported. Efficiency is computed as

$$e = \frac{T_1}{p * T_p}$$

where T_1 is the amount of time required to solve the task on one processor, p is the number of processors, and T_p is the amount of time required to solve the task using p processors. Higher efficiencies are better; efficiencies greater than 1.0 represent superlinear speedup.

The naive parallel implementation of standard VI (or PSVI) is essentially a non-prioritized version of Parallel-P-EVA. States are aggregated into partitions, and partitions are assigned to processors. Instead of prioritizing partitions, however, each processor sweeps over every partition repeatedly. After processing a partition, the processor communicates new state values to foreign processors in exactly the same way that Parallel-P-EVA does.

All priority queues used the *H2* priority metric. All code was implemented in C, using MPI[1]. Experiments were run on a fully connected cluster of dual processor 2.4GHz Pentium 4s, each having 2G RAM and Myrinet interconnects.

9.5 On-Demand Data Setup

Several experiments were run to explore the cache behavior of the P-EVA algorithm. The primary questions relate to the cache efficiency of the *H2* and *H1* metrics, and not parallelizability, so only the P-EVA algorithm was tested. Naturally, voting did not affect cache behavior, so it was not included in the experimental matrix.

All cache tests used a non-associative cache. That is, a partition could be cached in any available slot of the partition cache. When adding a partition to a cache that was already full, the least-recently-used (LRU) measure was used to determine which partition should be overwritten.

Cache efficiency was tested as function of the capacity of the partition cache. The capacity is expressed as a percentage of the total number of partitions. So, for example, if a problem used 900 partitions, and the cache capacity was 90 partitions, a cache size of "10%" would be reported. Cache efficiency was measured in terms of the hit rate, which is computed as the number of cache hits, divided by the total

[1] Source is available at http://aml.cs.byu.edu/code/

number of hits and misses.

Chapter 10 also reports a term called the "number of recomputes per partition." This number is computed as the average number of cache misses per partition, and can be thought of as the number of times a partition would need to have been recomputed.

Chapter 10

Results

To facilitate the exposition of the results, the charts and graphs describing the results of our experiments are collected in Chapter 12. This chapter summarizes, discusses and analyzes the results. Conclusions based on these results are drawn in Chapter 11. Section 10.1 discusses the results of experiments with priority metrics and voting, Section 10.2 discusses the results of experiments with parallelization, and Section 10.3 discusses the results of on-demand data.

10.1 Priority Metric and Voting Results

Our experiments designed to quantify the merits of priority metrics and voting generated many positive results. First, P-EVA always demonstrated better time to convergence than VI, while maintaining accuracy. Second, P-EVA scaled extremely well, both with the number of states and the dimensionality of the underlying problem used to generate the MDP. Third, the algorithm never processed certain unreachable partitions, which is a boon from the standpoint of efficiency. This result also motivated one of the most significant contributions of the thesis, which is the analysis

of "sufficient subsets" in Section 8.5. Fourth, we noted that the additional space requirements scale linearly with the size of the problem. However, several issues were encountered: first, it is difficult to tune the parameters of the algorithm, particularly the number and configuration of partitions. Second, we observed mixed effectiveness of the various enhancements.

10.1.1 Positive Results

Figures 12.1, 12.2, 12.3, 12.4, 12.5, and 12.6 all show that P-EVA clearly outperformed normal VI (although for different reasons in different cases). To solve a 160,000 state version of MCAR, for example, value iteration required about 17 seconds, but P-EVA-H2 with voting required only about half a second. For a 160,000 state version of SAP, VI required 19 seconds, but P-EVA-H2 with voting required only about 0.3 seconds. For a 3.7 million state version of DAP, P-EVA-H1 required only 7 seconds; normal VI required 90 seconds. In fact, the only situations in which VI outperformed P-EVA were in some hand-tuned problems (designed as counter-examples to the hypothesis that P-EVA always outperformed VI), but never in our "real-world" problems. It is also very interesting to note that P-EVA solved both MCAR and SAP in about the same amount of time for any given discretization, even though both represented very different problems.

Voting was almost always very effective. In most experiments, voting reduced time and backups by at least a factor of 2, and even in the cases when it did not help, we never saw a situation where it hurt in any statistically significant way. This is a very positive result, considering how trivial it is to implement. We hypothesize that more sophisticated intra-partition backup orders will increase performance even more.

P-EVA appears to scale extremely well: for a constant outdegree, it appears to scale linearly with the number of states. It also appeared to scale relatively well with dimension. This is surprising considering how naïve our partitioning method is: conceptually, a hypercube-based partition seems to be a losing proposition as dimensionality increases, because the surface-area-to-volume ratio of the partition means that one would expect a prohibitive number of cross-partition transitions. This appears to have been outweighed by the fact that partitioning anything at all helps dramatically.

P-EVA never processed certain states in the MCAR problem. Figure 12.20 demonstrates this graphically: large swathes of the state space (indicated by an almost-black color) were never processed, because the agent can never reach the goal state from them. This is a significant result from a practical standpoint. No additional code or information about the problem was necessary, but a full 19% of the state space was never processed. This behavior manifested itself in the algorithm through partitions with a priority of zero that never changed.

To ensure that the policies resulting from P-EVAwere valid, a 75,000,000 state version of the double-arm pendulum was run (an empirically determined minimum resolution needed for a decent control policy). A very good result is that P-EVA solved it (to $\epsilon = 0.0001$) only about four hours. The resulting control policy performed perfectly, and represented the first time we solved DAP using any algorithm. We should note that this problem was run on an SGI Origin 3000 (a 64-bit machine was needed to address the 8G of RAM required by the code); the SGI was about half as fast as the P4 used in all of the other experiments, and is a shared machine which is always heavily used.

The space complexity of P-EVA is also quite good. The largest overhead comes from the need to store a partial inverse model (representing the cross-partition transi-

tions), but this is always a subset of the whole problem. Additional memory is needed for the priority queue ($O(|S|)$), for the state-to-partition mapping ($O(|S|)$), and the partition-to-state maps ($O(|P|)$).

10.1.2 Negative Results

In order to obtain the best results, partition sizes had to be selected manually (however, it is also possible to present this as a positive result. The fact that the system was fast enough to allow us to tune this parameter is significant). Figures 12.18 and 12.19 demonstrate that partitioning is largely problem-dependent: for MCAR and SAP, adding any partitions at all dramatically improved performance, but adding too many worsened it again. For DAP, Figure 12.19 demonstrates that using only two partitions actually worsened performance, but that using more improved it again. The configuration yielding the fewest number of backups for MCAR and SAP was somewhere around 1200 partitions. We do not know how to predict this number, except to observe that using somewhere between 100 and 400 states per partition tended to yield very good results in all of the problems we tested.

It is clear that partitioning almost always helps, that voting almost always helps, and that the using the *H2* priority metric almost always yields better performance than using the *H1* metric. However, DAP represented an exception to all three. In the setup shown in Figure 12.19, using 16 partitions (only two partitions per dimension) caused P-EVA to perform worse than normal VI. Fortunately, once the number of partitions was increased, P-EVA began to perform better. From Figures 12.5 and 12.6, it appeared that both voting and the *H2* metric worsened performance; the largest performance gain here seems to be due to the partitions themselves. We also note from Figures 12.5 and 12.6 that, for a small number of states, normal VI slightly

outperformed any variant of our algorithm. Because each state always depends on $d+1$ other states, this is presumably because the problem was closer to being fully connected.

Related to the problem of selecting an appropriate partitioning is the jagged nature of most of the graphs. While the amount of time used needs to be averaged over several runs, the number of backups is deterministic. We hypothesize that as the number of states (or the number of partitions) changed, sets of mutually dependent states that were previously contained in a single partition became split among multiple partitions, which could account for the extra computation needed to drive the partitions to convergence. This is a further indication that a naïve partitioning is sub-optimal.

10.2 Parallelization Results

Experiments designed to quantify the parallelizability of the P-EVA family of algorithms also generated many positive results. Most significantly, Parallel-P-EVA outperformed all other algorithms tested, including P-EVA. Additionally, our theories regarding assignment of partitions to processors were largely validated, and we demonstrated that an obvious parallel implementation of normal VI (the PSVI algorithm) doesn't perform well at all. One mildly negative result is the fact that the parallel efficiency of the Parallel-P-EVA algorithm is not terribly good.

10.2.1 Positive Results

First, it is clear that the Parallel-P-EVA algorithm outperforms all other algorithms. The best run of Parallel-P-EVA solved a 4,000,000 state problem in 2.08 seconds; this can be roughly viewed as a solution rate of 1.92 million states per second. For comparison, the best run of PSVI solved 390,625 states in 7.0 seconds, for a rate of

55,803 states per second. Additional experiments were run with the P-EVA solution engine: its best run solved 360,000 states in 1.2 seconds, for a rate of 300,000 states per second.

Second, it is clear that the parallel implementation is viable, in the sense that it scales relatively well as the number of processors is increased. Figure 12.8 demonstrates that adding more processors always improved performance. The actual efficiency, however, is not terribly good, as explained in the next section.

Figure 12.7 shows the results of experiments with attractors, which largely validated our theories about information flow. As predicted, using only a few attractors (resulting in extremely large, contiguous blocks) did not perform well at all, but increasing the number of attractors almost always improved performance. The benefits quickly diminished however; there was no difference after about 30 attractors per process. The results in Figure 12.7 contain an extremely significant spike at 5 and 6 attractors per processor, which is consistent across all problems. Although the reason for this is unknown, it may serve to illustrate an important point. As with any random assignment method, it is possible to create a pathological distribution which may result in poor performance.

Figure 12.9 indicates that P-EVA scaled superlinearly on the MCAR problem, which deserves some investigation. The canonical explanation for superlinear scaling is *cache coherency*. For certain problems, and for a fixed problem size, increasing the number of processors increases the cache-to-data ratio. To explore this, we computed the average number of times each partition was processed. For this experiment, the partitions in the MCAR problem were processed an average of 9 times each – low enough to exhibit excellent cache behavior. The other possible explanation for this behavior is that concurrently processing partitions sometimes results in a backup order that is more optimal than the update order imposed by the priority metric, but

this theory remains to be validated.

The results also make it clear that a naive parallelization of value iteration performs quite poorly. Figures 12.10 and 12.11 indicate that the algorithm works well for MCAR and SAP, and is even slightly more efficient than Parallel-P-EVA (approaching an average efficiency of 0.55). The results on DAP and TAP, on the other hand, indicate that the parallel version *never* performed better than the serial version. Since the number of cross-processor transitions increases with the dimensionality of the underlying problem, this may imply that communication overhead is prohibitive.

Only a few results involving P-EVA were needed, and are reported here. P-EVA solves the 4,000,000 state / 10,000 partition MCAR problem in 44.2 seconds, the 360,000 state / 900 partition MCAR problem in 1.2 seconds, and the 1,000,000 state / 769 partition TAP problem in 7.5 seconds.

10.2.2 Negative Results

The parallel efficiencies of Parallel-P-EVA are shown in Figures 12.8 and 12.9. The results indicate that although adding more processors always helps, the parallel efficiency converges to about 0.45 for two out of three problems. Other parallel algorithms are considered good if they exhibit parallel efficiencies of about 0.75 or higher. Thus, these efficiencies leave something to be desired. However, on the third problem (MCAR), Parallel-P-EVA scales superlinearly, which is fascinating result. Possible explanations for this behavior are explored in the Chapter 11. Regrettably, this superlinear speedup is not consistent.

10.3 On-Demand Data Results

Experiments designed to test the cache efficiency of the *H1* and *H2* metrics yielded striking results. On the positive side, every experiment indicated that the *H2* metric greatly outperforms the *H1* metric, sometimes by as much as a factor of two. Additionally, reasonable cache efficiencies (above 80%) can be achieved using a cache capacity of only 20%. On the negative side, we note that updating partition dependents is expensive vis-a-vis partition caching, and that the information-frontier-only cache performances could be improved.

10.3.1 Positive Results

All of the experiments related to on-demand cache efficiencies yielded very consistent results. Figures 12.12, 12.13, 12.14, 12.15, 12.16, and 12.17 all show that the *H2* metric *dramatically* outperformed the *H1* metric in terms of cache performance. As an example, assume that we wished to limit our cache capacity to 22% of the total number of partitions, and that we count both information-frontier and auxiliary partition accesses. At this capacity, using the *H2* metric to solve the MCAR problem yields a hit rate of 90.92%. Using the *H1* metric, however, yields a hit rate of 66.91%. Counting only information-frontier partition accesses, the discrepancy is even greater: *H2* yields a 72.99% hit rate, while *H1* yields only 28.64%.

These efficiencies directly affect the number of recomputes per partition. At the same capacity of 22%, and again counting both information-frontier and auxiliary accesses, the *H2* metric would require an average of 3.08 recomputes per partition. The *H1* metric, on the other hand, would require about 22.02 recomputes. For information-frontier accesses only, *H2* would require 2.13 recomputes, while *H1* would require 10.37 recomputes.

This general pattern of results holds for all problems, for cache sizes from 0% capacity to about 80% capacity. After about 80% capacity, the *H1* metric sometimes yielded better cache efficiency, but not by much. At 100% capacity, both algorithms yielded a 100% cache hit ratio, as expected.

We also note that the marginal cache efficiency (the derivative of efficiency with respect to cache size) equals one at a cache capacity of about 22%. This is the point of diminishing returns, where one must add more than one unit of cache capacity to increase the cache hit rate by one unit. However, at 22% capacity, the *H2* metric yields a 90.92% hit rate on MCAR, 84.48% on SAP, and 86.03% on DAP. This is quite good, and implies that problems which are four times larger than available RAM can be efficiently solved.

10.3.2 Negative Results

There are several negative items of note on the "information-frontier-only" series of graphs (Figures 12.15, 12.16, and 12.17). First, we note that the scale of the "number of recomputes per partition" axis is consistently an order of magnitude less than the "number of recomputes per partition" scale for the information-frontier-plus-auxiliary-accesses experiments! This implies that updating the dependents of a partition is an expensive operation vis-a-vis cache efficiency. Although we can still achieve a good *percentage* cache hit rate, the *absolute value* of misses is very high.

We also note that the SAP problem exhibited much lower cache efficiency in the information-frontier-only setting. We hypothesize that this is because the SAP problem effectively has *two* information frontiers (we encourage the reader to go to the on-line appendix to build intuition about this fact). Even so, the *H2* metric still outperformed the *H1* metric.

Finally, we note that, in general, the cache hit rate in the information-frontier-only series of experiments is much lower than is desirable. More sophisticated caching strategies may alleviate this.

Chapter 11

Conclusions and Future Research

Based on our observations, there are several important conclusions which clarify directions for future research. Section 11.1 presents several conclusions related to priority metrics and voting: first, that all of the enhancements we have studied greatly improve performance; second, that more principled methods of all the enhancements are needed; and third, that the bulk of the benefit is derived from partitions and prioritization. Section 11.2 presents conclusions related to parallelization: first, that parallelization is possible, and second, that it maintains the benefits of prioritization. Section 11.3 discusses conclusions regarding on-demand data, the most significant of which is that the *H2* metric should be the priority metric of choice for solving large MDPs. Section 11.4 presents general research possibilities and final conclusions.

11.1 Priority Metrics and Voting

In the quest for an optimal sequence of backups, the gains to be had from prioritized computation are real and compelling, but there is a lack of understanding as to what constitutes optimality and how it can be achieved. A better understanding of why P-

EVA works is needed. More principled approaches to selecting priority metrics, voting systems, and partitioning schemes are essential. Ideally, such principled methods could still be combined in a unified architecture, with the same synergistic benefits we find in P-EVA.

Partitioning with a priority metric seems to be the most important improvement over VI. Even though we observed that (for some problems) our partitioning scheme was sub-optimal, and that our voting scheme was sub-optimal, and that both of our priority metrics are probably sub-optimal, the fact that they were not perfect seemed to make less of a difference than the fact that we partitioned anything at all. This was shown clearly by experiments with DAP: the addition of voting and the specific priority metric used do not affect things proportionately as much as the initial use of partitions.

It is clear that improved performance is possible for algorithms that exploit problem-specific structure, but it is also clear that more theory is needed to guide the development and selection of algorithmic enhancements. The most useful would be problem characteriziations and/or optimality definitions that would indicate which metric, voting scheme and partitioning scheme would be maximally effective. These may include such things as distributional properties of the reward functions, distributional properties of transition matrices, strongly/weakly connected components analyses, etc.

A more principled approach to partitioning is necessary. A good partition should a) ensure that overhead is reduced; b) minimize cross-partition transitions; c) ensure that the partition respects the priority metric; and d) ensure that a maximum number of dependents are contained within each partition. Several methods of partitioning are possible. Perhaps techniques from graph theory, such as a cost-weighted minimum cut algorithm, could be leveraged to determine an optimal (with respect to (d) above)

partitioning scheme. Of course, the number of states per partition does not have to be constant. Some initial experimentation using the METIS package (see, for example, Karypis and Kumar, 1998) to perform a k-way partitioning has proven very effective (see Alpert, 1996, for an excellent disseration on the subject). It may be possible that techniques from state aggregation literature may help. Dean and Givan (1997) describe a "stable cluster" creation technique, for instance, with properties that are desirable for a partition.

Other intriguing possibilities include on-line, variable, hierarchical or dynamic partitioning schemes. For example, the *H2* metric could correctly prioritize meta-partitions, and then the *H1* metric may be the correct metric for the partitions within the meta-partition, and then normal round-robin updating may be the correct update order within the partition. The choice of a single-level partitioning scheme was arbitrary; perhaps a solution is to generate a continuum of partitions.

A more principled approach to voting is necessary. Empirically, it is clear that voting can help if done properly, but a generalization is needed. Abstractly, voting can be considered a simple surrogate for an intra-partition priority metric. In the same way that partitioning a problem alleviates sub-optimal backups, partitioning a partition would increase its efficiency. A related observation is that voting fails for problems with very consistent transitions. For any given piece of a maze, for example, there may be an equal number of transitions to the north, to the south, and to the east or west. This indicates that optimal backup orders should be dependent upon where information enters the partition, and upon how it flows through the partition. This strengthens the idea that a continuum of partitions, with priority metrics at each level, may constitute an optimal solution.

A more principled approach to developing priority metrics is needed. *H1* and *H2* have intuitive appeal, but stand isolated from the rest of the system. For example,

the *H1* metric is optimal with respect to error reduction, but not with respect to total backups needed. This makes it a good choice for an algorithm that has a fixed amount of time, but perhaps not such a good choice for an algorithm that has as much time as needed.

There are also several miscellaneous research possibilities. Justifying the *H2* metric, perhaps in a unified prioritization framework, would be a valuable theoretical contribution. Being able to approximate the priorities between partitions (i.e., the HPP function), would have benefits across the board: it would simplify the processing the P-EVA algorithm must perform, it would reduce the inverse model sizes, it would reduce inter-process communication in a parallel setting, and it would improve disk-based cache performance by lowering the total number of partition recomputes needed.

Finally, a holistic framework is needed. Currently, each method P-EVA uses to improve efficiency exists somewhat independently of the other methods. Although this allows methods to be used independently, one wonders if greater efficiency is possible with a unified, non-decomposable architecture. For example, it is clear that metrics should be selected with respect to optimality definitions, and that partitions should be selected with respect to priorities, but this is currently not the case.

11.2 Parallelization

Several conclusions are possible based on the results discussed in Section 10.2. The strongest conclusion validates the original question posed. We assert that an effective parallel version of the P-EVA algorithm is possible, with or without voting, and with either priority metric, and that such parallelization maintains all of the performance benefits of the original P-EVA algorithm while permitting additional speedup. In

fact, there is preliminary evidence to suggest that the combination of parallelization with P-EVA can be synergistic.

As noted in Section 10.2, however, the efficiency of Parallel-P-EVA is not as good as it could be. That observation, combined with the strange spikes of poor performance in the attractor experiments, strengthens the case for a reliable, principled method of allocating partitions to processors. There are several interesting avenues of potential research along these lines. For example, it seems possible to use inter-partition relationships to approximate the information flow vectors, perhaps using some sort of geodesic distance metric. This may allow an algorithm to allocate partitions such that all processors are equally likely to be on the information frontier at any given time. Dynamic load-balancing of partitions to processors is also a possibility.

Of course, a more principled method of allocating states to partitions is also necessary. This point was made previously, in Section 10.1, although for different reasons. The P-EVA algorithms benefit from minimum cut k-way partitions not only because they further reduce the model inverse size, but also because they limit the number of dependent partitions that must be processed. In a parallel setting, minimum cut partitions also minimize communication overhead between processors. It is important to quantify the benefit of using such minimum cut partitions, as opposed to using our rather simplistic partitioning scheme. It would also be interesting to quantify the impact of different edge-cut criteria.

Finally, as noted, the efficiency of the Parallel-P-EVA algorithm leaves something to be desired. The most interesting idea for future research along these lines involves the ideas of approximate inter-partition priorities. If the HPP function could be approximated, instead of computed exactly, the amount of data that would need to be sent between processors would be dramatically reduced. It is also possible that more queuing theory, or a more sophisticated batching mechanism, may improve

efficiency and performance even further.

11.3 On-Demand Data

The results of our on-demand data experiments make several strong conclusions possible. The first conclusion is based on the striking cache performance of the *H2* priority metric, especially when compared to the *H1* metric. We believe that the performance benefits of the *H2* metric, combined with its cache efficiency, justify *H2* as the priority metric of choice for solving very large MDPs – especially those that will not fit into RAM. The only scenario in which *H1* outperformed *H2* was in the DAP experiments. In these experiments, *H1* usually required about half the time that *H2* required, but *H2* required about half of the recomputes per partition that *H1* required. In this case, overall performance may be a wash, depending on the expense of a cache miss. For a general problem, however, whose characteristics are not known beforehand, it makes sense to select the safest option, which appears to be *H2*. Of course, our experimental domain is fairly limited. Experimentation on more problems (ideally MDPs that were not derived from minimum-time optimal control problems) is necessary to truly justify this claim.

As mentioned in the other sections of this chapter, the idea of approximate inter-partition priorities could have profound benefits. The benefit is most clearly seen with respect to on-demand data, however. Recall that the number of recomputes necessary increased by an order of magnitude when auxiliary partition accesses were considered. Whether the algorithm accesses disk, or whether it recomputes transitions, each cache miss is expensive.

A final direction for future research is to benchmark different types of caches. As noted in Section 7.2, normal VI can greatly benefit from an intelligent predictive

cache. This is due to the regularity of partition accesses. It seems clear that partition accesses are also quite regular in many of our experiments, although they are regular in a more complicated way. A more sophisticated predictive cache, that tended to predict partition accesses close to the information frontier, might benefit our algorithms.

11.4 Final Conclusions

There are many more general research directions that can be taken from here, as well as several general conclusions that can be made. Our final section outlines a few.

For example, we have noticed that there appear to be two primary characteristics of MDPs that affect performance in different ways: "linearity," and "cyclicity." Our algorithms function best on systems with large swathes of regular, acyclic dependencies, i.e., problems with strong linearity. However, there are fascinating possibilities using more cycle-theoretic approaches. Instead of asymptotically approaching a solution, it may be possible to leverage cycle information to drive loops directly to convergence. Such an algorithm would be truly novel, and may provide substantial performance benefits.

In its most general sense, the algorithms we have presented can be considered *efficient information propagation* algorithms. It is also possible that other forms of information – instead of *value* information – could be propagated throughout a system. It seems reasonable that the insights gained from this thesis could be directly applied to such alternative propagation problems. For example, Munos and Moore (2002) discuss the propagation of *influence* and *variance* throughout an MDP – and they use a value iteration type of technique to do it. Additionally, some initial experimentation indicates that it may be possible to propagate model information directly, using a sort of "transition iteration" algorithm. Initial analysis of such an

algorithm indicates promise.

In fact, one of the most interesting avenues for future research is the idea of generalized convergent iterative systems. There are many problems which use successive approximations to approach a solution, such as linear system solving, finite-element problems in computational fluid dynamics, Bayesian network probability computation, etc. One hypothesis that deserves investigation is that all iterative, convergent linear systems can be thought of as information propagation problems. If so, the results of this thesis may be directly applied.

There are many, many more such ideas. Clearly, the results of this work have opened the doors of several fascinating avenues of research. The many possible directions indicate that there are strong possibilities for even more efficient solution methods in the future, as well as possibilities for broader application and utility. We strongly believe it is possible to solve, in feasible amounts of space and time, problems that are far larger than anything current algorithms are capable of solving. We also believe that there are many innovative applications of this technology to problems in many different domains, which are simply waiting to be found.

The most general conclusions are clear: that the gains to be had from prioritized computation are real and compelling; that such prioritization may be effectively approximated; that prioritized algorithms may be implemented in parallel in an efficient manner; and that such algorithms are not only temporally efficient, but spatially efficient as well.

As a final conclusion, perhaps we can say this: we have seen that value iteration is a very flexible algorithm that is amenable to many different types of improvements. We believe that there are still many, many improvements that can be made, and we hope that this thesis – and future ideas – will revitalize efficient VI as the solution method of choice for solving large MDPs quickly.

Chapter 12

Result Graphs

To facilitate the exposition of the results, all of the charts and graphs have been collected in this chapter. Four main sets of results are presented. In most of the experiments, wall clock time is the main performance metric (lower times are better). Exceptions are noted in the Figure captions.

The first set presents the results of experiments designed to quantify the benefits of prioritization and voting. Experiments were run on SAP, MCAR and DAP using normal VI, normal VI voting, and four variants in the P-EVA family of algorithms.

The second set presents the results of experiments designed to quantify the benefits of parallelization. In this set, the performance of Parallel-P-EVA and PSVI are compared, and the relative parallel efficiencies of the algorithms are shown.

The third set presents results showing the relative cache performance of the *H2* and *H1* priority metrics. Here, the performance criteria of interest is the cache hit ratio.

The fourth set presents miscellaneous results of interest. Some results on tuning partition sizes are presented, as well as some figures supporting the claim that P-EVA may not need to process all partitions in a problem.

Priority Metric and Voting Results

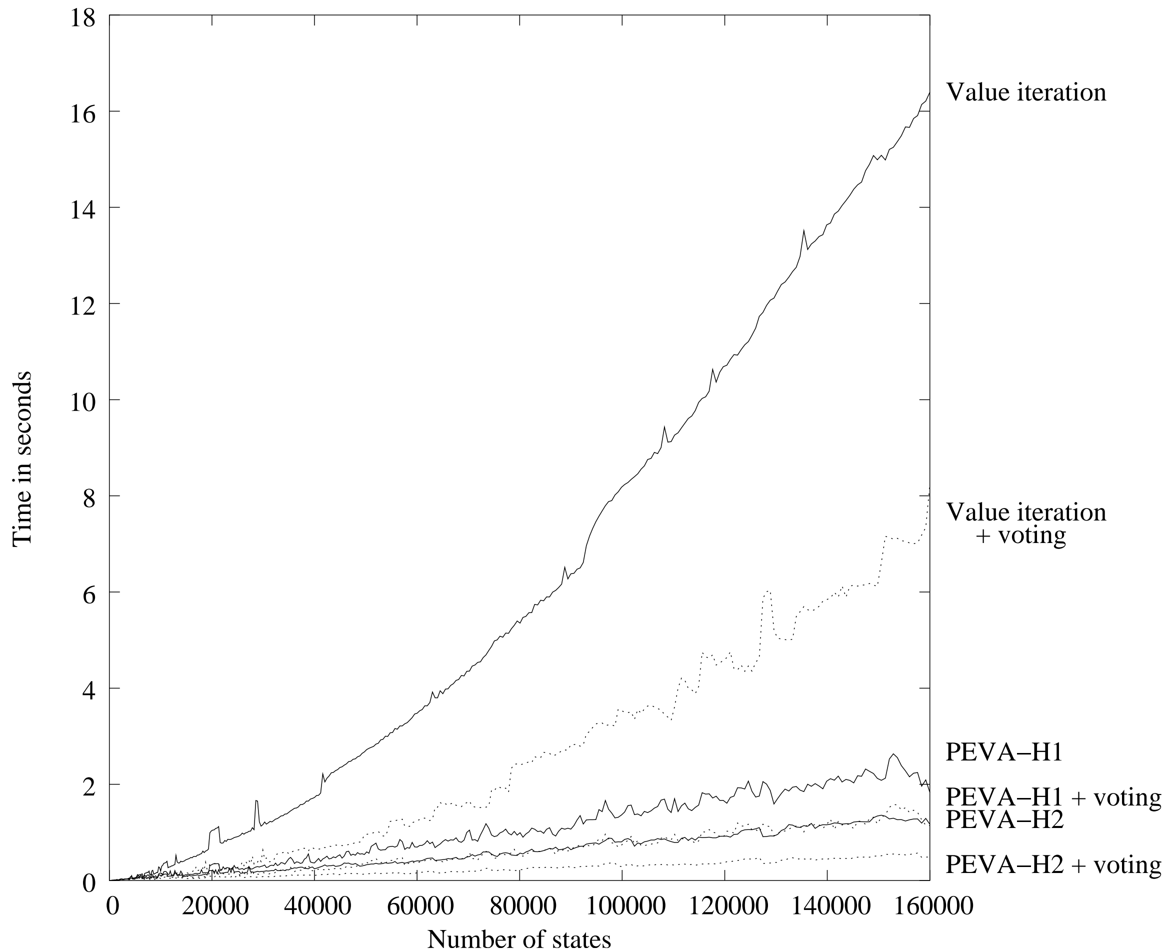

Figure 12.1: Performance results for MCAR in terms of time (less time is better). Voting always improved the results (even for normal VI), and the *H2* metric always performed better than the *H1* metric. ϵ was set at 10^{-4}. For P-EVA, there were always about 400 states per partition. Results were averaged over five runs.

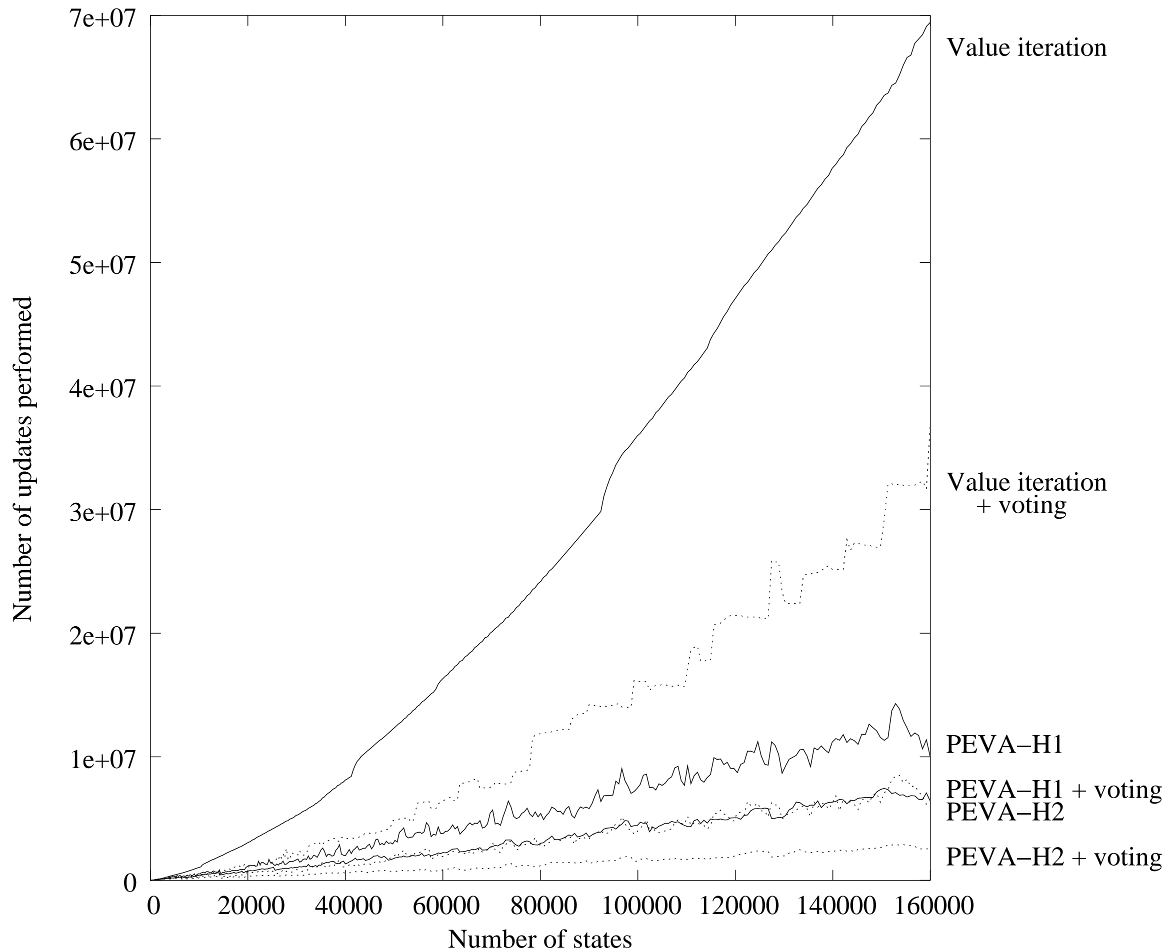

Figure 12.2: Performance results for MCAR in terms of the number of backups performed to the value function (fewer backups is better). Once again, voting always improves performance, and the *H2* metric always improves performance. ϵ was set at 10^{-4}. For P-EVA, there were always about 400 states per partition. Results were averaged over five runs.

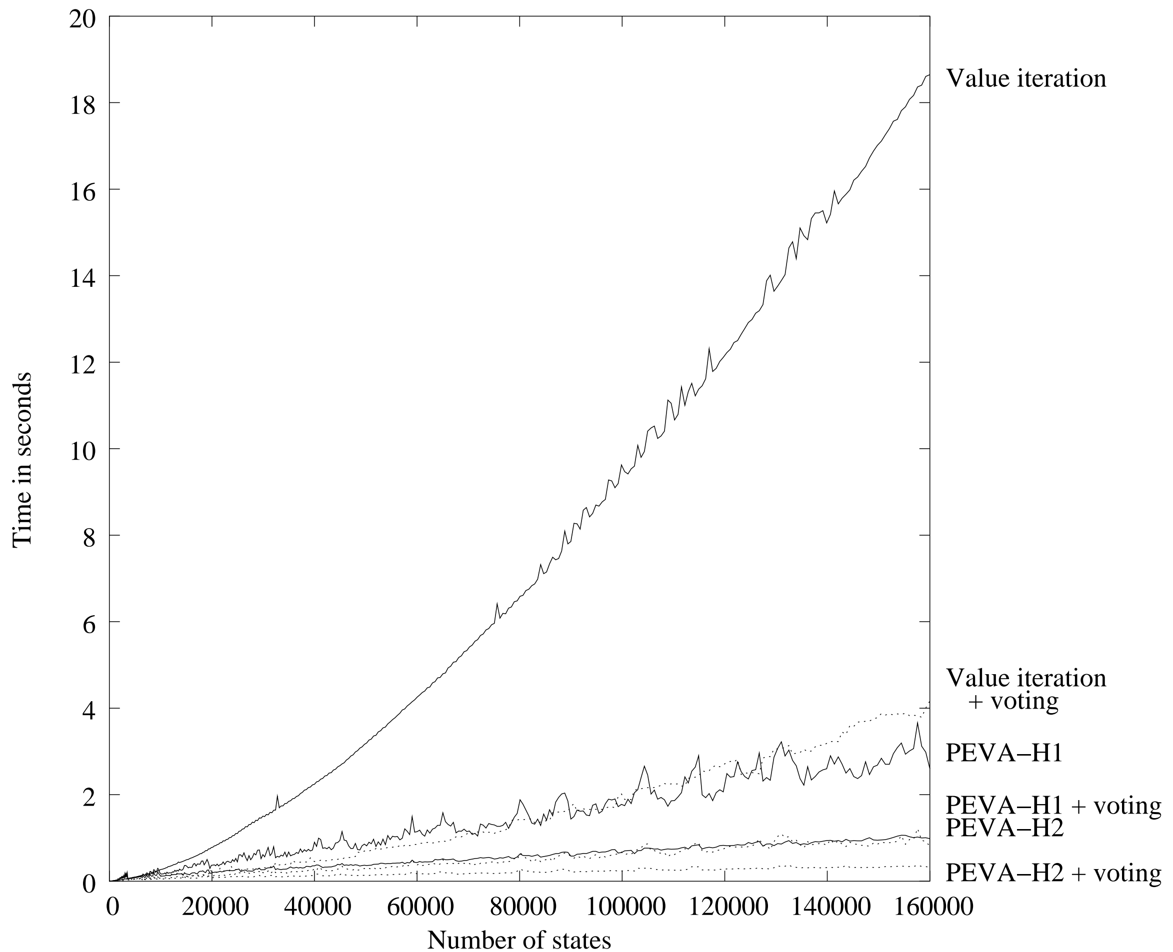

Figure 12.3: Performance results for SAP in terms of time (less time is better). Voting always had a positive impact (and an especially dramatic impact on normal VI). Once again, the *H2* metric outperformed the *H1* metric. ϵ was set at 10^{-4}. For P-EVA, there were always about 400 states per partition. Results were averaged over five runs.

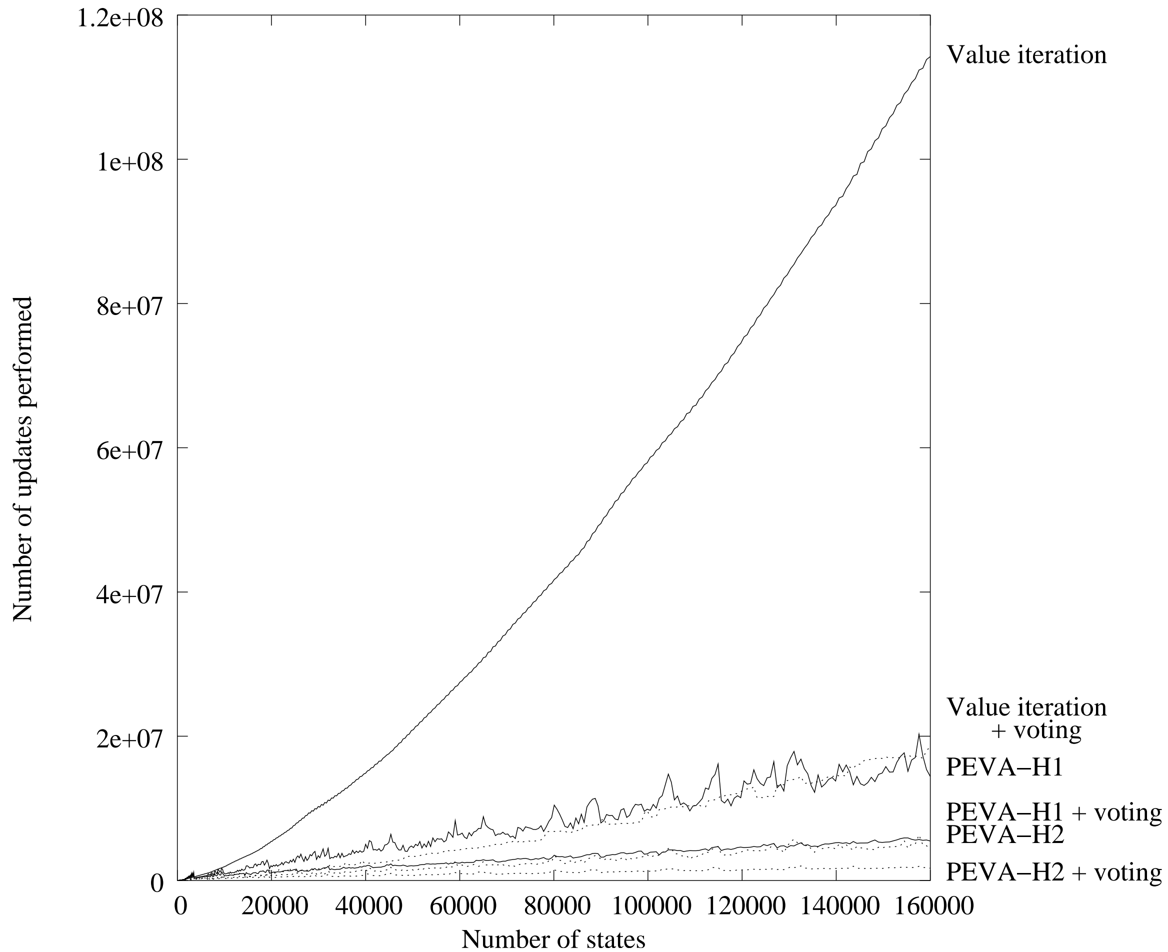

Figure 12.4: Performance results for SAP in terms of the number of back ups performed to the value function (fewer backups is better). These results largely correspond to those in Figure 12.3. ϵ was set at 10^{-4}. For P-EVA, there were always about 400 states per partition. Results were averaged over five runs.

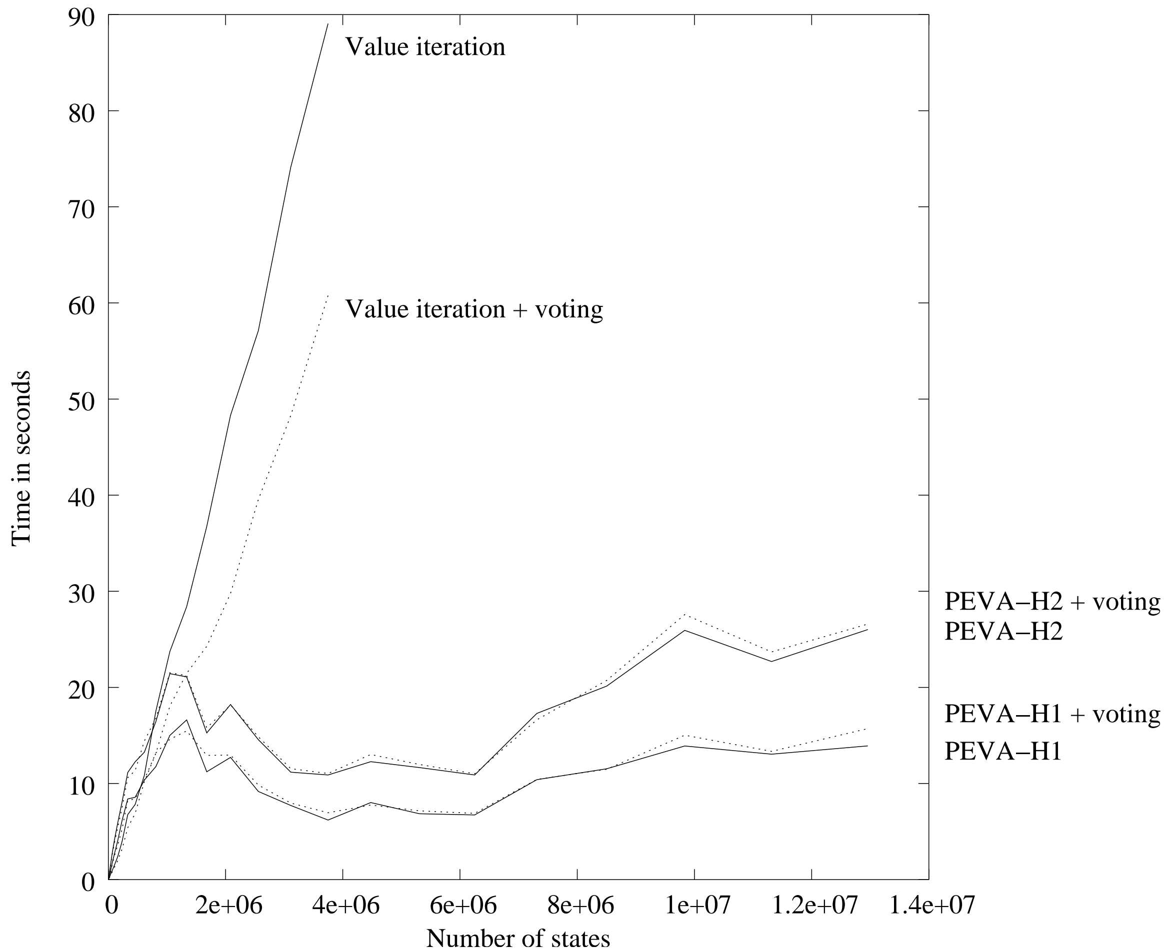

Figure 12.5: Performance results for DAP in terms of time (less time is better). The results contrast sharply with those obtained from MCAR and SAP: voting does not significantly change performance for *H2* or *H1* (slightly reducing backups, but slightly increasing time), and the *H1* metric performs better than the *H2* metric. For a small number of states, both variants of VI perform better than any P-EVA variant. ϵ was set at 10^{-4}. For P-EVA, there were always about 121 states per partition. Results were averaged over five runs.

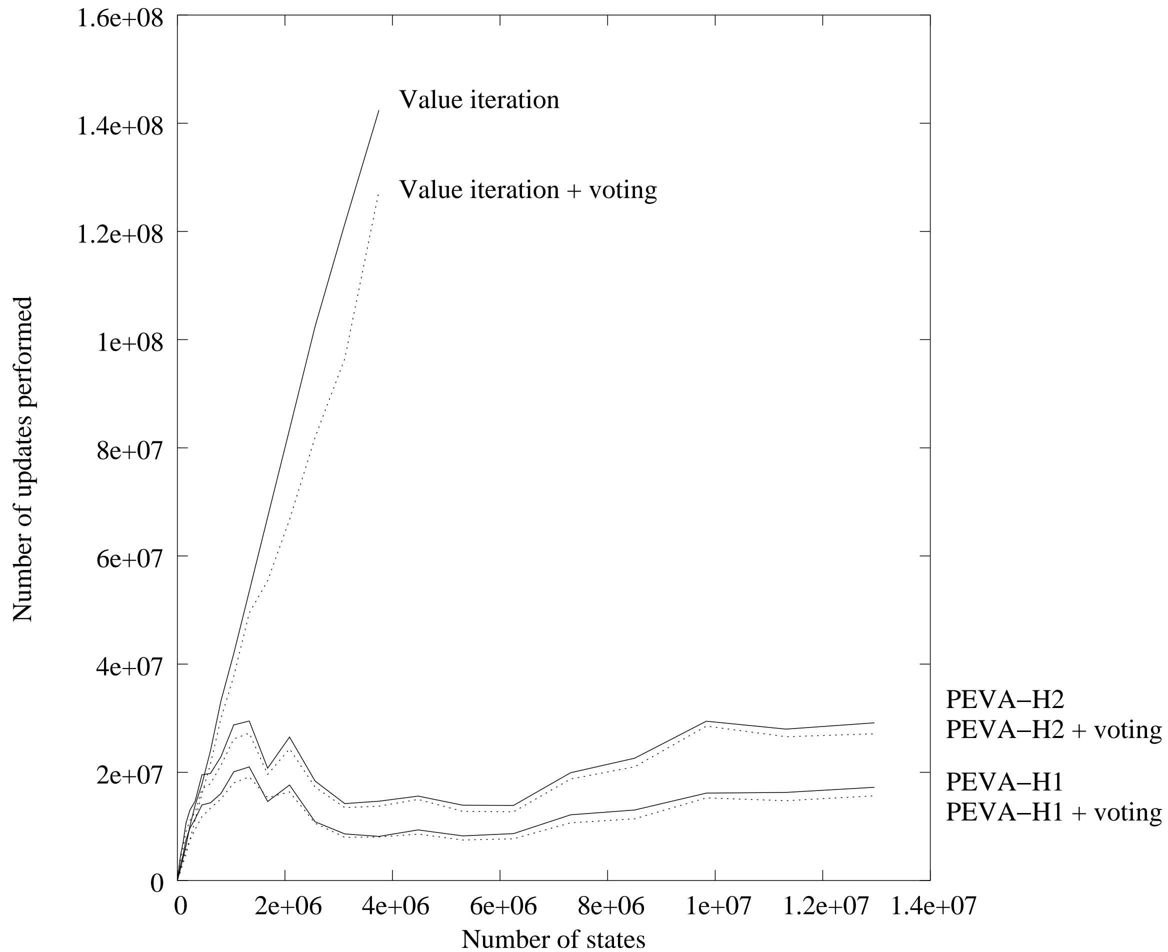

Figure 12.6: Performance results for DAP in terms of the number of back ups performed to the value function (fewer backups is better). These results correspond with those described in Figure 12.5. ϵ was set at 10^{-4}. For P-EVA, there were always about 121 states per partition. Results were averaged over five runs.

Parallelization Results

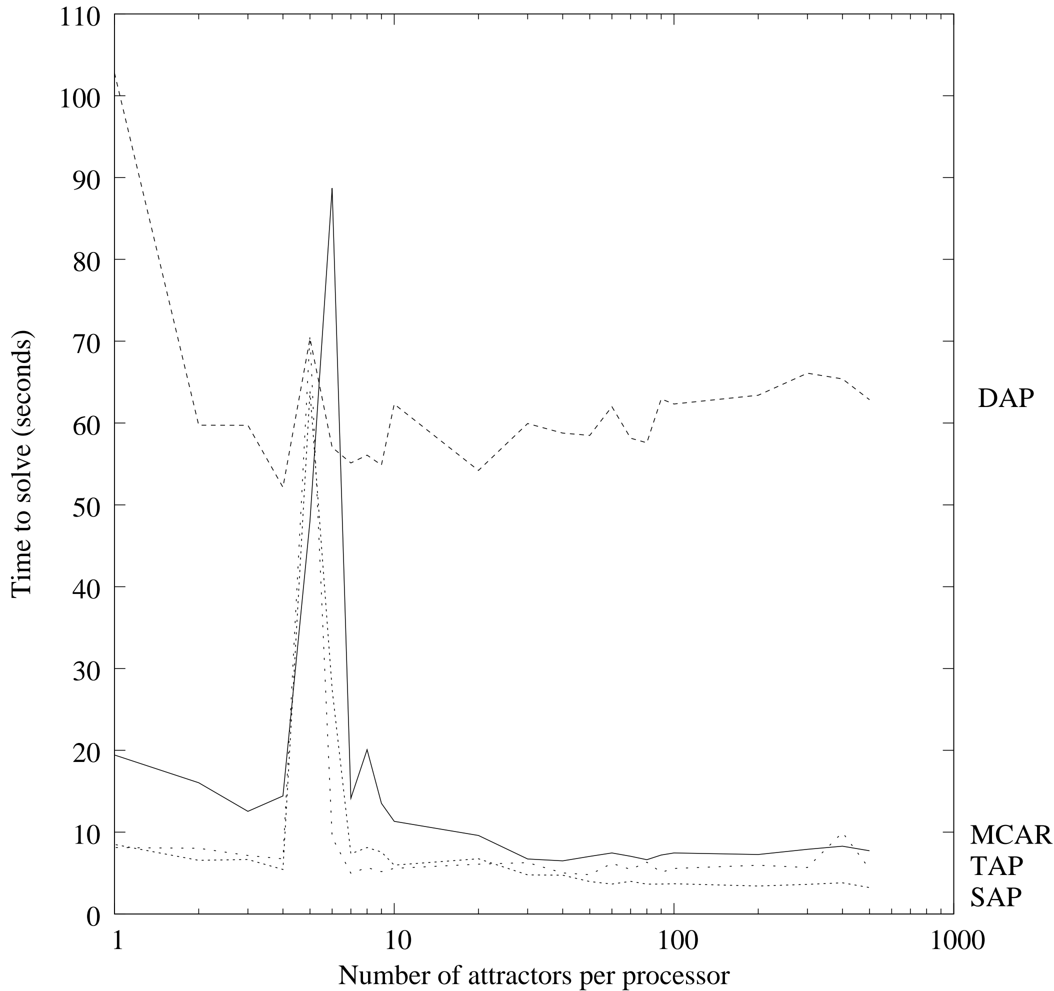

Figure 12.7: Performance (vertical axis) of Parallel-P-EVA as a function of the number of attractors used (horizontal axis). Lower time is better. All runs were on 8 processors. Adding attractors generally improves performance, until about 30-40 attractors. The unusual spikes at 5,6 and 7 attractors appear to be the result of a particularly bad decomposition. The configurations used were MCAR - 4,000,000 states, 10,000 partitions. SAP - 4,000,000 states, 10,000 partitions. DAP - 810,000 states, 10,000 partitions. TAP - 1,000,000 states, 729 partitions.

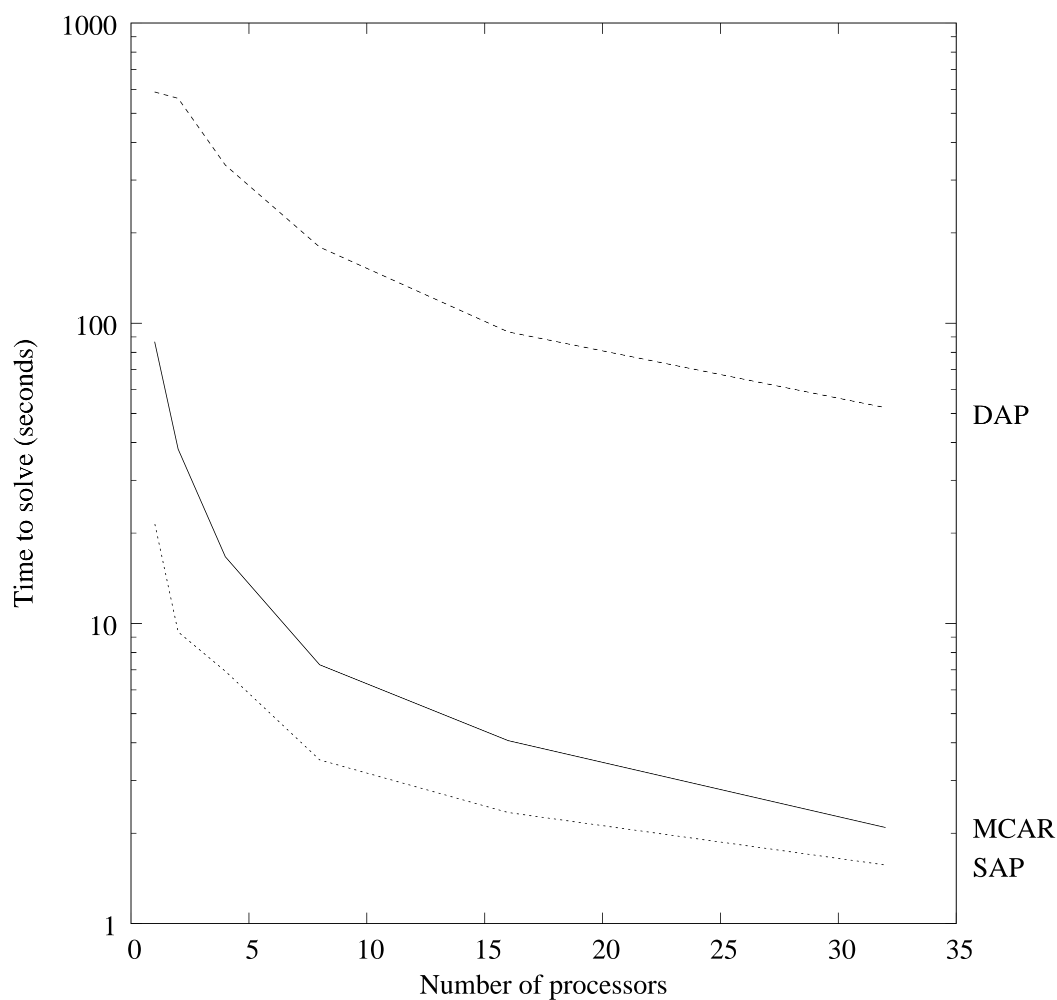

Figure 12.8: Performance (vertical axis) of the Parallel-P-EVA algorithm as a function of number of processors (horizontal axis). Note the log scale. Lower time is better. The configurations used were MCAR - 4,000,000 states, 10,000 partitions; SAP - 4,000,000 states, 10,000 partitions; DAP - 6,250,000 states, 10,000 partitions.

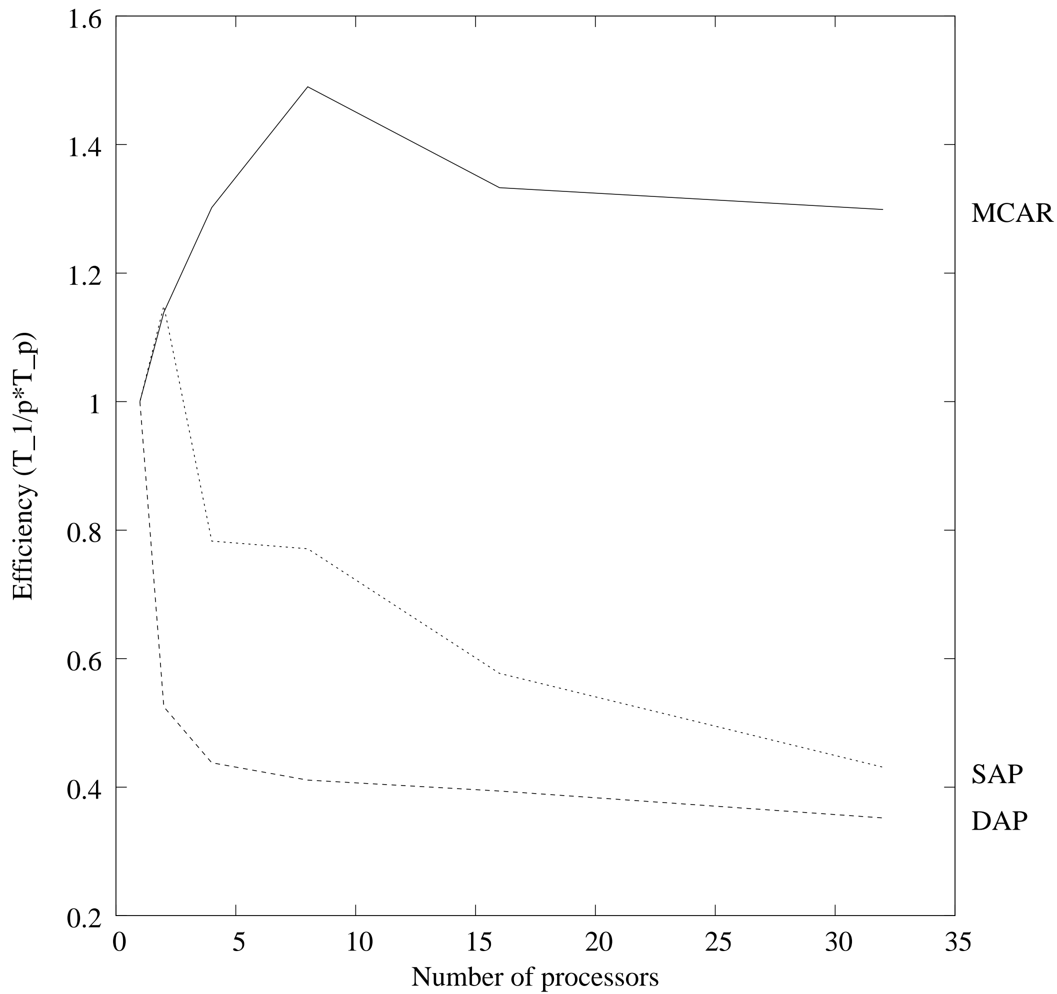

Figure 12.9: Efficiency of Parallel-P-EVA. The same configuration is used as in Figure 12.8. Higher efficiency is better. Note that Parallel-P-EVA achieves superlinear speedup on the MCAR problem, but approaches an efficiency of 0.4 for the other problems.

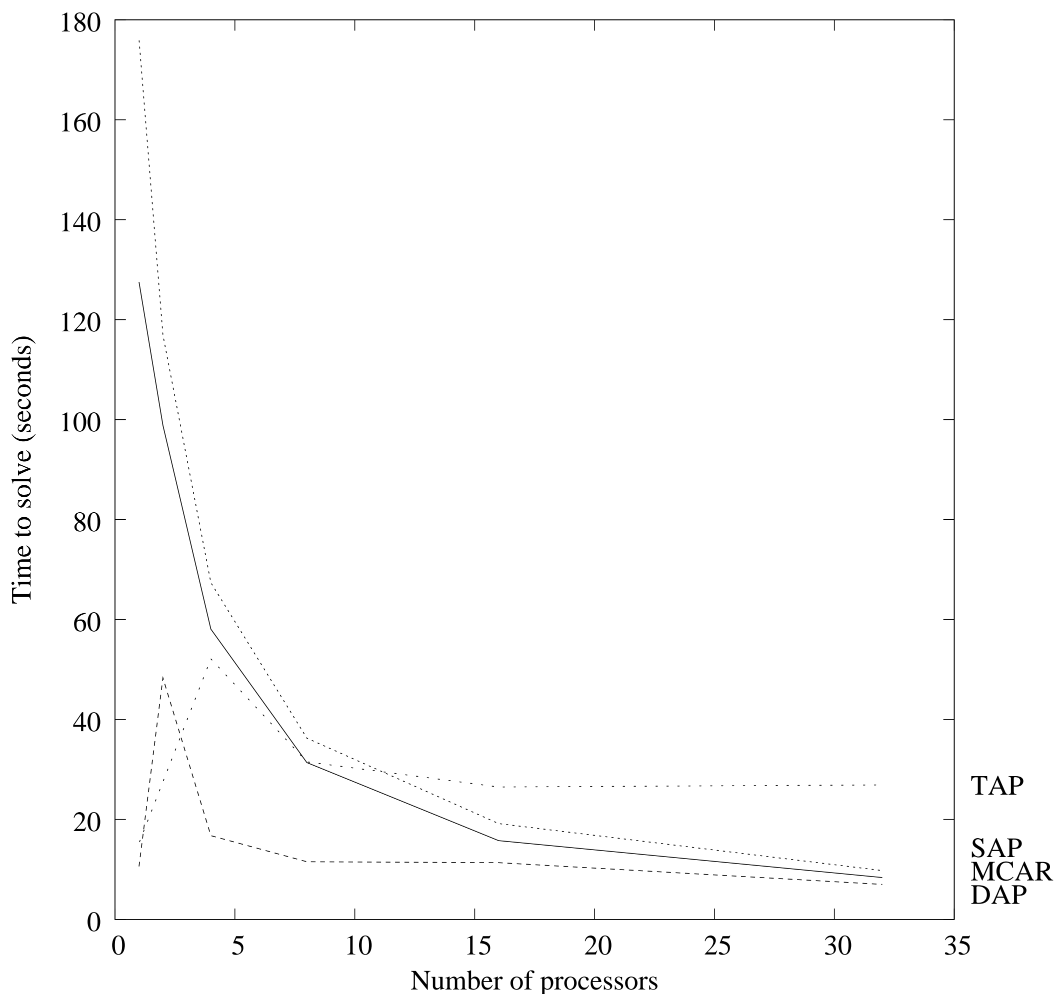

Figure 12.10: Performance of PSVI. Lower time is better. PSVI does quite well on SAP and MCAR, but it always negatively impacts performance on DAP and TAP, no matter how many processors are used. The configurations used were MCAR - 360,000 states, 900 partitions; SAP - 360,000 states, 900 partitions; DAP - 390,625 states, 625 partitions; TAP 1,000,000 states, 729 partitions.

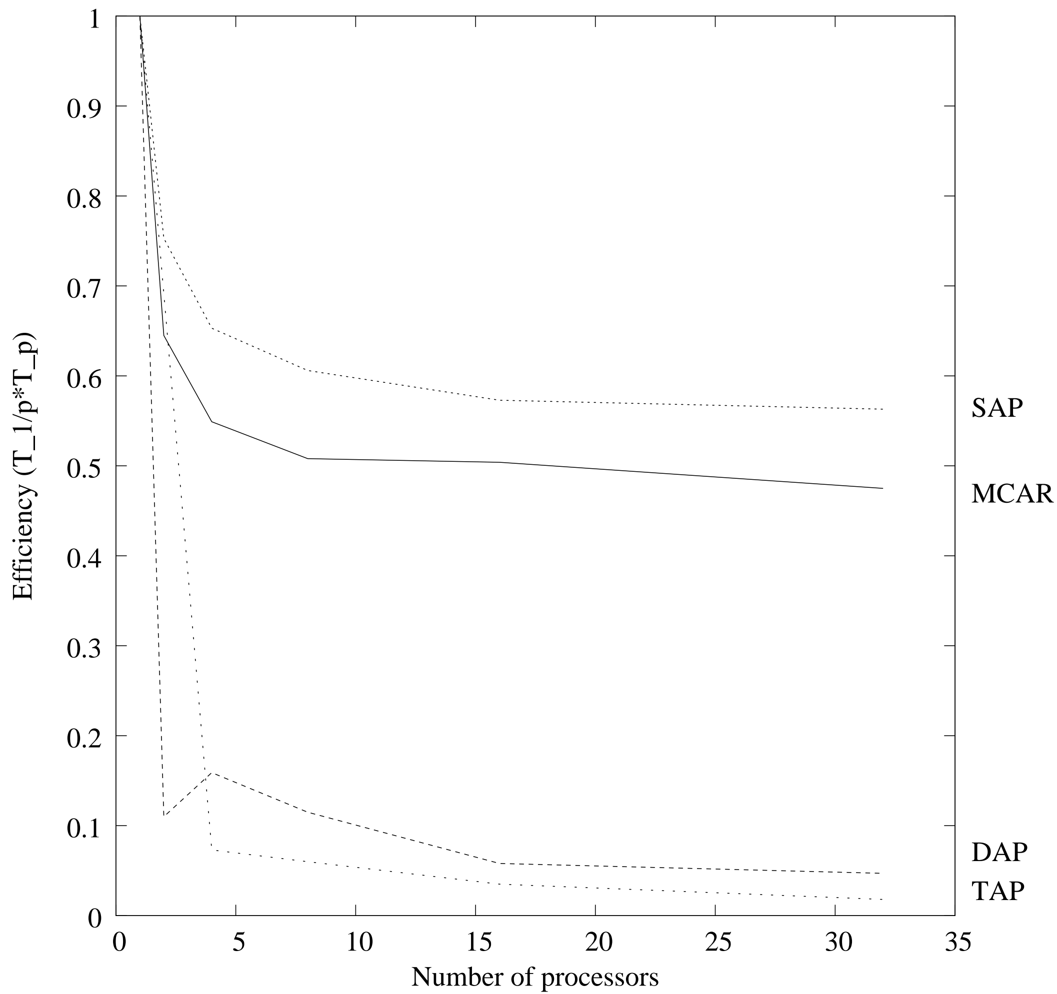

Figure 12.11: Efficiency of PSVI. The same configuration is used as in Figure 12.10. Higher efficiency is better. PSVI approaches an efficiency 0.6 and 0.5 for SAP and MCAR, which rivals Parallel-P-EVA. However, its efficiency on DAP and TAP approaches 0.05, which is unacceptably low.

On-Demand Cache Results

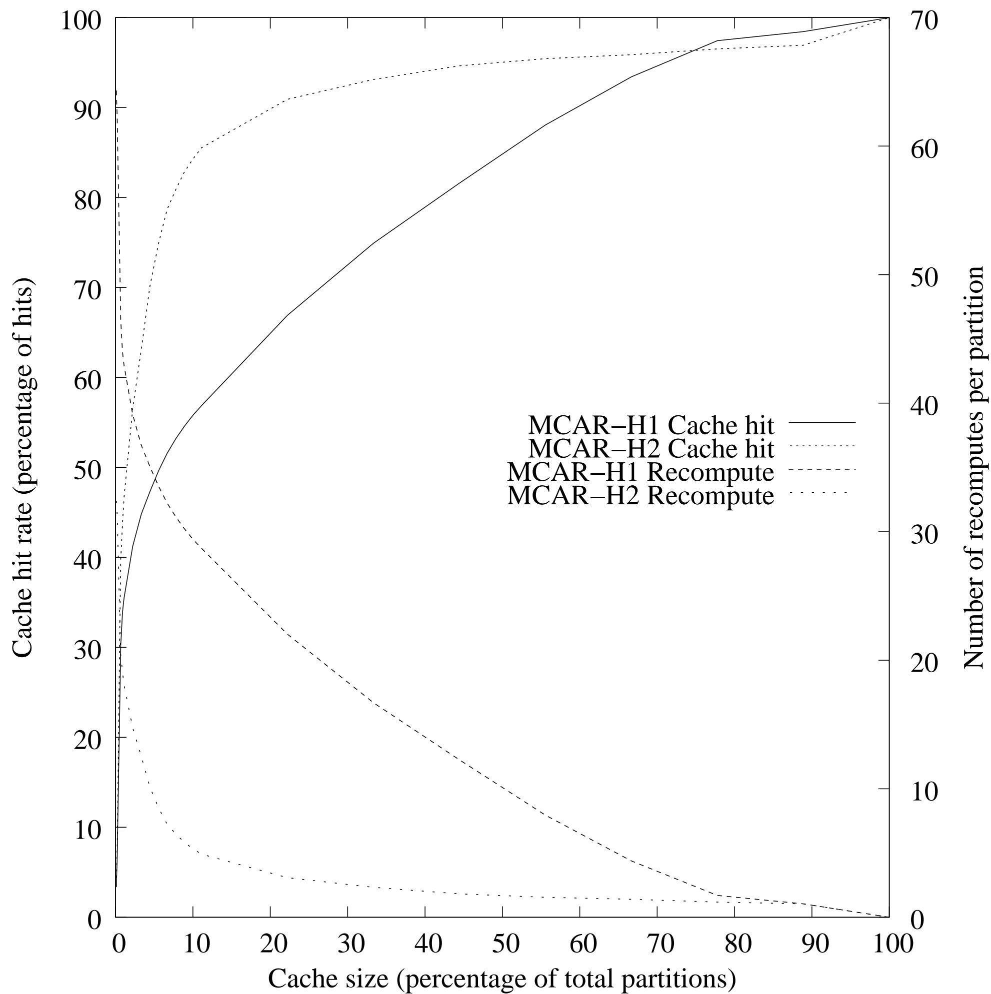

Figure 12.12: On the left axis, cache performance for the *H1* and *H2* metrics on the MCAR problem. On the right axis, the average number of times a partition is recomputed. A higher cache hit ratio is better, but a lower number of recomputes is better. For this problem, a 300x300 state discretization was used, and a 30x30 partition discretization was used.

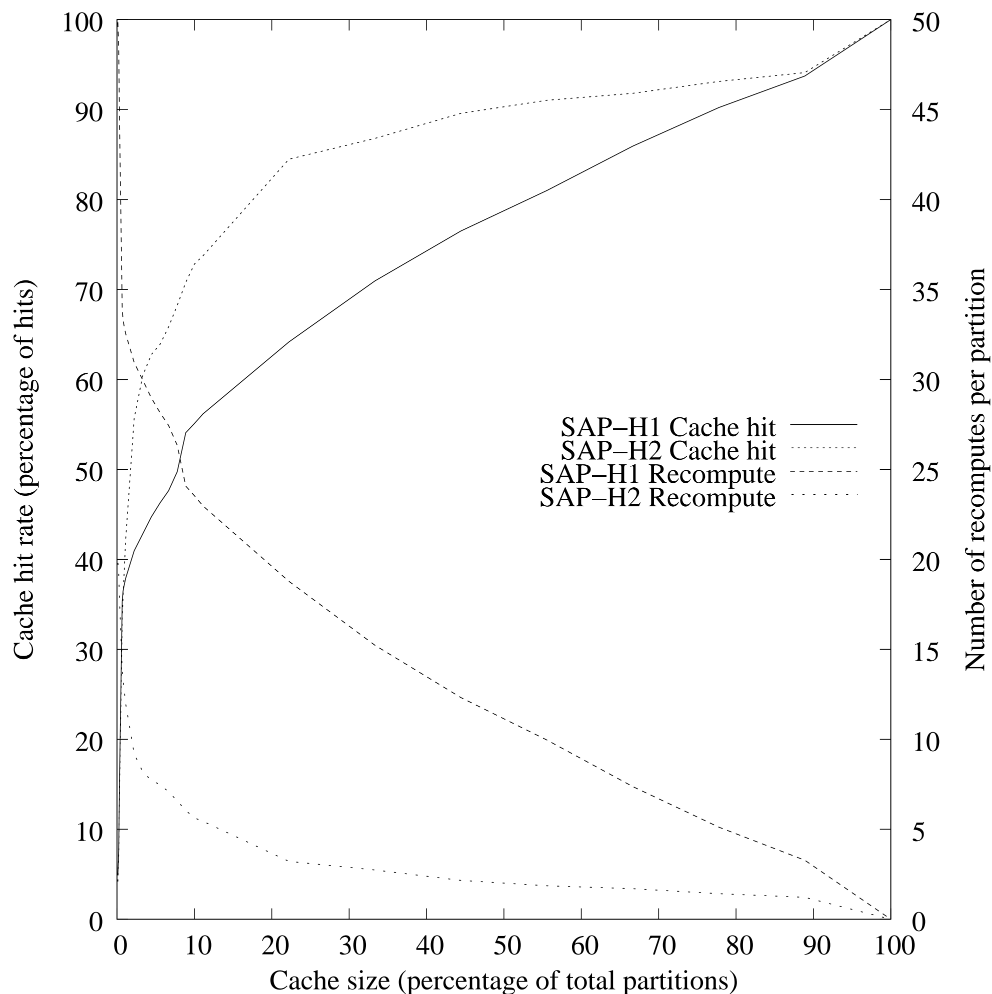

Figure 12.13: On the left axis, cache performance for the *H1* and *H2* metrics on the SAP problem. On the right axis, the average number of times a partition is recomputed. A higher cache hit ratio is better, but a lower number of recomputes is better. For this problem, a 300x300 state discretization was used, and a 30x30 partition discretization was used.

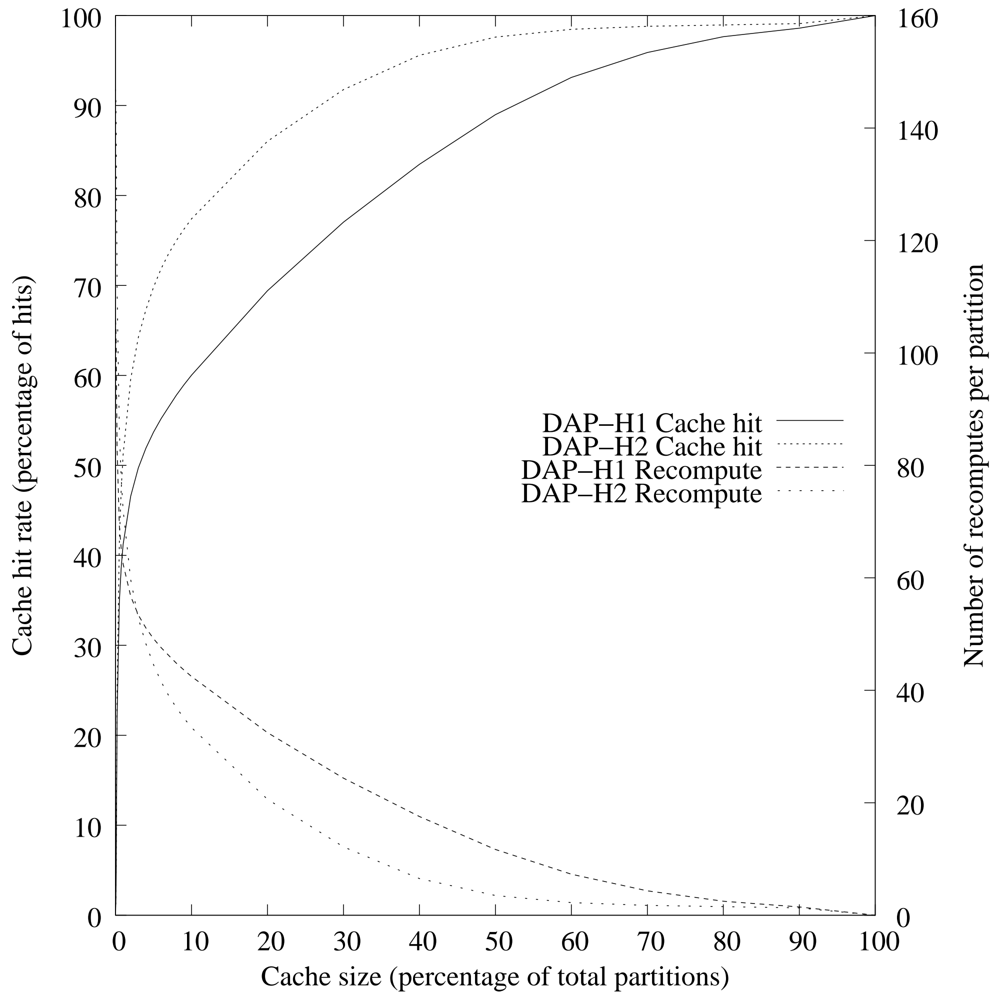

Figure 12.14: On the left axis, cache performance for the *H1* and *H2* metrics on the DAP problem. On the right axis, the average number of times a partition is recomputed. A higher cache hit ratio is better, but a lower number of recomputes is better. For this problem, a 30x30x30x30 state discretization was used, and a 10x10x10x10 partition discretization was used.

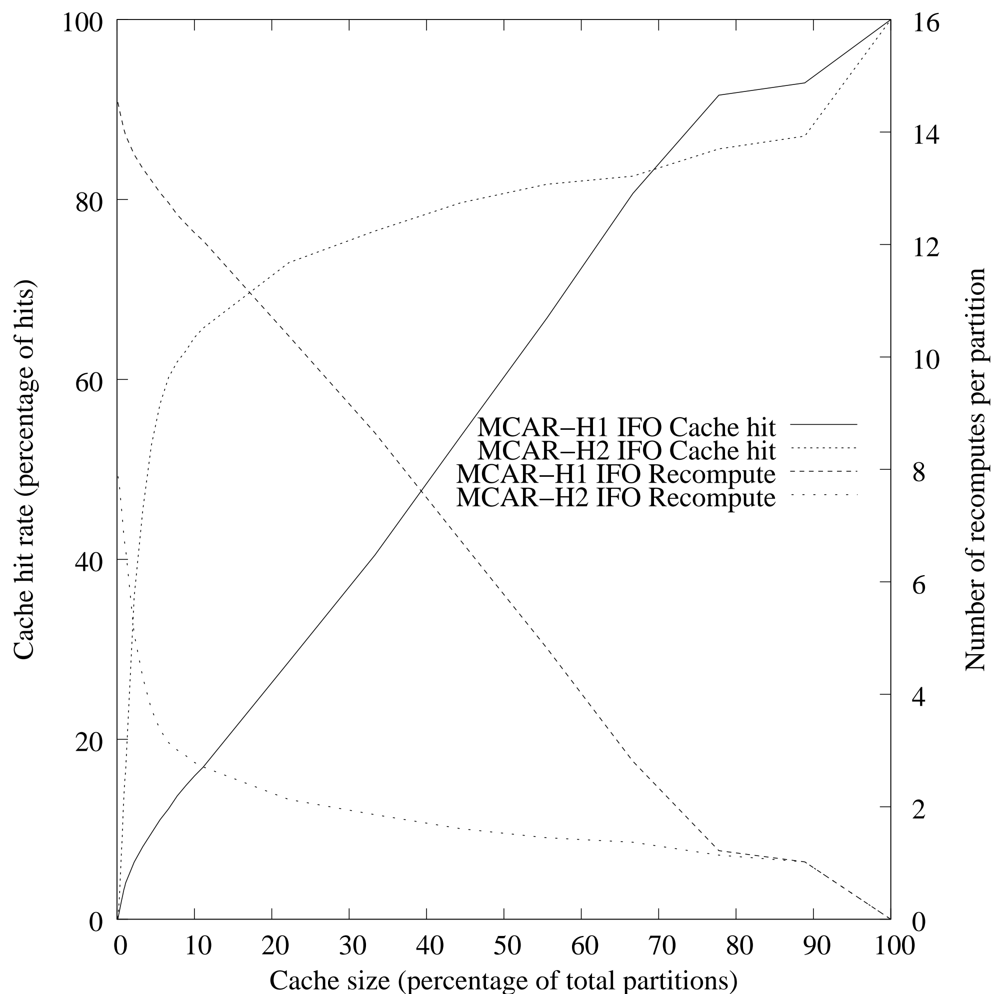

Figure 12.15: On the left axis, information-frontier-only cache performance for the *H1* and *H2* metrics on the MCAR problem. On the right axis, the average number of times a partition is recomputed. A higher cache hit ratio is better, but a lower number of recomputes is better. For this problem, a 300x300 state discretization was used, and a 30x30 partition discretization was used.

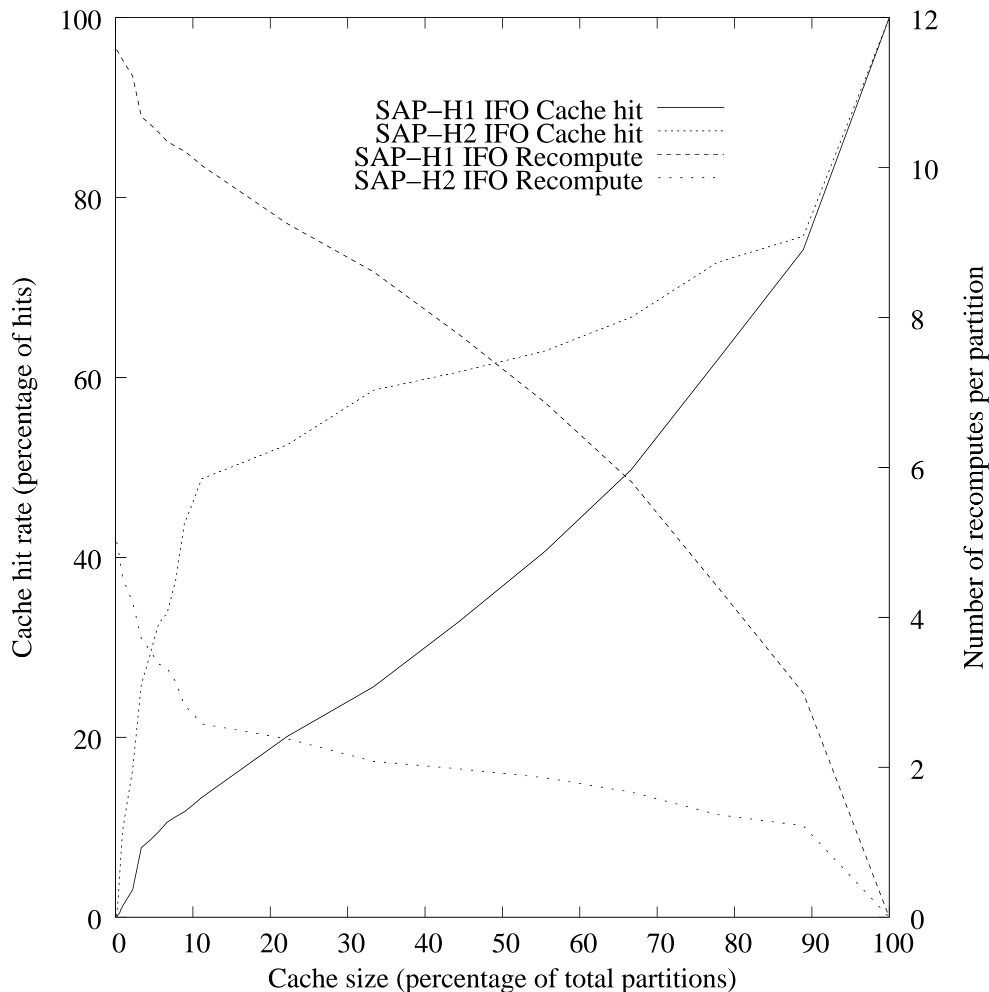

Figure 12.16: On the left axis, information-frontier-only cache performance for the *H1* and *H2* metrics on the SAP problem. On the right axis, the average number of times a partition is recomputed. A higher cache hit ratio is better, but a lower number of recomputes is better. For this problem, a 300x300 state discretization was used, and a 30x30 partition discretization was used.

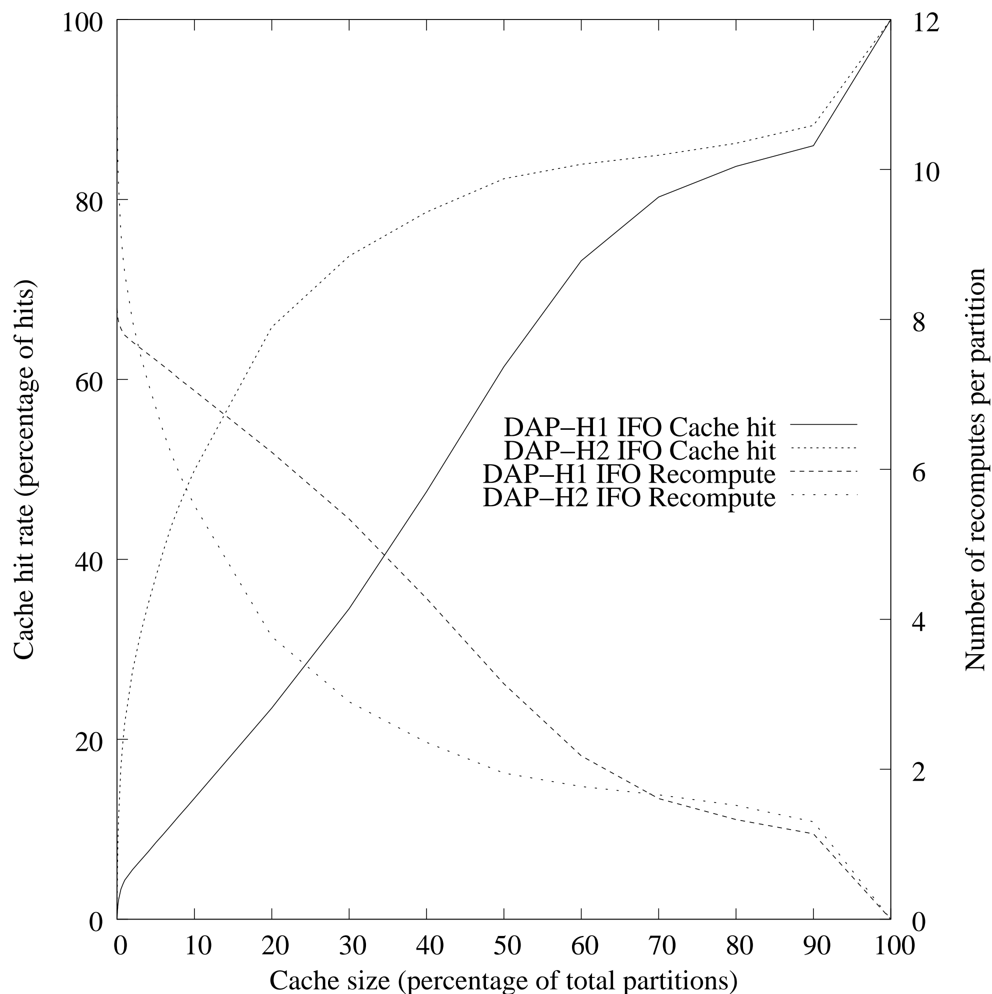

Figure 12.17: On the left axis, information-frontier-only cache performance for the *H1* and *H2* metrics on the DAP problem. On the right axis, the average number of times a partition is recomputed. A higher cache hit ratio is better, but a lower number of recomputes is better. For this problem, a 30x30x30x30 state discretization was used, and a 10x10x10x10 partition discretization was used.

Additional Results of Interest

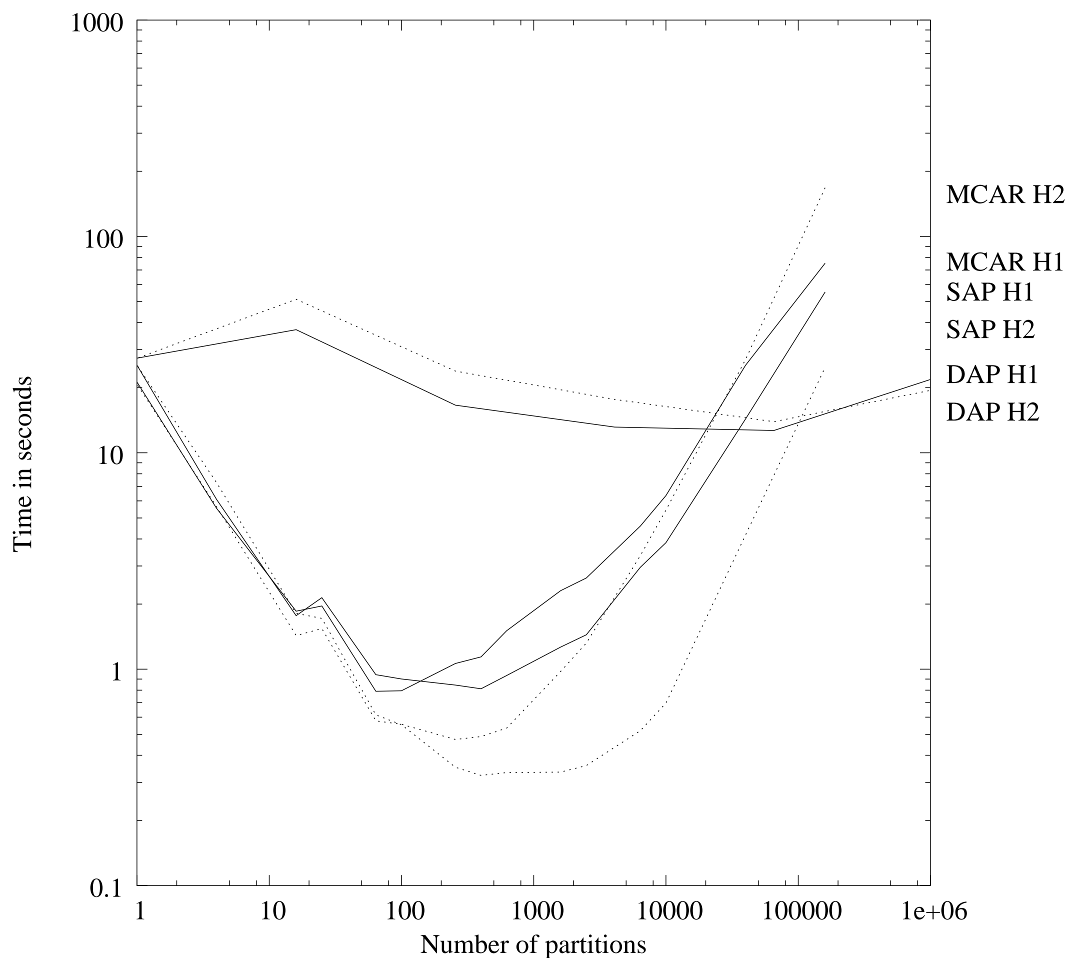

Figure 12.18: Performance in terms of time as a function of the number of partitions used. For MCAR and SAP, a constant 400x400 state grid was used; for DAP, a constant 32x32x32x32 state grid was used. At the far right, there is one state per partition; at the far left, there is only one partition encompassing the entire problem (which is equivalent to normal VI). Log scales were used for clarity, and especially to emphasize the fact that, for SAP and MCAR, using one state per partition yields worse performance than not using any partitions at all. Results were averaged over five runs.

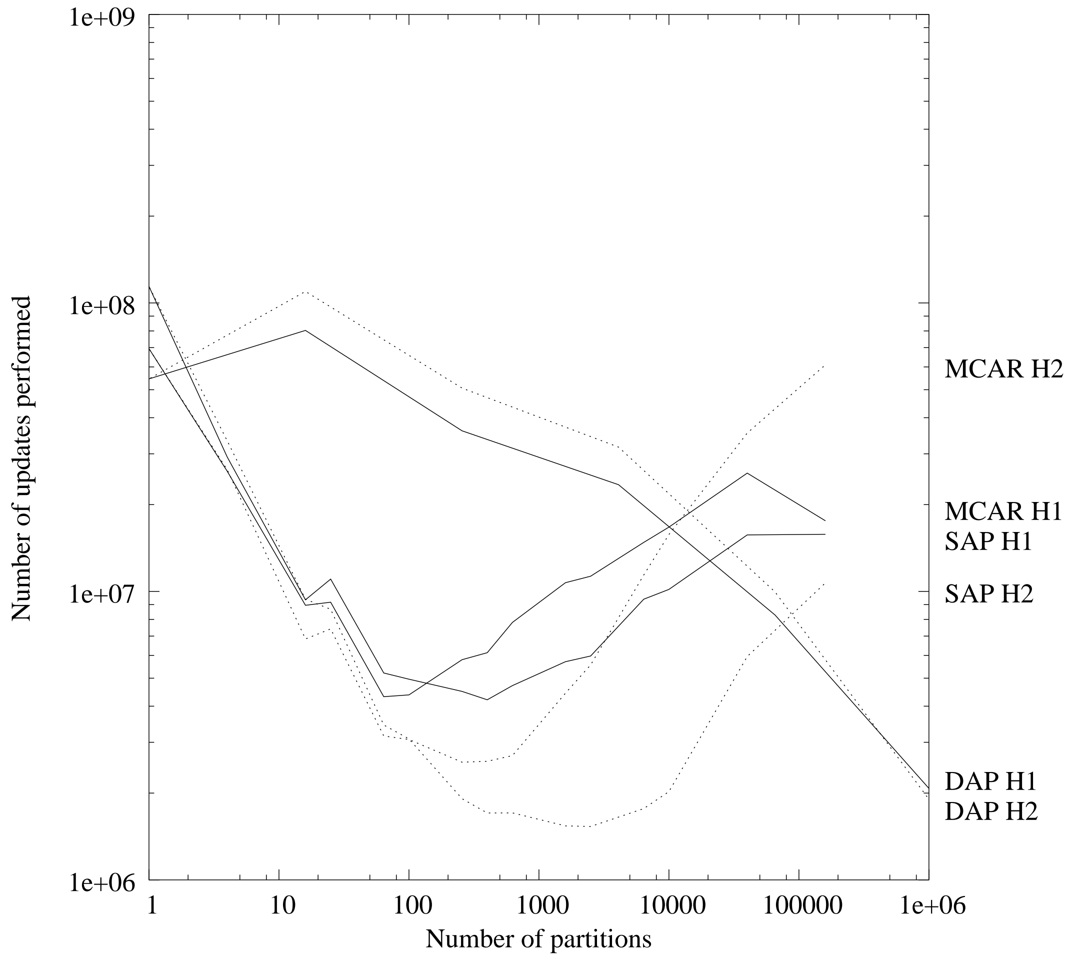

Figure 12.19: Performance in terms of backups as a function of the number of partitions used. The setup corresponds to that of Figure 12.18. Note how dramatically the number of backups needed to solve DAP drops as the number of partitions is increased.

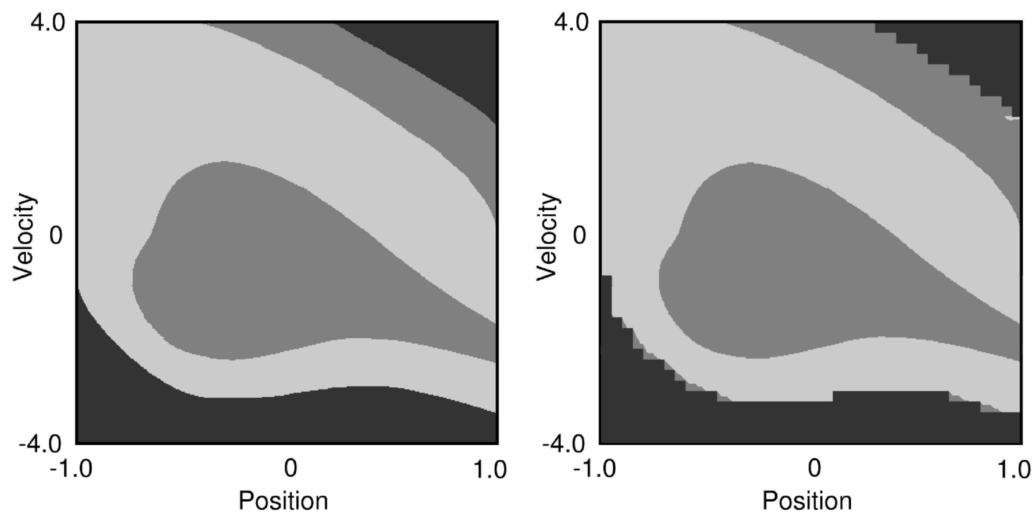

Figure 12.20: The MCAR control policy. Light grey is positive thrust, medium gray is negative thrust, and dark grey indicates that the partition was never processed. The left figure shows that, using one state per partition, the resolution of the unvisited states is very high, and corresponds exactly to the discontinuities in the value function. The right figure shows that, using about 100 states per partition, a partition can be unvisited even if some of the states inside would have been visited. To generate this figure, the traditional MCAR reward function was employed.

Appendix A

Additional Multi-Media Materials

The reader is encouraged to refer to

http://aml.cs.byu.edu/papers/solving_mdps/

for additional multi-media materials. Several videos are available which graphically demonstrate the different backup orders imposed by normal VI and the *H2* and *H1* priority metrics, on the MCAR and SAP problems. We also provide a video of the DAP being balanced by a solution produced with P-EVA. The P-EVA and Parallel-P-EVA source codes are also available for download. In addition, a graphical DAP simulator is available, with a control policy that balances the pendulum from any initial position. A graphical TAP simulator is also available.

Bibliography

Charles J. Alpert. *Multi-way graph and hypergraph partitioning*. PhD thesis, University of California Los Angeles, Los Angeles, CA, 1996.

David Andre, Nir Friedman, and Ronald Parr. Generalized prioritized sweeping. *Advances in Neural Information Processing Systems*, 10:1001–1007, 1998.

Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

Dimitri P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27:610–616, 1982.

Dimitri P. Bertsekas. Distributed asynchronous computation of fixed points. *Mathematics Programming*, 27:107–120, 1983.

Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 1995.

David Blackwell. Discrete dynamic programming. *Annals of Statistics*, 33:719–726, 1962.

Thomas Dean and Robert Givan. Model minimization in Markov Decision Processes. In *Proceedings of The Fourteenth National Conference on Artificial Intelligence*, pages 106–111, 1997.

Geoffrey J. Gordon. *Approximate Solutions to Markov Decision Processes.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1999.

Vijaykumar Gullapalli and Andrew G. Barto. Convergence of indirect adaptive asynchronous value iteration algorithms. *Advances in Neural Information Processing Systems*, 6:695–702, 1994.

Sham Kakade. *On the Sample Complexity of Reinforcement Learning.* PhD thesis, University College, London, United Kingdom, 2003.

George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning schemes for irregular graphs. Technical Report 36, University of Minnesota, Minneapolis, MN, 1996.

George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

Michael L. Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Brown University, Providence, RI, 1996.

Michael L. Littman, Thomas L. Dean, and Leslie P. Kaelbling. On the complexity of solving Markov Decision Problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.

Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159–195, 1996.

Yishay Mansour and Satinder P. Singh. On the complexity of policy iteration. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 401–408, 1999.

Christopher K. Monson. *Value Iteration in the Joint Space*. Master's thesis, Brigham Young University, Provo, UT, 2003.

Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.

Remi Munos and Andrew W. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49:291–323, 2002.

Jing Peng and John Williams. Efficient learning and planning within the dyna framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 437–454, 1993.

Jing Peng and Ronald J. Williams. Incremental multi-step Q-learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 226–232, 1994.

Philippe Preux. Propagation of Q-values in tabular TD(lambda). In *Proceedings of the Thirteenth European Conference on Machine Learning*, pages 369–380, 2002.

Martin L. Puterman. *Markov Decision Processes–Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc., New York, NY, 1994.

Martin L. Puterman and Moon C. Shin. Modified policy iteration algorithms for discounted Markov Decision Problems. *Management Science*, 24:1127–1137, 1978.

Stuart I. Reynolds. *Reinforcement Learning with Exploration*. PhD thesis, University of Birmingham, Birmingham, United Kingdom, 2002.

Gavin A. Rummery and Mahesan Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University, Cambridge, United Kingdom, 1994.

Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8:1038–1044, 1996.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

Ronald J. Williams and Leemon C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical Report NU-CCS-93-14, Northeastern University, Boston, MA, 1993.

David Wingate and Kevin D. Seppi. Efficient value iteration using partitioned models. In *Proceedings of the International Conference on Machine Learning and Applications*, pages 53–59, 2003.

David Wingate and Kevin D. Seppi. P3VI: A partitioned, prioritized, parallel value iterator. In *Proceedings of the Twenty-First International Conference on Machine Learning*, to appear 2004.

Nevin L. Zhang, Stephen S. Lee, and Weihong Zhang. A method for speeding up value iteration in partially observable Markov Decision Processes. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 696–703, 1999.

Nevin L. Zhang and Weihong Zhang. Speeding up the convergence of value iteration in partially observable Markov Decision Processes. *Journal of Artificial Intelligence Research*, 14:29–51, 2001.

Weiyu Zhu and Stephen Levinson. PQ-learning: an efficient robot learning method for intelligent behavior acquisition. In *Proceedings of the Seventh International Conference on Intelligent Autonomous Systems*, 2002.