University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2007

FASTER DYNAMIC PROGRAMMING FOR MARKOV DECISION PROCESSES

Peng Dai
University of Kentucky, daipeng@uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Dai, Peng, "FASTER DYNAMIC PROGRAMMING FOR MARKOV DECISION PROCESSES" (2007). *University of Kentucky Master's Theses*. 428.
https://uknowledge.uky.edu/gradschool_theses/428

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

FASTER DYNAMIC PROGRAMMING FOR MARKOV DECISION PROCESSES

Markov decision processes (MDPs) are a general framework used by Artificial Intelligence (AI) researchers to model decision theoretic planning problems. Solving real world MDPs has been a major and challenging research topic in the AI literature. This paper discusses two main groups of approaches in solving MDPs. The first group of approaches combines the strategies of heuristic search and dynamic programming to expedite the convergence process. The second makes use of graphical structures in MDPs to decrease the effort of classic dynamic programming algorithms. Two new algorithms proposed by the author, MBLAO* and TVI, are described here.

KEYWORDS: Decision Theoretic Planning, Markov Decision Process, Dynamic Programming, Heuristic Search, Topological Structure

Peng Dai

July 10, 2007

FASTER DYNAMIC PROGRAMMING FOR MARKOV DECISION PROCESSES

By

Peng Dai

Dr. Judy Goldsmith
Director of Thesis

Dr. Grzegorz W. Wasilkowski
Director of Graduate Studies

29 June 2007

RULES FOR THE USE OF THESES

Unpublished thesis submitted for the Master's and Doctor's degrees and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part requires also the consent of the Dean of the Graduate School of the University of Kentucky.

A library which borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date

THESIS

Peng Dai

The Graduate School

University of Kentucky

2007

FASTER DYNAMIC PROGRAMMING FOR MARKOV DECISION PROCESSES

THESIS

A thesis submitted in partial fulfillment of the
requirements of the degree of Master of Science in the
College of Engineering at the University of Kentucky

By

Peng Dai

Lexington, Kentucky

Director: Dr. Judy Goldsmith, Department of Computer Science

Lexington, Kentucky

2007

Copyright © Peng Dai 2007

DEDICATION

I dedicate this to my closure at the University of Kentucky.

ACKNOWLEDGMENTS

This thesis is the result of many sources of help. First of all, Dr. Judy Goldsmith, my advisor and thesis chair, who brought me into the world of AI planning, helped me in study and doing research, and supported me all the time, deserves my faithful thanks. I benefited a lot from our whole AI group, especially from Dr. Miroslaw Truszczyński and Dr. Lengning Liu. I also thank Dr. William Dieter and Dr. Andrew Klapper for their useful comments on earlier drafts of this document.

Table of Contents

Acknowledgments . . . iii

List of Tables . . . vi

List of Figures . . . vii

Chapter 1 Introduction . . . 1

Chaper 2 Search . . . 2
- 2.1 Basic Searching Algorithms . . . 3

Chapter 3 Decision Theoretic Planning . . . 5

Chapter 4 Markov Decision Processes and Related Algorithms . . . 6
- 4.1 Markov Decision Processes . . . 6
- 4.2 Two basic dynamic programming algorithms for indefinite horizon MDPs . 7
  - 4.2.1 Dynamic programming . . . 7
  - 4.2.2 Value Iteration . . . 8
  - 4.2.3 Policy Iteration . . . 9
  - 4.2.4 Limitations . . . 9

Chapter 5 Planning Algorithms with the Help of Search Strategies . . . 11
- 5.1 RTDP . . . 11
- 5.2 HDP . . . 12
- 5.3 A*-based Algorithms . . . 12
  - 5.3.1 AO* . . . 12
  - 5.3.2 LAO* . . . 13
  - 5.3.3 BLAO* . . . 14
  - 5.3.4 RLAO* . . . 14
  - 5.3.5 Comparison of LAO*, BLAO* and RLAO* . . . 17
  - 5.3.6 MBLAO* . . . 18
- 5.4 Experiments . . . 22

Chapter 6 Priority-based Algorithms . . . 30
- 6.1 Prioritized Sweeping . . . 30
- 6.2 Improved Prioritized sweeping . . . 30
- 6.3 Focussed Dynamic Programming . . . 31
- 6.4 Topological Value Iteration . . . 32
- 6.5 Experiments . . . 33

Chpater 7 Conclusion and Future Work . . . 37

Bibliography 38

Vita 41

List of Tables

5.1 Convergence time on 10,000-state 5-successor state random MDPs 21
5.2 Maximum number of Bellman backups each iteration on 10,000-state 5-successor state random MDPs 21
5.3 Number of iterations on 10,000-state 5-successor state random MDPs . . . 22
5.4 Convergence time for different algorithms on different MDPs ($\delta = 10^{-6}$) . . 24
5.5 Number of backups performed for each algorithm on different MDPs ($\delta = 10^{-6}$) 24
5.6 # of backups performed by MBLAO* with different thread numbers on MCar(300×300) instances 26
5.7 Speedups achieved against LAO* and percentage of backups from the backward searches ($\delta = 10^{-6}$) 28
6.1 Problem Statistics and convergence time in CPU seconds for different algorithms with different heuristics ($\delta = 10^{-6}$) 34
6.2 Problem statistics and convergence time in CPU seconds for different algorithms on solving artificially generated layered MDPs with different number of layers ($|s|$=20000, ma=10, ms=20, $\delta = 10^{-6}$) 35
6.3 Problem statistics and convergence time in CPU seconds for different algorithms on solving artificially generated layered MDPs with different state space (n_l=20, ma=10, ms=20, $\delta = 10^{-6}$) 35

List of Figures

5.1 Convergence time on 4-action 5-successor state random MDPs 18
5.2 Convergence time on 10,000-state 4-action random MDPs 19
5.3 Number of iterations on 10,000-state 4-action random MDPs 20
5.4 Convergence time on 10,000-state 5-successor state random MDPs with . . 22
5.5 Statistics of random MDPs with fixed state space size and maximum successor state number 25

Chapter 1

Introduction

The problem of decision theoretic planning has become a central research topic in AI, not only because it is an extension to classical planning, but also due to its close connection with solving real world problems. Markov decision processes (MDPs), a graphical and mathematical framework, has been utilized by AI researchers to model decision theoretic planning problems. Solving MDPs has been an interesting research area for a long time, because of the slow convergence of MDP algorithms on real world domains. This paper concentrates on advances in expediting the convergence of existing dynamic programming algorithms, a basic tool to solve an MDP.

Chapter 2

Search

From Russell and Norvig's [24] points of view, the job of AI is to design the *agent program*: a function guiding the behavior of an agent, mapping from percept to actions. A specific class of agents, called *goal-based agents*, is provided with the information of problem states, its own percept, and also goal information. For example, the goal of a taxi driver is to take a passenger to his destination, and the states of the problem are the different situations the driver can possible meet while driving. *Search* and *planning* are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

To illustrate why search is useful and how it is performed, let us consider the following scenario: an agent wants to discover a path from the current state s it is in to some goal state g of a goal set G. Along the path, there are correlated costs, and the agent wants to discover a path with the minimum expected cost. We denote this cost as the *value* of s. However, it only knows the values of some states in the problem space, the goal information, and the problem formulation. One possible strategy of the agent is to examine a set of possible sequences of actions leading to states of known value. The result of the examinations is a directed graph, where edges have costs attached. The agent then chooses one path with the minimum cumulative cost. This process of looking for such a sequence is called *search*. A typical search problem is the graph shortest-path problem. Many problems can be reduced to it, such as robot navigation, DNA sequence mapping [18], etc. A search algorithm takes a problem as an input and outputs a solution in the form of an action sequence, or a plan.

Research on search has concentrated on finding the right search strategy for different problems. The following four aspects are often used as criteria to evaluate those strategies [24]:

- *Completeness*: if there is at least one solution, is the strategy guaranteed to find one of them?

- *Time complexity*: what is the asymptotic time to find a solution in the worst and the average case?
- *Space complexity*: how much storage space is needed to perform the search in the worst and the average case?
- *Optimality*: is the strategy guaranteed to find the best if there are several solutions?

2.1 Basic Searching Algorithms

Search strategies can be classified into two categories: *uninformed search* and *informed search*. Uninformed search or blind search means that the agents (usually in the form of programs) have absolutely no idea about the number of steps it takes to a goal state or any information about state values. Their only ability is to distinguish a goal state from a non-goal state. Some of the well known uninformed search strategies are: *breadth-first search*, *depth-first search*, *iterative deepening search*, *bidirectional search*.

Strategies that use additional information of the problems are called *informed search* or *heuristic search*. A heuristic value function is usually denoted by h: $h(n)$ = *estimated cost of the shortest path from the state* n *to a goal state*. A straightforward use of the heuristic values is to initialize the values of the state space. By doing so, the initial values are usually more reasonable and informative than arbitrarily assigned initial values. Searching and planning with the help of heuristic functions is usually faster than without.

*A** [16] is a basic heuristic search algorithm used in state space search problems. It estimates the value of a state by combining the two evaluation functions g and h: $f(n) = g(n) + h(n)$, where $g(n)$ gives the cost from the start state to the state n, and $h(n)$ is the estimated cost of the cheapest path from the state n to the goal, so $f(n)$ is the estimated cost of the cheapest solution from the start state to the goal state that passes through n. Thus, in finding the cheapest solution we first search along states with lower f values. A* search is categorized as an informed search method, because it makes use of knowledge of the universe by referring to the heuristic function h. The heuristic function h is said to be *admissible* if it never overestimates the value of any state. Admissible heuristic functions

are especially useful in informed search algorithms. It has been proved that A* search is both complete and optimal when h is admissible. Interested readers can find the proofs in [24].

Chapter 3

Decision Theoretic Planning

In the problems discussed above, we assume there is only one possible outcome after taking an action. In the real world, there exist many problems with uncertain outcomes, or nondeterministic problems. In a nondeterministic problem, an action a makes the system change from one state s_1 to another state s_1'. However, s_1' is not uniquely determined by s_1 and a. Often, there are a set S_1 of possible resulting states after taking action a in state s_1, and the system changes to some state $s_1' \in S_1$ nondeterministically. *Decision theoretic planning* (DTP) is an attractive extension of the classical AI planning paradigm, because it models problems in which actions may have uncertain and cyclic effects. Roughly speaking, the aim of a DTP problem is to form a course of action that has low expected cost when guaranteed to achieve the goal. Much work on DTP has used the Markov decision process framework as a model.

Chapter 4

Markov Decision Processes and Related Algorithms

In this section, the definition of the Markov Decision Processes will be given first, followed by the introduction of basic strategies of solving MDPs.

4.1 Markov Decision Processes

MDPs are used widely in the AI literature for representing stochastic sequential decision problems. A standard MDP has the following components [9]:

A finite set of *states* S: A state $s \in S$ is a description of the system at a given time. If we regard that a system evolves discretely rather than continuously, we can partition a system into a sequence of *stages*, and a system is at one particular state at each stage. Any event will make the system change from one state s at stage t to the another state s' and proceeds to stage $t+1$. In our work, we assume time is measured in discrete units.

A finite set of *actions* A: At each stage t of the process and each state s, the agent has a set of applicable actions $A_s^t \subseteq A$. When an action is performed, the system makes a nondeterministic transition.

Transition functions $T : S \times S \rightarrow \mathbb{R}$: Each action is appended with a transition function that tells the likelihood of the system changing from one state to another state as a result of performing that action. $T_a(s'|s)$ gives the probability of transition from state s to state s' after performing action a.

Cost functions: $C : S \times A \rightarrow \mathbb{R}$: Each state-action pair (s, a) is associated with an instant cost. For some MDPs, cost functions can be replaced or complemented by the *reward functions*.

The *horizon* of an MDP is the total number of stages the system is evaluated. When the horizon is a finite number H, solving the MDP means finding the best action to take at each stage and state that minimizes the total expected cost. More concretely, the chosen actions $a^0, \ldots, a^{H-1}$ should minimize the value $f(s) = \sum_{i=0}^{H-1} C(s^i, a^i)$, where $s_0 = s$.

For *infinite-horizon* or *indefinite-horizon* problems, problems when the horizon is infinite or unknown, the cost is accumulated over an infinitely long path. To emphasize the relative importance of instant costs, a *discount factor* $\gamma \in (0, 1]$ is used for future costs. With discount factor γ, our goal is to minimize $f(s) = \sum_{i=0}^{\infty} \gamma^i C(s^i, a^i)$.

Given an MDP, we define a *policy* $\pi : S \times \mathbf{N} \rightarrow A$ to be a function from the state-stage pairs (s, n) to actions, where $n \in \mathbf{N}$ is the number of stages left. A *value function* V^π for policy π, $V^\pi : S \rightarrow \mathbf{R}$, denotes the value of the total expected cost starting from state s and following the policy π. A policy π_1 dominates another policy π_2 if $V^{\pi_1}(s) \leq V^{\pi_2}(s)$ for all $s \in S$. An *optimal policy* π^* is a policy that is not dominated by any other policies. It guides the agent to pick the most appropriate action at each stage and state that minimizes the total expected cost. We describe the expected cost accumulated by starting at state s and following the optimal policy by the *optimal value function* V^*. Note that V^* is unique, while π^* is sometimes not.

A *goal-based MDP* usually has two more components s_0 and G, where $s_0 \in S$ is an initial state or start state and $G \subseteq S$ is a set of goal state. An optimal policy guides the system from s_0 to some state in G with the smallest expected cost. So to solve a goal-based MDP means to find $V^*(s_0)$ and $\pi^*(\cdot)$. We usually only consider goal-based MDPs with infinite time horizons. Throughout the rest of the paper, we will use goal-based MDPs as our problem paradigm.

4.2 Two basic dynamic programming algorithms for indefinite horizon MDPs

This section studies two basic dynamic programming algorithms of solving MDPs: *value iteration* and *policy iteration*.

4.2.1 Dynamic programming

Bellman showed that the set of value functions V^π, evaluated according to different horizon values, can be solved by dynamic programming [2]. For finite-horizon MDPs, $V_0^\pi(s)$ is defined to be $C(s, \pi(s))$, and V_t^π ($t > 0$) is defined iteratively:

$$V_t^\pi(s) = C(s, \pi(s)) + \sum_{s' \in S} T_{\pi(s)}(s'|s) V_{t-1}^\pi(s'). \tag{4.1}$$

Similarly, the optimal value function $V_0^*(s)$ is defined to be $min_{a \in A(s)} C(s,a)$, and V_t^* ($t > 0$) is defined iteratively:

$$V_t^*(s) = min_{a \in A(s)}[C(s,a) + \sum_{s' \in S} T_a(s'|s)V_{t-1}^*(s')]. \tag{4.2}$$

For infinite-horizon or indefinite-horizon MDPs, since at any stage the number of stages left is infinite or indefinite, we can regard all stages as the same. In this case, the policy and value function of different states are *stationary* [3, 23, 2], because they do not depend on the number of stages left. The value functions of a policy π are defined as:

$$V^\pi(s) = C(s, \pi(s)) + \gamma \sum_{s' \in S} T_{\pi(s)}(s'|s)V^\pi(s'), \gamma \in (0, 1], \tag{4.3}$$

and the optimal value function is defined as:

$$V^*(s) = min_{a \in A(s)}[C(s,a) + \gamma \sum_{s' \in S} T_a(s'|s)V^*(s')], \gamma \in (0, 1]. \tag{4.4}$$

The *Bellman equation* is an equality that is satisfied by a system of value functions in the form of Equation 4.2 or 4.4. By applying Bellman equations, we can use dynamic programming techniques to compute the optimal value function. An optimal policy is easily extracted by choosing an action for each state that contributes to its optimal value function. The two basic dynamic programming algorithms to solve MDPs are value iteration and policy iteration.

4.2.2 Value Iteration

The basic idea of value iteration [2] is to iteratively refine the grossly initialized value functions. The pseudocode of a variant of value iteration named *Gauss-Seidel value iteration* [3, 8] is shown in Algorithm 1. This algorithm improves an evaluation function by means of dynamic programming. The values of each state are initialized by their best cost function values. Then value iteration iteratively updates the value function of the whole state space by applying Equation 4.4. We call one such update a *Bellman backup*. The *Bellman residual* of a state s is defined to be the difference between the value functions of s after and before a Bellman backup. The *Bellman error* is the maximum Bellman residual

Algorithm 1: Gauss-Seidel Value Iteration

Input: S, A, γ, δ
for every state s **do**
  $V(s) \leftarrow min_a C(s,a)$
  $\pi(s) \leftarrow argmin_a C(s,a)$
end for
repeat
  for every state s **do**
    Backup(s)
  end for
until (Bellman error of $S < \delta$)
Backup(s**)**
$V(s) \leftarrow min_a[C(s,a) + \gamma \sum_{s'} T_a(s'|s)V(s')]$
$\pi(s) \leftarrow argmin_a[C(s,a) + \gamma \sum_{s'} T_a(s'|s)V(s')]$

of all the states. We call a particular state s *converged* when the Bellman residual of s is smaller than a threshold value δ, and all its *successors*, the set of states that can be changed from s, are converged. Otherwise, s is not converged or *unconverged*. When value iteration finds, at some iteration, that the Bellman error is less than the input error bound δ, it concludes that all the states are converged and terminates.

4.2.3 Policy Iteration

Howard's *policy iteration* [17] is another approach for solving MDP problems. It has two interleaved phases: policy evaluation and policy improvement. In the policy evaluation phase the value functions of the state space under the current optimal policy are updated by solving systems of $|S|$ linear equations. The policy improvement phase updates the current optimal policy by choosing greedy actions based on the value functions calculated in the policy evaluation phase. If a certain policy improvement step has no policy change, the algorithm stops. In practice, although policy iteration spends more computational time at each iteration, it converges in fewer iterations than does value iteration.

4.2.4 Limitations

Value iteration and policy iteration are both optimal and complete. Although they converge in time polynomial in $|S|$ and $1/(1-\gamma)$ [19], the two algorithms suffer from efficiency

problems. For realistic problems when the state spaces are large, these algorithms take a long time to get an optimal policy. For this reason AI researchers have devoted energy to finding algorithms that can converge faster. The main drawback of both algorithms is that all the states in the state space are backed up in each iteration. There are several reasons that this is not necessary. First, some states are not reachable from the start state, so they are irrelevant in deciding the value function of the start state. Second, the value functions of some states converge faster than others, so in some iterations we actually only need to update values of a subset of the all the states. Third, the value functions of each state are initialized by instant cost functions, and sometimes this initialization is too conservative, so it normally takes quite a few iterations for VI and PI to converge.

Chapter 5

Planning Algorithms with the Help of Search Strategies

From the previous section we know that value iteration and policy iteration are not practical for solving MDPs with large state spaces. To overcome this problem, researchers have proposed algorithms that make use of search strategies. This section discusses planning algorithms that fall into this category.

5.1 RTDP

Barto et al. [1] proposed the real-time dynamic programming (RTDP) algorithm. It is the first dynamic programming approach combined with search. RTDP explores possible trials (paths from the start state to a goal state) to simulate the execution of the system. At the beginning of each trial, the current state is initialized to the start state. At each step of the trial, RTDP backs up the current state, picks a greedy action based on the current value function, and changes the current state stochastically according to the transition function. Each trial stops when a goal state is reached or a maximum number of steps are accomplished. We define a state s' *reachable* from another state s if s' can be changed from s within finite number of transition steps, otherwise we say s' is *unreachable* from s. It is easily seen that the states unreachable from the start state are ignored in the trials and therefore never backed up.

The advantage of RTDP is that it can find a good sub-optimal policy quite fast, because a policy is found when a trial terminates at a goal state. But its convergence is slow. Bonet and Geffner extended RTDP to labeled RTDP (LRTDP) [7], and the convergence of LRTDP is faster than RTDP. They define a state s as *solved* if the Bellman residuals of s and all the state that are reachable through the optimal policy from s are small enough. LRTDP labels solved states when a trial is finished. In later trials, solved states are regarded as "tip" states, and they are no longer backed up. LRTDP converges when the start state is solved.

5.2 HDP

HDP is another state-of-the-art algorithm by Bonet and Geffner [6]. It not only uses the similar labeling technique as LRTDP, but also discovers the strongly connected components in the solution graph of an MDP. HDP labels a component as solved when all the states in that component have been labeled. HDP expands and updates states in a depth-first fashion rooted at the start states. All the states belonging to the solved components are regarded as tip states. Experimental results show that HDP converges faster than LAO* and LRTDP on most of the racetrack MDP benchmarks when the heuristic function h_{min} [7] is used.

5.3 A*-based Algorithms

The A* algorithm, discussed in Section 2.1, is a heuristic search algorithm for deterministic planning problems. This section discusses in detail some extensions and related MDP algorithms to A*.

5.3.1 AO*

An extension to the A* algorithm, *AO** [22], applies to acyclic *AND/OR graphs* or acyclic MDPs. It finds an optimal solution that has the structure of a tree. Like other heuristic search algorithms, AO* finds the optimal solution without considering the whole state space. The *explicit graph* or *solution graph* of an MDP is a subgraph of the original MDP that includes all the states that are reachable by applying the optimal policy from the start state and related actions. In an MDP, the states in its explicit graph form the set of relevant states, because the values of other states do not directly influence $V^*(s_0)$. A *partial explicit graph* is defined similarly to a explicit graph, with the difference that the tip states of a partial solution graph may not be a goal state. As the algorithm proceeds, it constructs and expands a partial explicit graph G, which initially only contains the start state. A tip (leaf) state of G is *terminal* if it is a goal state; otherwise, it is *nonterminal*. AO* keeps *expanding* the partial solution graph until there are no more nonterminal tip states. A nonterminal tip state is expanded by adding to the explicit graph one of its actions and all the successor states that currently are not in the explicit graph. A set Z that contains all the newly added

states and their *ancestors* (the set of relevant states that can reach them) is built after the expansion. AO* then repeatedly deletes from Z states with no descendents in Z and backs up the deleted states until Z becomes empty. It is always possible to back up states in Z in this way, since AND/OR graphs do not have cycles, and neither do the sets Z.

5.3.2 LAO*

Algorithm 2: Improved LAO*

Input: S, A, γ, δ
for every state s **do**
  $V(s) \leftarrow$ heuristic value
  $\pi(s) \leftarrow argmin_a V(s)$
end for
repeat
  for all state s **do**
    $s.expanded \leftarrow false$
  end for
  $G \leftarrow \emptyset$
  $G \leftarrow G \cup \{s_0\}$
  Search(G, s_0)
until (Convergence Test(G, δ))
return $V(\cdot)$, $\pi(\cdot)$
Search(G, s**)**
$s.expanded \leftarrow true$
Backup(s)
$a \leftarrow \pi(s)$
for every successor state s' of a **do**
  if $s'.expanded = false$ **and** s' is not a goal state **then**
    $G \leftarrow G \cup \{s'\}$
    Search(G, s')
  end if
end for

---

*LAO** [15, 14] is an extension to the AO* algorithm that can handle solutions which contain loops. Thus, it can handle MDPs. Instead of updating states in Z by their topological order, it calculates their values by value iteration, because topological orders among these states may not exist. An improved version of LAO*, improved LAO* (ILAO*) [14], interleaves the explicit graph construction and value iteration. Its pseudocode is shown in Algorithm 2. Different with LAO*, ILAO* backs up states only once after each expansion.

Several runs of explicit graph construction are done until all the states in the most recent explicit graph are converged.

5.3.3 BLAO*

Bhuma and Goldsmith extended the LAO* algorithm to BLAO* [4, 5] by searching from both the start state and the goal state in parallel. BLAO* is the first bidirectional heuristic search MDP algorithm. In detail, BLAO* has two searches: forward search and backward search. Both searches use heuristic functions and start concurrently at each iteration. The forward search is similar to the ILAO* algorithm. It begins at the start state and expands the successor states of best known action towards the goal state. The backward search originates from the goal state. During the backward search, a state s in G that has not been expanded is expanded along its *best predecessor state*. For one state s' to become the best predecessor state of another state s, it has to fulfill two requirements. First, s' must belong to the set of states L, such that the current best action of each state in L has s as a successor state. Second, s' must be a state that has the highest probability of reaching s of all the states in L.

Each forward (backward) search branch terminates when the search loops back to an expanded state, or reaches the goal (start) state. When both searches terminate, we call it an *iteration*. After each iteration, the convergence test checks whether there exists some unexpanded states or whether the maximum Bellman residual of all the states in G exceeds some predefined threshold value. If not, the algorithm returns the optimal value function and policy.

5.3.4 RLAO*

We studied the BLAO* algorithm [11] and discovered that, originally, the efficiency of BLAO* has been greatly underestimated [5]. To fully compare unidirectional and bidirectional heuristic search ideas, we introduced a sheer backward heuristic search algorithm named RLAO*.

In the graphical representation of LAO*, each state points to several actions—those

Algorithm 3: BLAO*

Input: S, A, γ, δ
for every state s **do**
  $V(s) \leftarrow$ heuristic value
  $\pi(s) \leftarrow argmin_a V(s)$
end for
$iteration \leftarrow 0$
repeat
  $iteration \leftarrow iteration + 1$
  for every state s **do**
    $s.expanded \leftarrow false$
  end for
  $G \leftarrow \emptyset$
  $G \leftarrow G \cup \{s_0\} \cup \{goal\}$
  //Start the following two threads concurrently
  Forward Search(s_0)
  Backward Search($goal$)
until (Convergence Test(G, δ))
return $V(\cdot)$, $\pi(\cdot)$
Forward Search(s**)**
$s.expanded \leftarrow true$
$a \leftarrow \pi(s)$
Backup(s)
while a has any unexpanded successor state s' **do**
  if s' is not the goal state **then**
    $G \leftarrow G \cup \{s'\}$
    Forward search(s')
  end if
end while
Backward Search(s**)**
$s.expanded \leftarrow true$
Backup(s)
$s' \leftarrow$ best predecessor state of s
if s' is not the start state **and** has not been expanded **then**
  Backward search(s')
end if

Algorithm 4: Reverse LAO*

```
Input: S, A, γ, δ
for every state s do
   V(s) ← heuristic value
   π(s) ← argmin_a V(s)
end for
repeat
   for all state s do
      s.expanded ← false
   end for
   G ← ∅
   G ← G ∪ {s_0}
   Search(G, s_0)
until (Convergence Test(G, δ))
return V(·), π(·)
Search(G, s)
s.expanded ← true
Backup(s)
for every successor state s' of a do
   if s'.expanded = false and s' is not the start state then
      G ← G ∪ {s'}
      Search(G, s')
   end if
end for
```

that are applicable at that state, and each action points to some other states, which are the successors of applying such action. For RLAO*, we maintain the same graphical structure. In addition, we keep a *reverse graph*, which contains the same set of vertices and edges as the original graph, but the directions of all the directed edges in the original graph are reversed. This means all the states point to the actions that lead to them, and all the actions point to the states in which they can be applied.

The pseudocode of RLAO* is given in Algorithm 4. The main idea is to propagate the value functions from the goal to the start state by means of expansion. In the main function, we iteratively expand the graph. In each iteration, we pick the goal state and expand it. In expanding a state, we mark it as expanded, perform a backup and check if it has any unexpanded successors in the reverse graph. If so, we pick one such state and expand it. The RLAO* algorithm can also be seen as a depth first search on the reverse

graph. In this case, if we look at each expansion of LAO* as moving forward one step, we can think of one expansion of RLAO* as moving backwards one step. The convergence judgment of RLAO* is very similar to the LAO* algorithm, with the difference that the explicit graphs are constructed in the backward manner.

5.3.5 Comparison of LAO*, BLAO* and RLAO*

To compare the three closely related algorithms—LAO*, BLAO*, and RLAO*, we constructed families of *random MDPs*. The parameters of random MDPs are:

- the size of the state space, $|S|$;
- the maximum number of actions each state can have, m_a;
- the maximum number of successor states of each action, m_s.

Given a configuration of parameters, we build an random MDP as follows: For each state s, we let the pseudorandom number generator of C pick the number of actions from $(1, 2, \ldots, m_a)$. Then, for each action, we allow the number of successor states of that action to be $(1, 2, \ldots, m_s)$ with equal probability. The successor states are chosen uniformly from S together with normalized transition probabilities. For each random MDP configuration, we generated 20 MDPs and ran the three algorithms on them and averaged their statistics. The threshold value δ here was consistently chosen to be 10^{-6}. In our experiments, we found that RLAO* ran about 5% faster than LAO* and BLAO* when the state space was small (under 5000 states) and sparsely connected ($m_a = 2, m_s = 5$). For densely connected graphs with large state space, Figure 5.1 shows the convergence times when $|S|$ changes. Figure 5.2 and Figure 5.3 show the convergence times and number of iterations as m_s varies. Table 5.1 gives part of the convergence times when the number of actions per state varies. Figure 5.4 plots the convergence times of the three algorithms when states have 10 to 50 actions available.

Our experiments discovered that RLAO* converged slower than the other two when the action number was large, because the branching factors of the backward search graphs were usually larger than those of the forward search graphs.

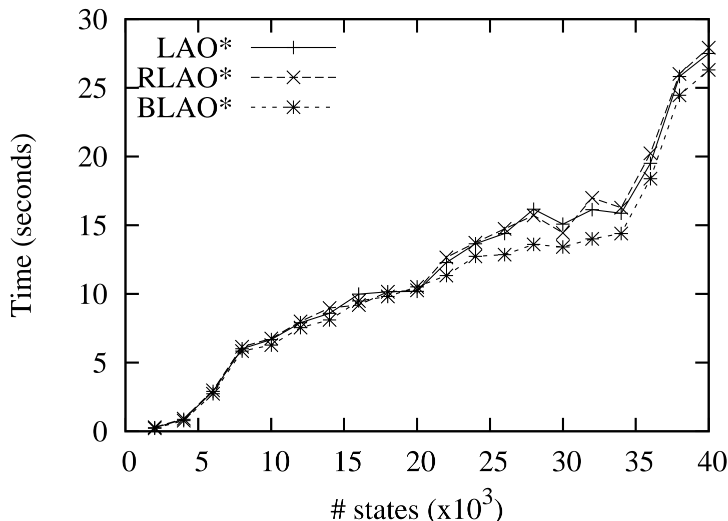

Figure 5.1: Convergence time on 4-action 5-successor state random MDPs

When the number of actions per state was relatively small, BLAO* displayed no advantages over the other two algorithms. Nevertheless, when the number of actions was larger than 10, BLAO* beat the other two. This is because the backward expansion of BLAO* managed to keep the number of Bellman backups under control, shown in Table 5.2 and Table 5.3. We also implemented another version of BLAO*, hoping to further control the expanded states by strengthening the backward search. In the backward search of BLAO* the expansion is always undertaken along the best previous state. In the new BLAO* algorithm the backward expansion is not only along the best previous state, but every possible predecessor. This means all the states that can reach the current state are backed up. However, the comparison between the two versions of BLAO* showed that the new algorithm did not trivialize the problem, which from a different point of view, proved the effectiveness of BLAO*.

5.3.6 MBLAO*

From our extensive experiments on the performance of LAO* and BLAO* [11], we noticed that BLAO* consistently outperformed LAO* by about 10% in racetrack domains. More promisingly, in random MDPs with large m_a, BLAO* sometimes ran three times as fast as LAO*. This performance gain was not achieved by decreasing the size of G to

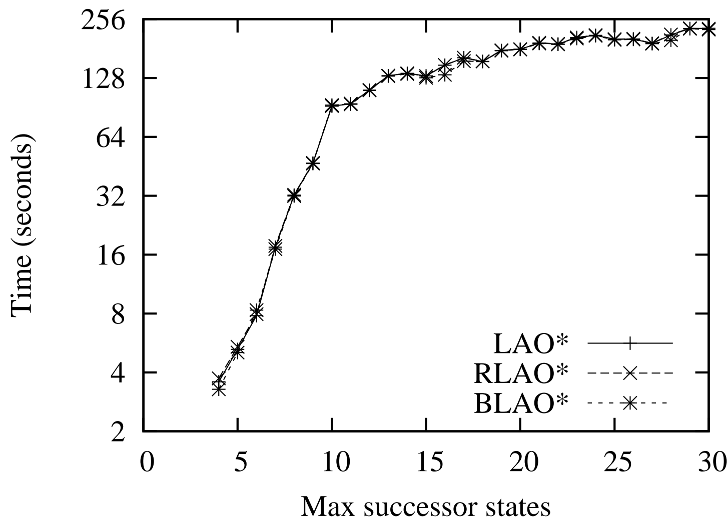

Figure 5.2: Convergence time on 10,000-state 4-action random MDPs

a larger extent. Rather, BLAO* converged faster than LAO* because it performed fewer backups. In order to see why this happens, let us take a closer high-level look at the heuristic search ideas.

Our first key observation is that, among all existing heuristic search MDP planners, the heuristic functions in use are mostly generated in the backward manner. Take one of the most recent heuristics, h_{min} [7], for example. It is an estimation of a least expensive path from a state to the goal state.

$$h_{min}(s) = min_{a \in A(s)}[C(s,a) + min_{s'} h_{min}(s')], \gamma \in [0,1], T_a(s'|s) > 0. \tag{5.1}$$

The calculation of this heuristic function can be implemented by a breadth-first backward search from the goal state. In fact, heuristic functions with similar semantics to h_{min} are all generated in the backward manner. The h_{min} heuristic replaces the normal weighted sum of the successors' values in the Bellman equation by the heuristic value of its best successor, so it is a lower bound of V^*. Another fact is that the heuristic values of states near the initial state are often less accurate than those of states near the goal, since further "away" from the goal, a larger error in the estimation is propagated. In unidirectional heuristic search algorithms, we only search in one direction: from the initial state to the goal state. So, as long as the search has not reached the portion that is near the goal, when we do Bellman

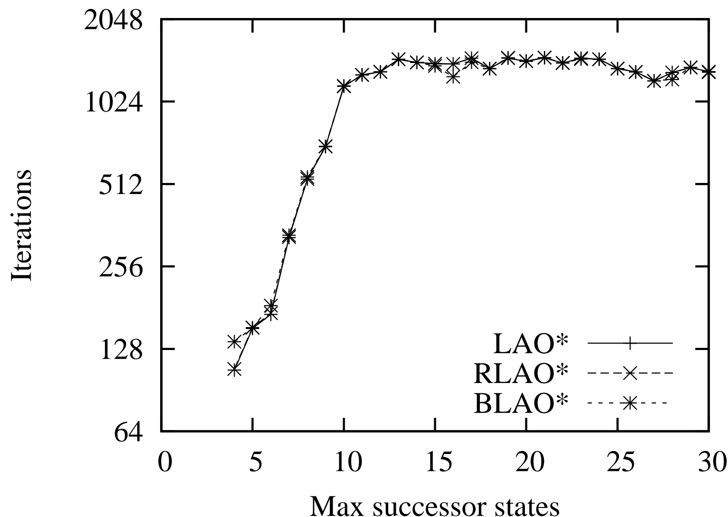

Figure 5.3: Number of iterations on 10,000-state 4-action random MDPs

backups, the values $V(\cdot)$ appeared on the right hand side of Bellman equations are quite crude, since they are initiated by inaccurate heuristic functions and have not been polished enough. So, the backups performed during these steps are not very useful. In BLAO*, by doing backward searches from time to time, we can propagate more accurate values from the goal state. Backups performed along the backward searches help refine the value functions of states that are far away from the goal. In this case, the backups performed during the early steps of the forward search make more sense.

The intuition behind MBLAO* is based on the above observation. We wondered: can we further decrease the number of backups? Our previous attempt to change the single-source backward search trial into a single-source backward search tree was not successful. In MBLAO*, we tried something new. The idea is to concurrently start several threads. One of them is the same as the forward search in BLAO*, and the rest of them are backward searches, but with different starting points. In that way we change the single-source backward search trial into multiple-source backward search trials. The reason for this change is: On the one hand, one backward search from the goal could help propagate more accurate values from the goal, but not other sources. On the other hand, the value of a state depends on the values of all its successors, so the function of a single-source backward search is

# actions	LAO*	RLAO*	BLAO*
2	33.670000	32.830000	32.170000
4	13.570000	13.710000	13.070000
6	4.450000	4.600000	4.190000
8	1.600000	1.880000	1.310000
10	1.880000	2.210000	1.190000
20	1.420000	1.920000	0.760000
30	0.750000	1.770000	0.320000
40	0.660000	1.460000	0.240000
50	0.470000	1.430000	0.170000

Table 5.1: Convergence time on 10,000-state 5-successor state random MDPs

# actions	LAO*	RLAO*	BLAO*
10	9064	9986	7455
15	8700	9967	7326
20	8247	9987	6135
25	8875	9994	6135
30	9103	9980	4179
35	8788	9995	4138
40	6421	9996	5879
45	5948	9972	3057

Table 5.2: Maximum number of Bellman backups each iteration on 10,000-state 5-successor state random MDPs

limited. This could be complemented by backward searches from other places.

We define the *optimal path* of an MDP to be the most probable path originating from s_0, if we follow the optimal policy and choose the successor states the ones with the highest transition probabilities. Before we know the optimal policy, “optimal paths” are only constructed relative to the best known policy. The starting points for the backward searches are selected from states along the optimal path, not including the start state. Since planning is mostly interested in the states on the optimal path because of their contributions to $V^*(s_0)$, value propagations from middle points on this path could be helpful. Also note that the optimal path may change from iteration to iteration, so the sources and trajectories of the backward searches may also change. The pseudocode of MBLAO* is shown in Algorithm

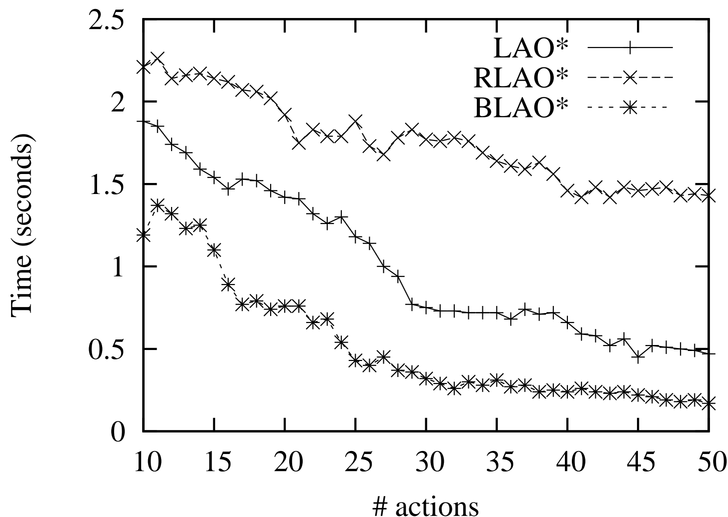

Figure 5.4: Convergence time on 10,000-state 5-successor state random MDPs with

# actions	LAO*	RLAO*	BLAO*
10	25	35	29
15	25	29	29
20	24	23	24
25	19	23	19
30	6	12	10
35	6	8	7
40	7	9	11
45	10	8	18

Table 5.3: Number of iterations on 10,000-state 5-successor state random MDPs

5. The input number n gives the total number of forward and backward search threads in one iteration. As long as states in the explicit graph G have not converged, MBLAO* initializes n (> 1) threads in parallel. One of them is the forward search originating from the start state, and the rest of them are backward ones. After all the searches finish and we say one iteration is done, the algorithm checks whether G is converged.

5.4 Experiments

We investigated the performance of MBLAO* by comparing it to VI, BLAO*, and several state-of-the-art unidirectional heuristic search MDP planners: LAO* (the improved version), LRTDP, HDP and FDP. In our experiments, we tested the MDP planners on four

Algorithm 5: MBLAO*

Input: S, A, γ, δ, n
for every state s **do**
  $V(s) \leftarrow$ heuristic value
  $\pi(s) \leftarrow argmin_a V(s)$
end for
$iteration \leftarrow 0$
repeat
  $iteration \leftarrow iteration + 1$
  $G \leftarrow \emptyset$
  for every state s **do**
    $s.expanded \leftarrow false$
  end for
  //Start the following n threads concurrently
  Forward Search(s_0)
  for $i \leftarrow 1$ to $n - 1$ **do**
    pick one state s along the optimal path
    $G \leftarrow G \cup \{s\}$
    Backward Search(s)
  end for
until (Convergence Test(G, δ))
return $V(\cdot)$, $\pi(\cdot)$

MDP domains from the literature: racetrack [1], mountain car (MCar), single-arm pendulum (SAP) and double-arm pendulum (DAP) [27] and the random MDP domain. Racetrack MDPs are simulations of a race car on tracks with various sizes and shapes. The state space is define by the position on the track and the instantaneous velocity of the car. At each state, a car can take at most nine different actions, that is, to move toward eight directions or to stay put. Each action has a small possibility of failure, which leads the car to an unintended state. When a car runs into a boundary, it is sent back to the starting point. We chose two racetrack MDPs. One is a small track with 1849 states, and the other has 21371 states. Mountain car is an optimal control problem, whose aim is to make the car reach the destination with enough momentum within minimum amount of time. Like the racetrack problems, the state space of MCar is defined by position and velocity of the car. SAP and DAP are domains that address minimum time optimal control problems in two and four dimensions respectively. They are similar to the MCar domain. The difference between

SAP and DAP, on the one hand, and MCar, on the other, is that the goal states[1] in SAP and DAP are reachable from the entire state space. All the algorithms except VI in our list are initial-state driven algorithms, but in MCar and SAP and DAP domains, we do not have any assumptions about initial states. When we run algorithms on them, we randomly pick 10 states from the state space as initial states[2], and average the statistics over these experiments. For more detailed discussions on MCar, SAP and DAP, please refer to [27].

Domains	explicit graph size	VI	LAO*	BLAO*	MBLAO*	LRTDP	HDP	FDP
Racetrack(small)	76	0.06	0.01	0.01	**0.00**	0.02	0.78	0.09
DAP(10^4)	9252	1.27	0.74	0.73	**0.52**	1.01	70.83	4.24
Racetrack(big)	2250	2.08	1.73	1.37	**0.80**	20.06	10.49	11.13
MCar(300×300)	2660	6.45	1.21	1.01	**0.65**	8.78	1.17	173.07
SAP(300×300)	49514	48.51	4.36	3.64	**3.11**	N/A	N/A	N/A
MCar (400×400)	24094	N/A	0.78	0.57	**0.30**	0.55	1.57	N/A

Table 5.4: Convergence time for different algorithms on different MDPs ($\delta = 10^{-6}$)

Domains	explicit graph size	VI	LAO*	BLAO*	MBLAO*	LRTDP	HDP	FDP
Racetrack(small)	76	29584	3195	2986	**1877**	5166	5781	32207
DAP(10^4)	9252	721091	250959	250105	**232922**	353487	217959	1831535
Racetrack (big)	2250	854840	325988	305916	**283577**	2728517	577160	3176598
MCar(300×300)	2660	1102981	91015	86120	**43225**	453156	71640	83309915
SAP(300×300)	49514	48690019	3015618	2764891	**1828943**	N/A	N/A	N/A
MCar (400×400)	24094	N/A	457594	401740	**352719**	821740	400039	N/A

Table 5.5: Number of backups performed for each algorithm on different MDPs ($\delta = 10^{-6}$)

A specific domain with a specific state space is called a *problem*. For example, MCar with two dimensions, each of which has size 300, is one particular MCar problem. We first tested the convergence times of the algorithms on the "real-world" domain problems. For MCar, SAP and DAP problems, we randomly chose 10 initial states for each problem and call each pair consisting of a problem and an initial state an *instance*. For every particular problem, we averaged the statistics of the chosen instances. Throughout these experiments, we fixed the thread number of MBLAO* to be 10. The convergence time and number of backups performed by each algorithm were listed in Tables 5.4 and 5.5. We used the h_{min} heuristic function in all the experiments discussed in this section and set the cutoff time to be 30 minutes.

[1] Regarding goal states in MCar, SAP and DAP, we refer to states that have positive instant reward. Each SAP or DAP problem has only one goal state, which stands for the equilibrium point. But an MCar problem has several, because the destination can be reached with various speeds.

[2] For MCar problems, we pick initial states with the constraints that they can reach a goal state.

In Table 5.4 we see that MBLAO* outperformed other algorithms in all the listed problems. Excluding MBLAO*, BLAO* outperformed the rest of the algorithms except on the MCar(400×400) problem, where it ran slower than LRTDP. The reason for the generally fast convergence of BLAO* and MBLAO* is clearly demonstrated by Table 5.5. The numbers of backups performed by BLAO* were smaller than those by LAO*, and MBLAO* performed even fewer.

We also tested the listed algorithms on the random MDP problems. We found in [11] that the increases in the state space size $|S|$ or m_s, the maximum number of successor states each action can have, did not contribute much to the speedup of BLAO*. Therefore, in the experiments, we fixed $|S| = 10000$ and $m_s = 5$ and varied m_a. Convergence times were plotted in Figure 5.5. We chose problems with a relatively small space size because those random MDPs can easily fit in memory and can therefore be solved relatively quickly. Note that the statistics on larger problems were similar to those with $|S| = 10000$. In this set of experiments, we generated 20 instances for each possible m_a value, and averaged the convergence time of each algorithm on the instance for that particular m_a.

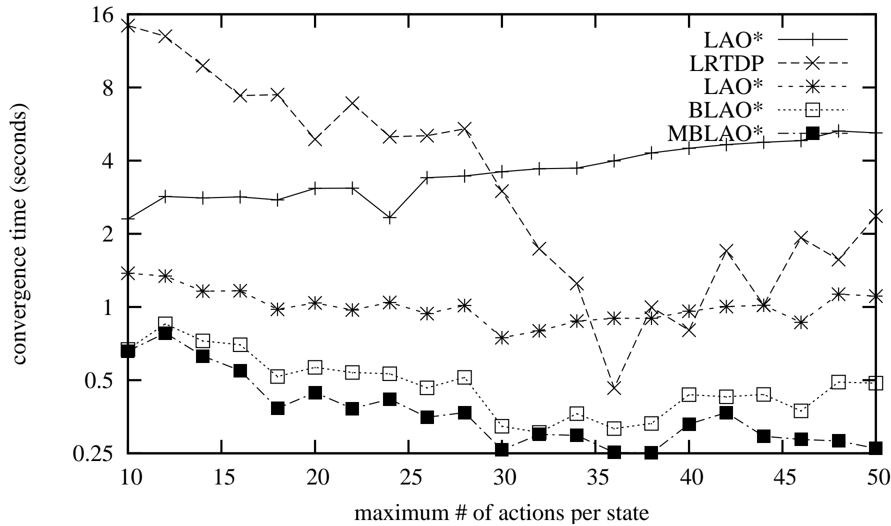

Figure 5.5: Statistics of random MDPs with fixed state space size and maximum successor state number

From Figure 5.5, we observed that MBLAO* and BLAO* always converged faster than VI, LAO* and LRTDP. LAO* was faster than VI. LRTDP was slower than VI for small

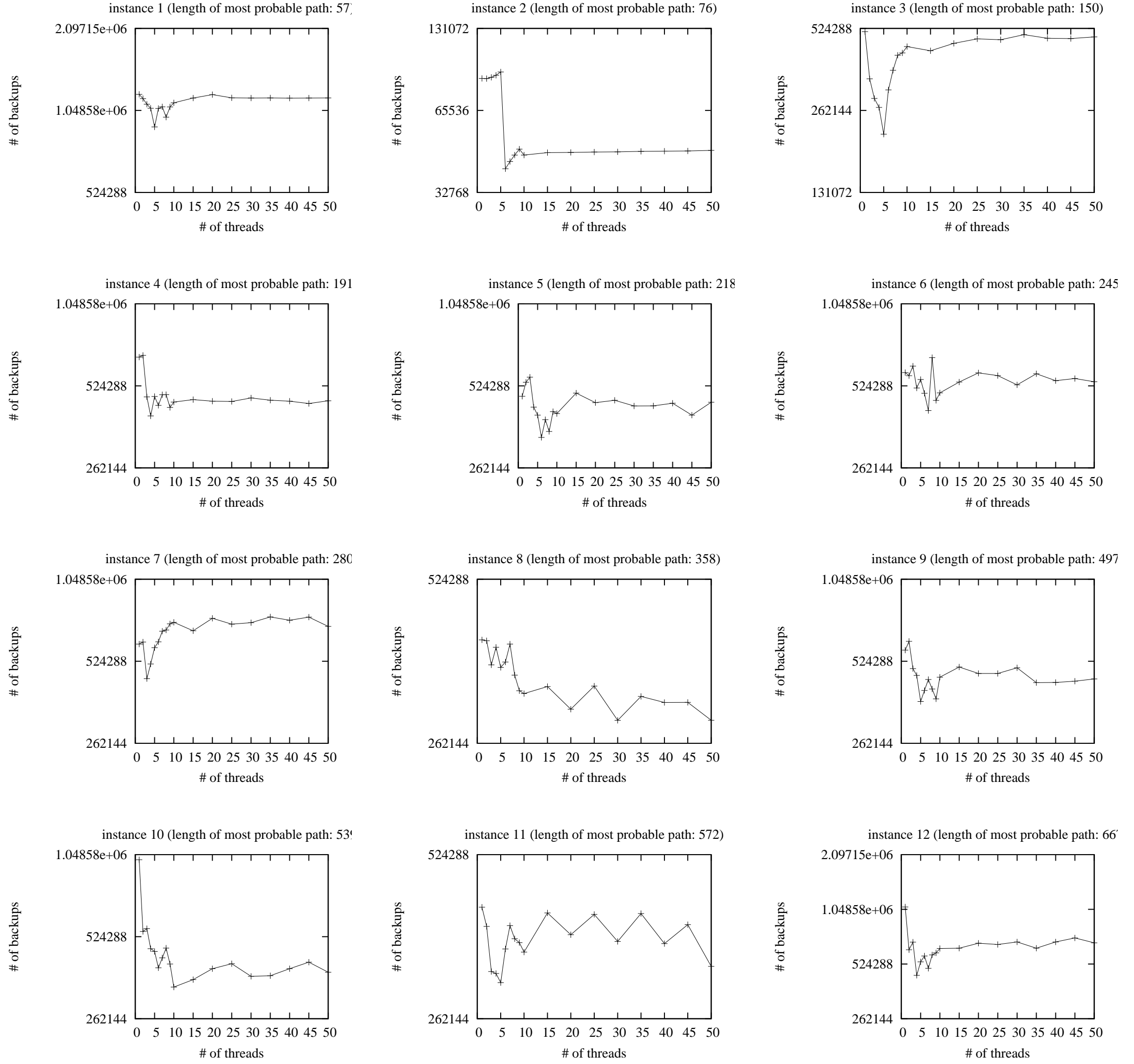

Table 5.6: # of backups performed by MBLAO* with different thread numbers on MCar(300×300) instances

action numbers, but as good as LAO* for large action numbers. Following the pattern that BLAO* outperforms LAO*, MBLAO* outperformed BLAO* in all the random MDP problems. In the best case, MBLAO* ran 80% faster than BLAO*, more than three times faster than LAO*, eight times faster than LRTDP and 18 times faster than VI, when $m_a = 50$.

We then measured how the number of threads influenced the performance of MBLAO*. In this experiment, we used a MCar(300×300) problem in particular. We chose 12 instances and tested LAO*, BLAO*, and MBLAO* (with thread numbers from 3 to 50) on

them. The numbers of backups performed for each instance are plotted in the figures of Table 5.6, and the figures are arranged in ascending order of the length of the optimal path of different problem instances. In each figure of Table 5.6, LAO* and BLAO* are regarded as MBLAO* with one and two search threads. We define the *optimal thread number* with respect to a problem instance to be the number of threads in a run of MBLAO* that performed the least number of backups on that instance. From observing these figures, we had the following conjectures.

- A larger thread number does not necessarily result in better performance. This is because, when the number of threads is larger than "enough", some of the backward propagations are unnecessary, so the backward expansions themselves cause a lot of overhead. Furthermore, too many backward searches distracts the algorithm from the central efforts on the forward search.
- For different instances, the performance curves (figures in Table 5.6) neither have a common shape nor have the same optimal thread numbers (the corresponding x values of the minimums in the figures).
- No rule is found about how the length of the optimal path influences the optimal thread number. As the length of the optimal path increases, the optimal thread number does not increase monotonically. At first, we considered that the harder the problem—which probably leads to a longer optimal path—the more backward searches we need. We tried to find a function of the optimal thread number based on the length of the optimal path but failed. This can be partially by the fact that the optimal thread number depends on many other factors, such as the internal graph structure and branching factor of an MDP. However, our experiments show that the optimal thread number seldom exceeded 10. So we do not expect MBLAO* to need very large thread numbers in general.

From the experiments we have recounted, we know that bidirectional search algorithms outperform their unidirectional counterparts. We want to know whether this is because the

backward search is itself superior to the forward search. We proposed RLAO* [11], a completely backward search algorithm. Experimental results showed that RLAO* converges more slowly than LAO*, except for very small problems with small branching factors. The faster convergence of LAO*, as compared to RLAO*, indicates that the speedup we get from making the searches bidirectional is not simply because backward search is better than forward search.

In our next experiment we computed the speedups of BLAO* and MBLAO* against LAO* and calculated the proportion of backups done by the backward searches on problems from all the domains. For each domain we picked 50 problems (except for the racetrack, where we only picked three problems) and averaged the results. We abbreviate Random MDP as RMDP in Table 5.7. The first row of Table 5.7 gives the *accuracy* of the initial heuristic value. It is calculated as the percentage of the initial heuristic value of the start state against its value with respect to the optimal policy. For example, if the initial heuristic value for the start state is 10.0, and its actual value is 20.0, then the accuracy is 50%. The second and fourth rows show the speedups BLAO* and MBLAO* achieved against LAO*. The third and fifth rows show the percentage of backups performed by backward searches. For MBLAO*, we chose the statistics for MBLAO* with the optimal thread number of threads. The optimal thread numbers here did not exceed 10 and were seldom smaller than 5.

		Racetrack	DAP	MCar	SAP	RMDP ($\|S\| = 10^4$)	RMDP ($\|S\| = 5 \times 10^4$)
Accuracy (%)		66.91	64.65	68.19	72.29	57.12	59.76
BLAO*	Speedup	1.15	1.18	1.11	1.20	1.85	1.73
BLAO*	Backward	< 0.1	< 0.1	0.155	< 0.1	$\simeq$0.1	0.320
MBLAO*	Speedup	2.06	1.56	1.28	1.39	3.32	2.99
MBLAO*	Backward	0.126	0.148	0.206	< 0.1	$\simeq$0.1	0.458

Table 5.7: Speedups achieved against LAO* and percentage of backups from the backward searches ($\delta = 10^{-6}$)

From Table 5.7 we see that for both BLAO* and MBLAO*, the backward searches occupied a very small portion of the entire backups performed. This means forward searches are the main focus of the bidirectional search algorithms since the majority of the time was spent on forward search. The backups performed during the backward searches help

improve the usefulness of backups done in the forward search; the time spent in backward searches is well worth it, given the consequent performance gain. In Table 5.7, we can see that the backward searches never performed over 0.5% of the backups done by both searches, yet the best speedup bidirectional search achieved was more than a factor of three.

Another interesting observation is that the accuracies of initial heuristic values of random MDP problems were around 10% lower than for the other problems. This is because random MDP problems are often highly nondeterministic, so that the least costly estimates are sometimes too optimistic. Coincidentally, BLAO* and MBLAO* achieved the best speedups on random MDP problems. This fact leads to our consideration about what types of problems BLAO* and MBLAO* are best for. The basic function of the backward search is to propagate and refine value functions. Thus, the worse the heuristic estimates are, the more profitable the backward search can be in the first few iterations. This is because, during the backups performed by the backward searches, the value improvement space is larger. On the other hand, the ability of the forward search to detect this improvement is limited when the forward search frontier has not reached the part with good heuristic estimates. Furthermore, crude initial heuristics also mean we might have to consider a larger number of branches, some of which are suboptimal, before the value functions converge. Therefore, the backward searches in these types of problems are multi-functional.

Chapter 6

Priority-based Algorithms

Heuristic search with the help of reachability analysis is not the only approach to expedite the convergence of the states. Another solution is to generate an order of state backups, or to update value functions of states according to some priority functions, so that it takes fewer iterations for the value functions to converge than backing up states in an arbitrary order. Priority-based algorithms is another area we have been working on. In this section, we discuss three priority-based algorithms as well as our work.

6.1 Prioritized Sweeping

The prioritized sweeping (PS) algorithm was originally a reinforcement learning algorithm [21]. It has been extended to a dynamic programming technique to solve MDPs by using the same prioritization method on state backups. In a goal-based MDP, we usually start from the goal state and sweeps towards the initial state. The algorithm keeps a priority queue to determine when to back up a state. Initially, the priority queue only contains the goal state. At each step, PS pops a state from the queue with the highest priority and backs up all its predecessors, all the predecessors whose Bellman residual are greater than some threshold value are inserted into the priority queue, where the priorities of these states are their Bellman residuals. PS tends to pay more attention to the state space where the maximum potential changes in the value functions can be done.

6.2 Improved Prioritized sweeping

Improved Prioritized sweeping (IPS) [20] is an improved version of the prioritized sweeping based dynamic programming algorithm. In IPS, three possible ways were proposed to generalize the priority computation so that it works for general positive-cost MDPs:

- Changes in value, meaning the larger the change in value function, the higher the

priority value. However, if the value functions are arbitrarily initialized, sometimes the great changes in value only indicates how inaccurate the initial value function is;

- Low upper bound on value, implemented by scaling the value function by a newly-introduced monotone increasing function $m(\cdot)$ on value functions. It is more advanced than changes in value because it is useful regardless of how value functions are initialized;

- Probability of reaching the goal, meaning the higher the probability of reaching the goal, the higher the priority.

By trial and error, McMahan and Gordan found that a priority function combining the last two approaches works well for racetrack MDPs [1].

6.3 Focussed Dynamic Programming

Focussed Dynamic Programming [13] combines ideas from deterministic search and dynamic programming methods. It is named so because the algorithm puts efforts into only those states that are able to reach the goal states. Focussed dynamic programming attempts to restrict the set of expanded states to those that are definitely necessary for optimal solution construction, as well as paying attention to the order of updating states in dynamic programming.

Following the Focussed Dynamic A* (D*) algorithm [25], focussed dynamic programming does expansions in the backward manner, from the goal state towards the start state. All states are initially assigned infinite values, and the value of any state at any time is an upper bound of the state's optimal value. The algorithm maintains a priority queue of states to be expanded, ordered by increasing the summation of the heuristic cost from the start state to s, $p(s)$, and the heuristic cost from s to the goal $q(s)$. When a state s is popped off the queue, each of its predecessors is backed up. The Bellman residual is calculated and compared with a threshold value β. If it is greater than β, this predecessor is inserted back to the queue with its new $p+q$. The termination condition of this algorithm is that the lowest key value of the queue is greater than the start state's value.

6.4 Topological Value Iteration

We proposed a new prioritized based algorithm named Topological Value Iteration (TVI) [12]. TVI is based on our observation that the values of an MDP are dependent on each other. In an MDP M, if state s' is a successor state of s after applying an action a, then $V(s)$ is dependent on $V(s')$. For this reason, we want to back up s' before s. We can regard value dependency as causal relation over their designated states. Since MDPs are cyclic, the causal relation can be cyclic and therefore quite complicated. The idea of TVI is the following: We group states that are mutually causally related together and make them a *metastate*, and let these metastates form a new MDP M'. Then M' is no longer cyclic. In this case, we can back up metastates in M' according to their reverse topological order. In other words, we can back up these big states in only one *virtual* iteration.

To find those mutually causally related states, we studied the graphical structure of an MDP. An MDP M can be regarded as a directed graph $G(V, E)$. The set V in G has two kinds of nodes. The first is state nodes, and each node represents a state in the system. The second is action nodes, and every action of the MDP is mapped to a vertex in the graph. The edges, E, in the graph are used to represent causal relations in M. If there is an edge e from a state node s to an action node a, this means a is an applicable action of s. Conversely, an edge e pointing from a to s' means if we apply action a, the system has a positive probability of changing to state s'. For such M and G, if we can find certain path $s \rightarrow a \rightarrow s'$, we know that state s is causally dependent on s'. So if we simplify G by removing all the action nodes, and changing paths like $s \rightarrow a \rightarrow s'$ into a directed edge from s to s', we can get a causal relation graph G_{cr} of the original MDP M. If there is a path from state s_1 to state s_2, then we know s_1 is causally dependent on s_2. Thus the problem of finding mutually causally related groups of states can be reduced to the problem of finding the strongly connected components in G_{cr}.

We use to Kosaraju's algorithm [10] of detecting the topological order of strongly connected components in a directed graph. Note that in HDP, Bonet and Geffner [6] used Tarjan's algorithm to detect strongly connected components in a directed graph, but they

do not use the topological order of these components as a clue to systematically back up each component. Rather, they use the same marking strategy as in LRTDP [7] to mark all the solved connected components and regard states in the solved components as "tip" states.

The pseudocode of topological value iteration algorithm is shown in Algorithm 6. We first use Kosaraju's algorithm to find the set of strongly connected components $c \in C$ in graph G_{cr}, and their topological order. Note that each c here maps to a set of states in the original MDP M. We then apply value iteration to solve each $c \in C$. Since there are no cycles among those components, we can apply value iteration only once on each component.

Algorithm 6: TVI

Input: S, A, γ, δ
change the MDP into a directed graph G_{cr}
run Kosaraju's algorithm on G_{cr} and get a topological order O on all the SCCs in G_{cr}
while $O \neq \emptyset$ **do**
  $scc \leftarrow$ the first SCC in O
  $O \leftarrow O - \{scc\}$
  Value Iteration(scc, δ)
end while
return $V(\cdot)$, $\pi(\cdot)$

6.5 Experiments

We tested TVI on two types of problem domains. The first domain is a model simulating PhD qualifying exams. Consider the following scenario from a fictional CS department: To be qualified for a PhD student in Computer Science, one has to pass exams in each CS area. Every two months, the CS department offers exams in each area as long as there are students participating. Each student is free to take each exam as many times as he wants as long as he has not passed that exam. Each time, one student can take at most two exams. We consider two types of grading criteria. The first criterion is quite simple. For each exam, we only have pass and fail (and of course, untaken). Students who have not taken the exam and who have failed the exam before have the same chance of passing that exam. The second

criterion is a little trickier. We assign pass, conditional pass, and fail to each exam, and the probabilities of passing certain exams vary, depending on the student's most recent grade on that exam. We consider a state in this domain as a value assignment of the grades of all the exams. For example, if there are five exams, $\langle fail, pass, pass, condpass, untaken \rangle$ can be one possible state. We refer to the first criterion MDPs as $QE_s(x)$ and second as $QE_t(x)$, where x refers to the number of exams.

The second domain is artificially-generated "layered" MDPs. The rule of generating these MDPs are the following: For each MDP, we define the number of states, and partition them evenly into a number n_l of *layers*. These layers are numbered by numerical values. We allow states in higher numbered layers to be the successor states of states in lower numbered layers, but not vice versa, so each state has only a limited set of allowable successor states $succ(s)$. The other parameters of these MDPs are the same as random MDPs (discussed in Section 5.3.5). The advantage of generating MDPs this way is that these layered MDPs contain at least n_l connected components and the size of the biggest component is at most $|S|/n_l$. We believe there are layered MDPs that code actual applications. For example, in some role-playing games a character has to accomplish a number of subgoals in order to finally achieve an ultimate goal, and the MDPs of these games are layered.

algorithm	$QE_s(7)$	$QE_s(8)$	$QE_s(9)$	$QE_s(10)$	$QE_t(5)$	$QE_t(6)$	$QE_t(7)$	$QE_t(8)$
$\|S\|$	2187	6561	19683	59049	1024	4096	16384	65536
$\|a\|$	28	36	45	55	15	21	28	36
$v*(s_0)$	11.129919	12.640260	14.098950	15.596161	7.626064	9.094187	10.565908	12.036075
h_min	4.0	4.0	5.0	5.0	3.0	4.0	4.0	5.0
VI($h=0$)	1.08	4.28	15.82	61.42	0.31	1.89	10.44	59.76
LAO*($h=0$)	0.73	4.83	26.72	189.15	0.27	2.18	16.57	181.44
LRTDP($h=0$)	0.44	1.91	7.73	32.65	0.28	2.05	16.68	126.75
HDP($h=0$)	5.44	75.13	1095.11	1648.11	0.75	29.37	1654.00	2130.87
TVI($h=0$)	0.42	1.36	4.50	15.89	0.20	1.04	5.49	35.10
VI(h_{min})	1.05	4.38	15.76	61.06	0.31	1.87	10.41	59.73
LAO*(h_{min})	0.53	3.75	19.16	126.84	0.25	1.94	14.96	123.26
LRTDP(h_{min})	0.28	1.22	4.90	20.15	0.28	1.95	16.22	124.69
HDP(h_{min})	4.42	59.71	768.59	1583.77	0.95	30.14	1842.62	2915.05
TVI(h_{min})	0.16	0.56	1.86	6.49	0.19	0.98	5.29	30.79

Table 6.1: Problem Statistics and convergence time in CPU seconds for different algorithms with different heuristics ($\delta = 10^{-6}$)

We considered several variants of our first domain, and run VI, LAO*, LRTDP, and TVI on them. The results were shown in Table 6.1. The statistics show the following:

n_l	20	40	60	80	100	200	300	400	500	600
VI($h=0$)	19.482	17.787	12.310	24.538	31.684	40.782	46.554	52.248	64.706	77.658
LAO*($h=0$)	6.088	6.584	5.702	7.297	9.250	12.801	12.463	12.259	16.255	21.991
LRTDP($h=0$)	9.588	9.651	8.483	8.352	11.165	11.108	13.125	13.443	14.160	22.057
TVI($h=0$)	2.563	2.686	2.587	2.624	2.805	3.186	3.061	4.065	4.264	5.131

Table 6.2: Problem statistics and convergence time in CPU seconds for different algorithms on solving artificially generated layered MDPs with different number of layers ($|s|$=20000, ma=10, ms=20, $\delta = 10^{-6}$)

$\|S\|$	10000	20000	30000	40000	50000	60000	70000	80000
VI($h=0$)	9.322	27.683	38.529	46.178	64.915	72.087	95.479	117.359
LAO*($h=0$)	3.056	6.201	10.561	13.663	21.409	23.461	29.248	38.746
LRTDP($h=0$)	3.903	8.814	10.725	20.708	27.156	53.773	49.954	50.730
TVI($h=0$)	1.212	2.589	4.055	5.571	7.133	8.626	10.307	11.969

Table 6.3: Problem statistics and convergence time in CPU seconds for different algorithms on solving artificially generated layered MDPs with different state space (n_l=20, ma=10, ms=20, $\delta = 10^{-6}$)

- The TVI algorithm outperformed the rest of the algorithms in all our instances, with different heuristic values. Generally, this fast convergence is due to both the appropriate update sequence of the state space and avoidance of unnecessary updates.
- The h_{min} heuristic [6] helped the convergence of TVI more than it helped VI, LAO* and LRTDP, especially in the QE_t domains.
- TVI outperformed HDP, which also uses connected components. This is because the ways of dealing with components of two algorithms are different. HDP updates states of all the unsolved components together in a depth-first fashion until they all converge. We pick the optimal sequence of backing up each component. We only back up one of them at a time. Moreover, our algorithm does not spend time checking whether all the components are solved, and we only update a component when it is necessary.

The statistics of the performance on layered MDPs were included in Table 6.2 and Table 6.3. For each element of the table, we smoothed it by taking the average of running 20 instances of MDPs with the same configuration. Note that varying $|s|$, n_l, m_a, and m_s can yield huge number of MDP configurations, but only a few representatives are picked.

For the first experiment, we fixed the state space to be 20,000 and changed the number of layers. Statistics in Table 6.2 show our TVI dominated others. Notice that, as the layer number increased, the MDPs became more complex, since the states in large numbered layers have relatively small $succ(s)$, and therefore cycles in those layers were probably more common, so it took greater effort to solve large numbered layers than small numbered ones. The results also displayed a tendency that when the number of layers increased, the running time of each algorithm also increased. However, the increase rate of TVI was the smallest (the greatest against smallest running time is 2 of TVI in Table 6.2 versus 4 of VI, 3.5 of LAO*, and 2.3 of LRTDP). This was due to the fact that TVI applied the best update sequence. As the layer number became large, although the update of the large numbered layers required more effort, the time spent on the small numbered ones remained stable.

For the second experiment, we fixed the number of layers and let the state space size vary. Again, TVI was better than other algorithms as seen in Table 6.3. When the state space was 80,000, TVI could solve the problems in around 12 seconds. This indicated that TVI can solve large problems in a reasonable amount of time. Note that the statistics we include here represent the common cases, but were not chosen intentionally in favor of TVI. Our best result was that, TVI converged in 9 seconds for MDPs with $|S|$=20,000, n_l=200, na=20, ns=40, while VI needed more than 100 seconds, LAO* took 61 seconds and LRTDP required 64 seconds.

Chapter 7

Conclusion and Future Work

Markov decision processes are a useful tool in representing decision theoretic planning problems. Classic dynamic programming algorithms such as value iteration and policy iteration usually converge quite slowly on big problems. This paper introduces several efficient new algorithms in expediting the convergence of dynamic programming approaches.

Multi-threaded BLAO*, a family of bidirectional heuristic search algorithms, extends the single-source backward search of BLAO* into a number of backward searches with different sources. It takes the advantage of the fact that backups performed in the backward search help propagate heuristic values from states whose heuristics are more accurately initialized. Experimental results have provided evidence that MBLAO* converges faster than not only BLAO*, but several state-of-the-art forward heuristic search algorithms.

Topological value iteration, a simple priority-based algorithm, is flexible, since it studies the graphical structure of an MDPs and makes full use of it. TVI solves MDPs by discovering the strongly connected components in the directed graph representation of an MDP and prioritizing dynamic programming on the components by their topological order. It has been shown to be especially useful on MDPs with evenly distributed strongly connected components.

We believe that heuristic search and priority-based approach are very promising research topics in AI planning. One of our ongoing research project is on using graphical structure to expedite the convergence time in reinforcement learning MDP algorithms [26]. Apart from regarding the two topics individually, an integration of heuristic search and prioritization is also very interesting. Actually, focussed dynamic programming [13] can be regarded as a combination of both. In the future, we plan to dig deeper along this path. We also think these two strategies can be used in combination with other common techniques such as factored MDPs, value approximation, and linear programming.

Bibliography

[1] A.G. Barto, S.J. Bradke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence J.*, 72:81–138, 1995.

[2] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[3] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 2000-2001.

[4] Kiran Bhuma. Bidirectional LAO* algorithm (a faster approach to solve goal-directed MDPs). Master's thesis, University of Kentucky, Lexington, 2004.

[5] Kiran Bhuma and Judy Goldsmith. Bidirectional LAO* algorithm. In *Proc. of Indian International Conferences on Artificial Intelligence (IICAI)*, pages 980–992, 2003.

[6] B. Bonet and H. Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proc. of 18th International Joint Conf. on Artificial Intelligence (IJCAI-03)*, pages 1233–1238. Morgan Kaufmann, 2003.

[7] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. 13th International Conf. on Automated Planning and Scheduling (ICAPS-03)*, pages 12–21, 2003.

[8] Craig Boutilier. Planning, learning and coordination in multiagent decision processes. In *Conference on Theoretical Aspects of Rationality and Knowledge*, pages 195–201, 1996.

[9] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *J. of Artificial Intelligence Research*, 11:1–94, 1999.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[11] Peng Dai and Judy Goldsmith. LAO*, RLAO*, or BLAO*? In *AAAI Workshop on Heuristic Search*, pages 59–64, 2006.

[12] Peng Dai and Judy Goldsmith. Topological value iteration algorithm for Markov decision processes. In *Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 1860–1865, 2007.

[13] Dave Ferguson and Anthony Stentz. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[14] Eric Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence J.*, 129:35–62, 2001.

[15] Eric A. Hansen and Shlomo Zilberstein. Heuristic search in cyclic AND/OR graphs. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 412–418, 1998.

[16] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[17] R.A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.

[18] Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proc. of the 17th national conference on Artificial intelligence (AAAI-00)*, pages 910–916, 2000.

[19] Michael L. Littman, Thomas Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proc. of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 394–402, Montreal, Quebec, Canada, 1995.

[20] H. Brendan McMahan and Geoffrey J. Gordon. Fast exact planning in Markov decision processes. In *Proc. of the 19th International Joint Conference on Planning and Scheduling (ICAPS-05)*, 2005.

[21] Andrew Moore and Chris Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *J. of Machine Learning*, 13:103–130, 1993.

[22] Nils J. Nilson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, Ca., 1980.

[23] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley, New York, 1994.

[24] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[25] Anthony Stentz. The focussed D* algorithm for real-time replanning. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1652–1659, 1995.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[27] David Wingate and Kevin D. Seppi. Prioritization methods for accelerating MDP solvers. *J. of Machine Learning Research*, 6:851–881, 2005.

Vita

1. Background.

   (a) Date of Birth: Feb. 1st, 1979

   (b) Place of Birth: Nanjing, China

2. Academic Degrees.

   (a) M.S., June 2004

   School of Computing, National University of Singapore, Singapore

   (b) B.S., June 2001

   Computer Science Department, Nanjing University, Nanjing, China

3. Professional Experience.

   (a) Software Engineer, Alcatel Shanghai Bell, No.388 Ningqiao Road, Pudong, Jinqiao, Shanghai, China

4. Professional Publications.

   (a) Peng Dai and Judy Goldsmith. Multi-threaded BLAO* Algorithm. In *Proc. 20th International FLAIRS Conference*, pages 56-62, 2007.

   (b) Peng Dai and Judy Goldsmith. Topological Value Iteration Algorithm for Markov Decision Processes. In *Proc. of 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages1860-1865, 2007.

   (c) Peng Dai and Judy Goldsmith. LAO*, RLAO*, or BLAO*?. In *Proc. of AAAI Workshop on Heuristic Search*, pages 59-64, 2006.