2004

Bidirectional LAO* Algorithm (A Faster Approach to Solve Goal-directed MDPs)

Venkata Deepti Kiran Bhuma
University of Kentucky, deeptikiran@uky.edu

ABSTRACT OF THESIS

Bidirectional LAO* Algorithm
(A Faster Approach to Solve Goal-directed MDPs)

Uncertainty is a feature of many AI applications. While there are polynomial-time algorithms for planning in stochastic systems, planning is still slow, in part because most algorithms plan for all eventualities. Algorithms such as LAO^* are able to find good or optimal policies more quickly when the starting state of the system is known.

In this thesis we present an extension to LAO^*, called $BLAO^*$. $BLAO^*$ is an extension of the LAO^* algorithm to a bidirectional search. We show that $BLAO^*$ finds optimal or ϵ-optimal solutions for goal-directed MDPs without necessarily evaluating the entire state space. $BLAO^*$ converges much faster than LAO^* or $RTDP$ on our benchmarks.

KEYWORDS: Planning under uncertainty, Markov decision processes, heuristic search, Decision-theoretic planning, bidirectional search.

Venkata Deepti Kiran Bhuma

Date

Bidirectional LAO* Algorithm
(A Faster Approach to Solve Goal-directed MDPs)

By

Venkata Deepti Kiran Bhuma

Dr. Judy Goldsmith
(Director Of Thesis)

Dr. Grzegorz W. Wasilkowaski
(Director Of Graduate Studies)

Date

RULES FOR THE USE OF THESES

THESIS

Venkata Deepti Kiran Bhuma

The Graduate School
University of Kentucky
2004

Bidirectional LAO* Algorithm
(A Faster Approach to Solve Goal-directed MDPs)

THESIS

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

By

Venkata Deepti Kiran Bhuma

Lexington, Kentucky

Director: Dr. Judy Goldsmith, Associate Professor of Computer Science

Lexington, Kentucky

2004

Acknowledgments

First, I thank my Thesis Chair, Dr. Judy Goldsmith, for her continuous guidance and support. She has extended her support during the period of research as well as throughout my Master's studies. She stood beside me in each and every stage of my research. Her confidence in me has made this work possible.

I thank my parents for providing me this opportunity to pursue higher studies at this university and to make my dreams come true. I thank Dr. Hansen and Dr. Zilberstein for providing their code for my research. I thank my friends and my roommates who were extremely helpful and understanding during the crucial periods of my research and for being patient to my frustrations I showed on them during those periods. I thank all the professors at the University of Kentucky for their ardent support in completion of my research and Master's studies at the University of Kentucky. I also thank the staff members of Computer Science Department, my colleagues and seniors for their direct or indirect support during my stay here and getting accommodated to this environment.

Table of Contents

List of Tables

List of Figures

List of Files

1. VBThesis.pdf 795KB

Chapter 1

Introduction

Planning under uncertainty is necessary in many artificial intelligence applications. Artificial Intelligence is the science of making machines, called agents, do tasks that humans can do or try to do. In real-world situations, most of the time an agent is not provided with complete data or the supplied data is not sufficient enough to make decisions. When the data is incomplete or when the agent has multiple outcomes for its each action, the results of each action are to be compared to select a best one. The framework of decision theory with uncertainty, called decision theoretic planning, can be used to compare or evaluate plans and actions of an agent.

A Markov Decision Process (MDP) is a model of a stochastic dynamical system, with costs or rewards associated with system states and action choices. MDPs model problems of decision making, where an action might transform a state into one or many possible successor states, with some probability of occurrence for each state transition. The uncertainty in the choice of possible next states arises because the agent may have incorrect information about the current state of the system or incomplete access to the true state of the system.

Solutions to MDPs are policies. A policy is a mapping from the set of states to the set of actions. A goal-directed MDP is an MDP which contains a set of states called goal states, which are preferred to other states. A Solution to a goal-directed MDP is policy that causes the agent to reach a goal state with high probability.

One method for finding optimal policies for MDPs is dynamic programming. Dynamic programming has two approaches: Policy Iteration and Value Iteration [6]. The basic disadvantage of dynamic programming is that each of potentially many iterations performs a re-evaluation of every state in the potentially very large state space. Real-Time Dynamic Programming ($RTDP$) [1] can also be used to find an optimal policy without necessarily evaluating the entire state space, but the

convergence of *RTDP* is slow.

Another approach to solving MDPs is to construct a policy by exploring only those states that are reachable from the start state. For deterministic systems, one can use algorithms such as A^* [11]. When the system forces the planner to consider multiple outcomes in order to evaluate each node (as the AND of its successors), the algorithm AO^* may be used [11]. However, MDPs are more complex: Each of multiple transitions may lead to multiple possible states, including some that have already been visited. The underlying state-space graph can be modeled as an AND/OR graph [10] (described in Chapter 4); Hansen and Zilberstein have extended the $A0^*$ algorithm to LAO^*, an algorithm that works on exactly such graphs [8]. LAO^* is most directly comparable with Real Time Dynamic Programming. Hansen and Zilberstein gave a family of MDPs for which LAO^* is significantly faster than *RTDP*. We present here an extension of LAO^*, $Bidirectional LAO^*$, ($BLAO^*$) which outperforms LAO^* on those MDPs. Much of the work discussed here was presented at *IICAI'03* [4].

The remainder of the thesis is organized as follows:

Definitions: In this chapter, we introduce the terminology that we use in the rest of this thesis.

Dynamic Programming methods: Here, we describe general methods to solve MDPs. In particular, we explain Policy Iteration, Value Iteration and Real-Time Dynamic Programming methods.

AND/OR graphs and the LAO* algorithm: In this chapter, we focus on the search algorithms that solve the basic stochastic shortest-path problems and how they are extended to solve MDPs. We explain the structure of AND/OR graphs and LAO^* algorithm that solves MDPs.

BLAO*: In this chapter, we present $BLAO^*$. The algorithm and the implementation details are discussed. We prove that $BLAO^*$ converges in a finite number of steps and finds an *optimal* or ϵ-*optimal* solution for goal-directed MDPs. Furthermore, $BLAO^*$ converges without necessarily evaluating the entire state space.

Examples: Here we describe various problems that can be used as benchmarks for testing and comparing the performance of various algorithms, $RTDP$, LAO^* and $BLAO^*$. Initially we used the race-track examples described in [1] as our test cases to compare the performances of LAO^* and $BLAO^*$. Then, we developed random Markov Decision Processes which are further

used as test cases to compare the performances of LAO^* and $BLAO^*$. The generation of these examples is discussed in this chapter.

Analysis: Here we present the results of experimental comparisons of the algorithms $RTDP$, LAO^* and $BLAO^*$ on several race-track examples and randomly generated MDPs. We show how these results validated our intuition.

Conclusions: In this section we describe future opportunity of research. We also describe the implementation of $BLAO^*$ to solve factored MDPs.

Chapter 2

Markov Decision Processes

A Markov Decision Process [2] models a stochastic dynamical system, where the system can be in one of a number of distinct states at each discrete time step and the system's state changes over time in response to actions. Actions are chosen to optimize the costs or rewards associated with state-action pairs.

State: A state is a description of the system at a particular time. We assume that the set S of states is finite.

Action: An action is an event performed by an agent in order to change the system's state. The choice of an action is under the control of the agent but the results of taking an action may be uncertain. We assume that the set A of actions is also finite. We denote the set of possible actions for state i as $A(i)$.

State transitions: State transitions describe how the next state of the system depends on the past state and action taken. State transitions can be represented by transition matrices. A transition matrix $p(a)$ describes how the system evolves in time when an action a is taken on any state. That is, $p_{ij}(a)$ denotes the probability of ending in state j when an action a is taken in state i.

Cost: The cost function is a mapping from the set of pairs of states and actions to $\mathbf{R}$, that is, $C : S \times A \to \mathbf{R}$. Thus, a cost $C_i(a)$ is incurred whenever an action a is taken in state i. We assume that cost function cannot take negative values.

Policy: A policy π is a mapping from the set of states S to the set of actions A. That is, $\pi : S \to A$.

This paper focuses on a special class of MDPs whose objective is to reach some goal states where an assumption is made that goal states can be reached from every other state. The cost of taking any action on goal states is 0 and $p_{ii}(a) = 1$ for any action a and every state i which is a goal. An optimal policy minimizes the expected cost of reaching a goal state from each start state.

Horizon: A horizon specifies the number of stages actions have to be executed.

Value function: The value function f^π for a policy π is a mapping from the set of states to $\mathbf{R}$. That is, $f^\pi : S \rightarrow \mathbf{R}$. It is defined as [2]

$$f^\pi(i) = \begin{cases} 0 & \text{if } i \text{ is a goal state;} \\ C_i(\pi(i)) + \sum_{j \epsilon S} p_{ij}(\pi(i)) f^\pi(j) & \text{otherwise.} \end{cases}$$

A policy π' dominates a policy π if

$$f^{\pi'}(i) \leq f^\pi(i) \text{ for every state i,}$$

where f^π denotes a value function under a policy π and is defined below. An optimal policy, π^*, dominates every other policy.

There is an optimal policy for any MDP [9]. The value function for any optimal policy is called the optimal value function, denoted f^*. It holds that [2]:

$$f^*(i) = min_{a \epsilon A(i)}[C_i(a) + \sum_{j \epsilon S} p_{ij}(a) f^*(j)].$$

Chapter 3

Dynamic Programming Methods

Solving an MDP means finding an optimal or ϵ-optimal (explained in Section 3.2.1) policy that minimizes the value of each state. In this section we review standard algorithms for solving MDPs. A general method to solve MDPs is by using dynamic programming techniques based on Bellman's *principle of optimality* [2]. There are two approaches to dynamic programming: Value Iteration and Policy Iteration.

3.1 Policy Iteration

Policy iteration modifies the policy under construction explicitly, rather than improving the value function in each iteration [9]. An arbitrary policy π is used initially. Each iteration of Policy Iteration consist of two phases, the *evaluation phase* and the *improvement phase*.

1. *Evaluation phase*: In iteration $t+1$, compute the value function for all states i under the current policy π as
$$f^{t+1}(i) = C_i(\pi(i)) + \sum_{j \epsilon S} p_{ij}(a) f^t(j).$$

2. *Improvement phase*:

   (a) For each state $i \epsilon S$, find an action a such that
   $$C_i(a) + \sum_{j \epsilon S} p_{ij}(a) f^{t+1}(j) \text{ is minimized.}$$

   (b) Replace $\pi(i)$ with this a.

Repeat the above two steps until $\pi(i)$ does not change for any state $i \epsilon S$. There are only $|S|^{|A|}$ possible policies and each next policy improves the value, so the value iteration converges in less than or equal to $|S|^{|A|}$ iterations.

3.2 Value Iteration

In this method, the value function f is evaluated and improved in each iteration until it converges to an optimal or ϵ-optimal value [2]. In the first iteration of Value Iteration, assign arbitrary values to the value function f for each state i. Then in each iteration, evaluate the new value function f^{t+1} by improving the old value function f^t on each i as follows:

$$f^{t+1}(i) = min_{a \epsilon A(i)}[C_i(a) + \sum_{j \epsilon S} p_{ij}(a) f^t(j)]. \tag{3.1}$$

If the error bound of the evaluation function (defined in Section 3.2.1) is less than or equal to ϵ

1. then, extract an optimal or ϵ-optimal policy from the current value function using the following equation:
$$\pi(i) = argmin_{a \epsilon A(i)}[C_i(a) + \sum_{j \epsilon S} p_{ij}(a) f^{t+1}(j)], \tag{3.2}$$
2. else evaluate and improve the value function (using equation 3.1) until the error bound is less than or equal to ϵ.

3.2.1 Calculation of Error Bound

A *stochastic shortest-path problem* is defined in [3] as an MDP with a set of goal states. All goal states are absorbing states, that is, no action can cause a transition out of a goal state, and the cost of taking an action on a goal state is zero. An error bound for stochastic shortest-path problems is computed based on the *Bellman residual*. The *Bellman residual* at state $t+1$ is

$$r = max_{i \epsilon S} |f^{t+1}(i) - f^t(i)|.$$

The *mean first passage time* [3] for a state i is the expected number of time steps required to reach a goal state from a state i under the current policy π. It is denoted $\psi^\pi(i)$. It can be computed by solving the following system of linear equations for all $i \epsilon S$ [8]:

$$\psi^\pi(i) = 1 + \sum_{j \epsilon S} p_{ij}(\pi(i)) \psi^\pi(j).$$

A lower bound, LB, on the optimal value function [8] for each state i is:

$$LB(i) = f^{\pi}(i) - \psi^{\pi}(i)r.$$

An upper bound, UB, on the optimal value function [8] for each state i is:

- $UB(i) = f^{\pi}(i) + \psi^{\pi}(i)r$ for value iteration, and
- the current evaluation function for policy iteration.

An error of the current evaluation function is computed as the maximum difference between the upper and lower bounds for a state i, where $i \epsilon S$. If the error bound is less than ϵ then the solution is said to be ϵ-*optimal*.

3.3 Real-Time Dynamic Programming

The focus of our research is solving stochastic shortest-path problems. Our objective is to find a solution that minimizes the expected cost of reaching a goal state from the start state. *Policy Iteration* and *Value Iteration* give policies that maximize the probability of reaching some goal state from each state. The dynamic programming approaches do not take advantage of the notion of start state and must therefore evaluate the entire state space.

Real-Time Dynamic Programming (RTDP) was introduced by Barto et al. in [1]. *RTDP* takes advantage of the notion of start state and focuses on only those nodes that are reachable from the start space. *Trial-based RTDP* solves an MDP by performing a sequence of n simulated trials, where each trial consist of a sequence of m of steps. In each step, an action is chosen based on lookahead search and the estimated value of the current state is updated based on the value of the successor states visited as a result of the action. Thus, the value function is updated for only those states that are reachable from the start state.

Initially, the value function is set to some admissible heuristic function. Each trial starts at the start state and ends when a goal state is reached, or after m steps, whichever comes first. In each step, the value function f for a state i is updated as

$$f^{t+1}(i) = min_{a \epsilon A(i)}[C_i(a) + \sum_{j \epsilon S} p_{ij}(a) f^t(j)],$$

for those i reachable from the start state.

For each state i, an action that minimizes the above equation is performed and its successor states are visited. One of the successor states that is visited is made the current state and the trial

is continued until either a goal state is visited or m steps are executed. In a *trial-based RTDP* n such trials are executed, where n is an input parameter. After n trials, a policy is extracted from the value function f for each state i as

$$\pi'(i) = argmin_{a \epsilon A(i)}[C_i(a) + \sum_{j \epsilon S} p_{ij}(a) f^n(j)].$$

Thus, RTDP finds an optimal policy and ignores large regions of the state space whenever the set of states reachable from the start state is small. Unfortunately, RTDP is still impractically slow.

Chapter 4

AND/OR graphs and LAO^* Algorithm

The objective of goal-directed MDPs is to assign actions to states so that, with reasonably high probability, the start states are transformed into goal states while optimizing the expected quality of the solution.

The algorithms A^* and AO^* are used for state-space search. A^* finds a solution for a deterministic planning problem that takes the form of a sequence of actions giving a path from the start state to a goal state [10]. AO^* finds a solution that has a conditional structure and takes the form of an acyclic graph [11]. An extension of AO^* called LAO^* can apply similar planning to more complex underlying structures, namely, AND/OR graphs with loops [8]. Our work extends LAO^*.

4.1 AND/OR graphs and AO^* algorithm

AND/OR graphs were first described by Martelli and Montanari [10]. Hansen and Zilberstein extended the definition to allow the underlying graph to be cyclic; they described these structures as hypergraphs [8]. A hypergraph is a graph which also has *hyperarcs* (or simply *arcs*), each of which connects a node to a set of successor nodes. An AND/OR graph is useful for representing problems that can be solved by decomposing them into sets of smaller problems, all of which must then be solved in order to solve the larger problem. A node in an AND/OR graph is said to be *solved* if all of its successors are solved. To solve a node, all of its successors must be visited and its value must be updated based on the values of its successors. Note that one AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a

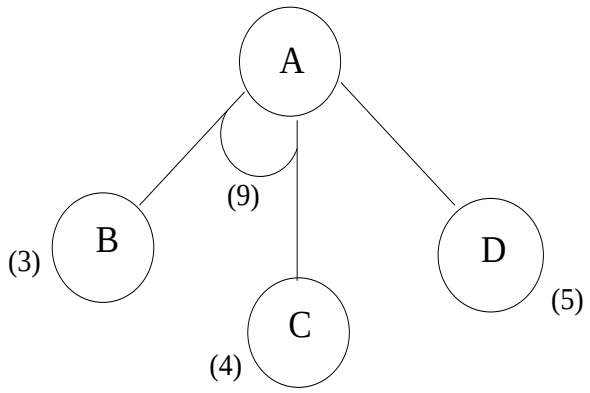

Figure 4.1: An example of AND/OR graph. AND arcs are connected by an arc. The arc connecting $\langle A, B\rangle$ and $\langle A, C\rangle$ is an AND arc.

solution. Several arcs may emerge from a single node, indicating a variety of ways in which the original problem can be solved (this is the OR part of the graph).

An example of an AND/OR graph is given in Figure 4.1. AND arcs are indicated with a line connecting all the components. Let us assume, for simplicity, that each arc with a single successor has a cost of 1 and each AND arc with multiple successors has a cost of 1 for each of its components. The cost of a node is the minimum over all arcs from that node of the cost of the arc, plus the cost of the child or children (if it's an AND arc) of that arc. The cost of a simple arc is 1, and the cost of an AND arc is the number of simple arcs that are joined in the AND.

Nodes whose children are not yet expanded, leaf nodes, are evaluated using a heuristic, as discussed in the next paragraph. In the example, the node A is expanded producing two arcs, one leading to B and C and the other leading to D. The numbers at each node represent the value function, f, for that node. The value function is set to an admissible heuristic function initially. The cost of node A depends on the cost of its successors. The cost of node A is 9 if the path leading through B and C is chosen, else the cost of node A is 6 if the path through D is chosen. If we are interested in finding a least-cost path then node D is chosen for further expansion, else both B and C nodes are chosen for expansion.

The AO^* algorithm takes an *implicit graph* G, a start node, and a successor function as input. It constructs an explicit graph, G', that represents the part of the graph that has been explicitly generated so far. Initially, G' consists of only the start node. Each node in the graph points to its immediate successors. Initially, the value of each node is based on a heuristic estimate h'. $h'(i)$

denotes the estimated cost of a path from node i to a goal node. AO^* is based on an admissible heuristic function. Heuristic functions are discussed in detail in Section 5.

The algorithm begins at the start node and proceeds in steps, expanding one node in each step until a node that corresponds to a goal node is reached. At each step of the algorithm, a node i is selected and an action a, that minimizes the value of the node i, is performed on it. Whenever a node i is expanded, all of its successor nodes corresponding to an action are added to the graph G' and the value of node i is updated based on the value of its successor nodes. The new value of node i is propagated upwards and the ancestors of node i are updated based on the current value of the node i. The action that gives the node i this new value is chosen as the best action for node i. This procedure continues until a goal node is reached and then the values of the leaves are propagated upwards to find an optimal path from the start node to the goal node. AO^* finds an optimal path from the start node to a goal node without evaluating the entire state space.

4.2 LAO*

LAO^* was developed by Hansen and Zilberstein to solve MDPs which take the form of an AND/OR graph with loops [8]. Each node of an AND/OR graph corresponds to a node in an MDP. Each AND arc corresponds to an action which can transform the current state to more than one state. As the probability of each transition in an AND arc is less than one, all the successor states of a state, say i, under an action a, (that are connected by an AND arc) are to be solved in order to solve a state i.

Here is a high-level overview of LAO^*, $RTDP$, and $BLAO^*$: First, a reachability graph, G', on the states is generated and approximate values for those states are calculated. Then Value Iteration is performed *on those nodes* until it converges to an ϵ-approximation of the optimal policy for those nodes. (We assume $\epsilon \geq 0$.) The advantages of this technique over straight Value Iteration are twofold: the state space is potentially reduced by constructing a graph on a subset of the states, and the approximate values constructed in the graph phase of the algorithm lead to faster convergence than starting Value Iteration from scratch.

The input for the LAO^* algorithm is a graph G specified implicitly by start state s and a successor function, determined by the description of the MDP. The algorithm LAO^* builds an explicit graph G' that initially consists of only the start state. A *leaf state* in G' is a node that does not yet have any successors. The graph G' is expanded whenever an action is chosen to perform on a leaf state of the graph. The leaf state is said to be a *terminal state* if it is a goal state, else

it is called a *nonterminal state*. Whenever a leaf state is expanded, exactly one of its outgoing transitions, that is, one action or AND arc, is considered and each of its successor states in that arc are added to the graph. Note that the underlying graph may be cyclic. However, when G' is constructed and an edge is considered that leads to an already-visited node, that node is not added as a leaf. Thus, one can unambiguously refer to a *path* from a start state to a leaf.

In the construction of G', the action that minimizes the total expected cost is chosen for each state and is marked as the best action. Thus, a path from start state to any leaf state is a least-expected-cost path to that state. Whenever a state is expanded, all of its successors are kept on a stack and then they are also expanded. This process is continued until a state that does not have successors, or is a goal state, or already expanded is reached. Expansion of a state might change the value of the state and hence the value of its ancestors, which must then be updated. Thus, every time an action is chosen for state i, the value of each ancestor is recomputed and an action that minimizes the cost is chosen as the best action for that ancestor. This is called *backward propagation*. If the computed best action for an ancestor state is changed, then the current path from start state to state i may change. This process repeats until a state that has no successors, or a state to which an equivalent or better path has already been found, is reached. The value of the state is propagated backwards whenever a path that is better than the current path is found to that state.

Updating the value of a state directly, based on the value of its successors, is called the *cost-revision* step. A^* and AO^* algorithms use cost-revision steps for backward propagation of values. As MDPs contain loops, the backward propagation of values using an A^*-style cost-revision step cannot be applied. Hence, LAO^* was developed for goal-directed MDPs.

Chapter 5

Bidirectional LAO^* Algorithm

Bidirectional LAO^* ($BLAO^*$) is an extension to LAO^* that implements the LAO^* algorithm from the start state and the goal state concurrently. $BLAO^*$ is a heuristic search algorithm that can find optimal solutions without necessarily evaluating the entire state space. $BLAO^*$ converges much faster than LAO^* or $RTDP$ on the race-track benchmarks. But $BLAO^*$ shows only insignificant speedup in performance on random Markov decision processes.

The input to $BLAO^*$ is an implicit graph G given by start state S and a successor function and description of the set of goal states. The output of the $BLAO^*$ algorithm is an explicit graph G' that contains a subset of the state space S consisting of states that are reachable either from the start state or goal state, and an optimal policy which minimizes the expected value of the start state.

Bidirectional LAO* has two searches: forward search and backward search. The forward search starts at the start state and expands the most probable states towards the goal state. Similarly, the backward search starts at the goal state and expands the most probable states backwards towards the start state. If there are sets of goal or start states, we assume equal probability of starting in each member of the set. This assumption is easy to modify in the code.

The forward search uses the forward heuristic and the backward search uses the backward heuristic. The heuristics are described in Section 4.1 The forward search and the backward searches are started concurrently. The forward search is an implementation of the LAO^* algorithm, except for the termination condition, which is explained below. During the backward search, a subgoal is checked for all the actions and states from which the subgoal can be reached. The action from a state i that has the highest probability of reaching the subgoal from i is chosen for i, and then that state i with the highest probability of reaching goal is chosen for further expansion in the

backward direction. Whenever a state is visited, the state is pushed onto a single stack and all of its successors (predecessors or parents in the case of backward search) are considered for further expansion. Whenever the searches end, the states are popped up from the stack and their values are updated, based on the values of their successors (predecessors or parents in the case of backward search). Doing so ensures that only the states that are reachable by the new best actions are updated, instead of updating all the states in the explicit graph G' in each iteration. An array of length $|S|$ is maintained; Whenever a value of a state is updated, its entry in the array is marked so that that state is not updated again in the same iteration. That array keeps track of both forward and backward searches, and is used to check whether a state was already visited in either search.

The forward search from the start state terminates at any state that represents a loop back to an already expanded state, or a goal state, or a nonterminal leaf state, or if the node has already been expanded by the backward search. Similarly, the backward search terminates at any state that represents a loop back to an already expanded state, or a start state, or a nonterminal leaf state, or if the node has already been expanded by the forward search.

Whenever the two searches meet in an iteration, the algorithm checks whether any new nodes were expanded in that iteration. If so, then the searches are repeated because whenever a new node is expanded, the best action or the best solution to any of its ancestors or successors might change. Hence, the iterations are continued until no new node is expanded. When the number of new nodes expanded in an iteration is zero, the solution is checked for convergence. The convergence test is described in Section 4.2 If the solution graph contains any unexpanded leaf state then the searches are repeated. For a given ϵ, the solution converges when the solution graph does not have any nonterminal leaf states and the error bound on the value of the start state is less than ϵ. The solution that is obtained from the *Bidirectional LAO** can be optimal (if $\epsilon = 0$) or ϵ-optimal. The results of the algorithm are explained in Section 6.

*Bidirectional LAO** can be described in more detail as follows :

1. The explicit graph G' initially consists of start state s and goal state g.

2. Forward search and backward search are started concurrently.

3. *Expand and update:* While the best solution has some nonterminal leaf state do the following.

   (a) Expand some nonterminal leaf state s' of the explicit graph G' and check whether any of its successors have already been expanded or whether it is a terminal state. If it has

already been expanded or if it is a terminal state, then go to Step 4, otherwise add the successors to the solution graph.

(b) During forward search, update the value of each state based on the value of its successors that can be reached by performing the best action. During backward search, update the value of the state based on the value of the states from which this state can be reached by performing their respective best actions. Mark the best actions.

4. *Convergence test:* Perform *Value Iteration* on the states in the best solution graph. Repeat until

   (a) the error bound falls below ϵ (go to Step 5), or

   (b) the best current solution contains any unexpanded leaf state (go to Step 3).

5. Return the optimal (or ϵ-optimal) solution graph, G'.

At each stage in the construction of the explicit graph G', G' contains only states that are reachable from the start state. Only the values of the states that are in the explicit graph and that can be reached by marked actions are updated. Thus, the states that are not in the explicit graph, and the states that are in G' and cannot be reached by a series of marked actions do not have their values updated even when the values of their successors or ancestors are changed. $BLAO^*$ converges to an optimal or ϵ-optimal solution without necessarily evaluating the entire state space. The *solution graph* contains the states that are reachable from the start state when their respective best actions are taken.

5.1 Heuristics

Initially, the cost of reaching a goal state from a particular state can be estimated but not determined exactly. A function that calculates such cost estimates is called a *heuristic function.* A heuristic function is usually denoted by h. Thus, $h(n)$ is the estimated cost of the cheapest path from the state n to a goal state. h_M denotes a heuristic function for a given MDP M.

Recall that f^*_M is the optimal value function for a given MDP M. From here on, it is assumed that the MDP is fixed, and we write f^* for f^*_M, and h for h_M. A heuristic evaluation function h for an MDP is said to be *admissible* if $h(i) \leq f^*(i)$ for all i. The explicit graph G' may contain many unexpanded states. The choice of the state that is to be expanded next is nondeterministic. Hansen and Zilberstein proved that, for any admissible heuristic, LAO^* works correctly [8]. Like

A^* and AO^*, the performance of LAO^* and hence the performance of $BLAO^*$ can be improved by choosing a better selection heuristic. A heuristic can be based on reachability from the start state or on expanding a least-cost path to the goal state(s).

Both LAO^* and $BLAO^*$ use the *shortest-path heuristic* as described by Hansen and Zilberstein in [8]. The shortest path heuristic for forward search is computed by beginning from the set of goal states and determining the smallest possible number of backward state transitions in the underlying graph needed to reach each state in the underlying graph. Since the cost of reaching a goal from the state s is bounded by the number of arcs or transitions needed to reach that goal, this is an admissible heuristic. The shortest path heuristic for backward search is computed by beginning from the set of start states and determining the smallest possible number of state transitions needed to reach each state.

5.1.1 Admissibility

Given an initial admissible heuristic evaluation function, all state costs in the explicit graph are admissible after each iteration, and LAO^* converges to an optimal or ϵ-optimal solution without necessarily evaluating all problem states [8]. Simultaneous implementation of LAO^* from start and goal states is the basic principle of $BLAO^*$. Hence $BLAO^*$ also converges to an optimal or ϵ-optimal solution without necessarily evaluating the entire state space. We use $f^k(i)$ to denote the estimated value of state i after k iterations of the algorithm that defines G'.

Theorem 5.1 *[8] If the heuristic function h is admissible, $\epsilon \geq 0$, and Value Iteration is used to perform the cost revision step of LAO^*, then:*

1. *$f^k(i) \leq f^*(i)$ for every state i and every k;*
2. *$f^k(i)$ converges to within ϵ of $f^*(i)$ for every state i of the best solution graph after a finite number of iterations.*

$BLAO^*$ implements LAO^* bidirectionally and uses Value Iteration to update the value of states whenever a state is expanded. Hence the above theorem holds for $BLAO^*$ too.

Theorem 5.2 *If the heuristic functions for both forward and backward search are admissible, $\epsilon \geq 0$ and Value Iteration is used to perform the cost revision step of $BLAO^*$, then:*

1. *$f^k(i) \leq f^*(i)$ for every state i and every k;*

2. *$f^k(i)$ converges to within ϵ of $f^*(i)$ for every state i in the best solution graph after a finite number of iterations.*

Proof (1) The theorem can be proved by induction. We show that each iteration of a search, forward or backward, preserves the admissibility property. We start with the proof for the forward search. The backward search updates nodes in the same tree pattern that we show preserves admissibility in the forward search, so the proof of admissibility of the backward search follows directly.

The heuristic evaluation function h is admissible. Hence, whenever a state i is encountered for the first time in one of the searches, it is assigned an initial heuristic cost estimate $f^0(i) = h(i) \leq f^*(i)$, for the appropriate heuristic function h. Let us assume that $f^t(i) \leq f^*(i)$ for all $i \in S$ and all $t \leq k$. Suppose that for some state l, $f^k(l)$ is updated in the forward search; we will refer to the new value as $f'(l)$ and we compute it as follows:

$$f'(l) = min_{a\epsilon A(i)}[C_i(a) + \sum_{j\epsilon S} p_{ij}(a) f^k(j)] \tag{5.1}$$

$$\leq min_{a\epsilon A(i)}[C_i(a) + \sum_{j\epsilon S} p_{ij}(a) f^*(j)] = f^*(l) \tag{5.2}$$

In the course of iteration $k+1$, $f^{k+1}(i)$ is defined for each reachable state i. The set of reachable states is updated in a reverse search tree order: tip states are updated first, and then their parent states. Although MDPs usually contain loops, the nodes are updated at most twice in each iteration (at most once for each search), in this tree order.

We show that each update maintains the admissibility property. From Equation 5.2, it follows that $f'(i) \leq f^*(i)$. Assume that $f'(i) \leq f^{k+1}(i)$ for $i < l$ and $f'(i) = f^k(i)$ for $i \geq l$. Then we calculate $f^{k+1}(l)$ as follows:

$$f^{k+1}(l) = min_{a\epsilon A(i)}[C_i(a) + \sum_{j\epsilon S} p_{ij}(a) f'(j)] \tag{5.3}$$

$$\leq min_{a\epsilon A(i)}[C_i(a) + \sum_{j\epsilon S} p_{ij}(a) f^*(j)] = f^*(l). \tag{5.4}$$

In the forward search, j is a successor of state l, that is, $j < l$ and therefore $f^{k+1}(j)$ is defined before defining $f^{k+1}(l)$. Therefore, each update within a stage maintains the admissibility property. During backward search, the value of a state is updated based on the value of those states from

which it can be reached. Thus, by the same argument as in the forward search, the backward search is also admissible.

(2) The MDP has a finite state space and finitely many actions, and therefore there are only finitely many policies possible. Since each stage (up to the last one) finds a better policy, this process converges in finite time. Value Iteration is performed on the states in the solution graph. By the convergence proof of Value Iteration for stochastic shortest-path problems [3], after sufficient iterations, the error bound of the solution will be bounded by ϵ.

5.2 Convergence test

Once G' is constructed, Value Iteration is applied to the states in G' until the resulting policy is ϵ-optimal, for the given ϵ. As an iteration of Value Iteration might change the best action for any state, the solution graph after that iteration might contain some nonterminal leaf state. If so, then the control is passed to the main algorithm from the convergence test so that the solution graph is expanded and re-evaluated.

We refer to the Bellman residual of the start state as the *error bound*. The error bound is calculated when the solution graph does not contain any nonterminal leaf states in the solution. The convergence test is repeated until the error bound falls below ϵ. Once the error bound is $\leq \epsilon$, the solution will be ϵ-optimal.

Chapter 6

Examples

The performance of the $BLAO^*$ algorithm was examined on a set of race-track problems and randomly generated Markov Decision Processes. Race-track problems were used by Barto et al. [1] and Hansen et al. [8] to illustrate the behavior of the $RTDP$ and LAO^* algorithms, respectively. Because the race-track MDPs are highly structured and nearly deterministic, random MDPs were used as a sanity check benchmark for the three algorithms.

6.0.1 Race-track problem

A detailed description of the family of race-track problems is given by Barto, et al. [1]. These MDPs model idealized automobile racing. Race-track problems are discrete-time optimal control problems [1]. Here we consider a single car in the race track and the motion of the car is probabilistic. The track is divided into squares and each square represents a possible discrete location of the car. The length and shape of the race track can vary from instance to instance. The race track has a starting line and a finish line. A car can start at any point along the starting line and moves along the track to reach the finish line. The car can reach any point on the finish line. Each iteration of an algorithm starts when the car is at the starting line and ends when the car reaches the finish line.

The car accelerates or decelerates on the track to reach the finish line and to avoid collisions with the boundaries. The accelerations and decelerations are simulated such that the car can maintain its speed, or slow down or speed up in either direction by one step per move. Thus, the car can take any one of nine actions by changing its velocity by one in any one of eight directions or by keeping the velocity constant. An action can be successful and result in the intended state

with some probability p or can fail and result in some other state with a probability $1-p$, to make the transitions stochastic.

Whenever a car collides with the boundary, it is moved back to a random place in the starting line and the velocity of the car is set to zero in each direction. This ensures that the car can leave the track only by crossing the finish line. A new iteration of an algorithm starts whenever the car hits the boundary or reaches or crosses the finish line. Whenever a car crosses the finish line, it is assumed to stay in that state until the next iteration starts.

The set of start states consists of those states on the starting line that have velocity zero in both x and y directions. Those states other than the walls that are not in the track, but are reachable from the track in one time step from inside the race track constitute the set of goal states. The set of goal states is absorbing. That is, actions taken in a goal state do not cause a change of state. The cost of taking any action on a non-goal state is 1 and the cost of any action on any goal state is 0. The problem of interest is to find a policy that minimizes the expected infinite horizon cost. That is, to find a policy that reaches the finish line with the fewest expected moves, starting in any of the start states with equal probability.

6.0.2 Random MDPs

Race-track examples are closer to being deterministic than many MDPs. In a race-track problem, for many states and actions, the action succeeds with probability 0.9 and fails with probability 0.1, or has a transition with probability one. Thus the number of successor states when an action is taken on a state is at most two. On the other hand, the random MDPs we generated had up to ten successor states for an action (we chose ten arbitrarily). The set of states and the set of actions are finite. The number of states and actions in an MDP is given as input to the random MDP generator.

The start state is *state 0* and the goal state are chosen randomly. The cost of taking any action on a non-goal state is 1 and the cost of an action on a goal state is 0. Any action on a goal state will result in the goal state only and hence it is an absorbing state.

For each action and for each state, the number of successor states and the successor states are chosen randomly. The number of successor states n is picked randomly ($0 < n \leq 10$) and then n distinct states are picked randomly from the state space. The probability of reaching a successor state (among n states) is picked randomly from 0.0 to 1.0 (probability is 1.0 only if $n = 1$). [1] A state

[1] The algorithm that generates the random MDPs takes the time of execution as the seed and generates random values using the srand() and rand() functions in C.

is not repeated in the set of successor states and the sum of probabilities of reaching each successor state for a given state and a given action is equal to one. For each action, the set of successor states is sorted in descending order of their probabilities. This ensures that the best action associated with each state initially, will result in a successor state with maximum probability.

Each algorithm ($BLAO^*, LAO^*, RTDP$) is run on the generated MDPs for a series of iterations. Each iteration starts at the start state and ends at the goal state. The iterations continue until a policy that minimizes the expected infinite horizon cost is found or for a finite number of trials (for $RTDP$).

Chapter 7

Analysis

Hansen and Zilberstein showed that LAO^* converges to an optimal solution faster than *Value Iteration* or *Policy Iteration* [8]. In our experiments, $BLAO^*$ converged faster than LAO^* in almost every instance. The algorithms LAO^*, $BLAO^*$ and $RTDP$ have been implemented in C. The results shown in this section were obtained on a Pentium 4 processor with 256 KB cache.

7.0.3 Race-track Examples

The race-track problems are used by Barto et al. [1] and, Hansen and Zilberstein in [8] to illustrate the performances of $RTDP$ and LAO^* respectively. Hence we used similar race-track examples to evaluate the performance of $BLAO^*$ against $RTDP$ and LAO^*. $BLAO^*$, LAO^* and $RTDP$ were run on five different race tracks with varying length and shape. The race tracks with 21,371 and 9,271 states were used in Hansen's original comparison of LAO^* and $RTDP$. The others were generated for this paper. In particular, $RTDP$ converged significantly more slowly on the track with 32,380 states than $BLAO^*$ did. Table 7.1 summarizes the time taken to find an optimal solution for race tracks by $RTDP$, LAO^* and $BLAO^*$ to converge. The race-tracks differ in size and shape. The convergence time is calculated in the order of seconds. The value 0.00 for the convergence time shows that the algorithms took very less time, few micro seconds, to converge. Table 7.2 summarizes the average number of states expanded by each of LAO^* and $BLAO^*$ on each race track. The values in the tables show that $BLAO^*$ converges faster, and expands fewer states than LAO^*. In each case, we set $\epsilon = 0$.

Figure 7.1 compares the performance of $BLAO^*$ to those of LAO^* and $RTDP$ in solving race-track problems. Note that $BLAO^*$ converges to an optimal solution much more quickly than LAO^*

Table 7.1: Comparison of convergence time for $RTDP$, LAO^* and $BLAO^*$ algorithms on race track examples.

Average Convergence Time			
No. of states	RTDP	LAO*	BLAO*
46	0.19	0.00	0.00
1849	0.51	0.01	0.00
9271	0.87	0.37	0.16
21371	2.40	1.74	0.98
32380	6.34	7.44	2.64

Table 7.2: Comparison of states expanded for LAO^* and $BLAO^*$ algorithms on race-track examples.

Number of states expanded		
No. of states	LAO*	BLAO*
46	13	21
1849	323	330
9271	4390	4151
21371	14297	14108
32380	22756	22276

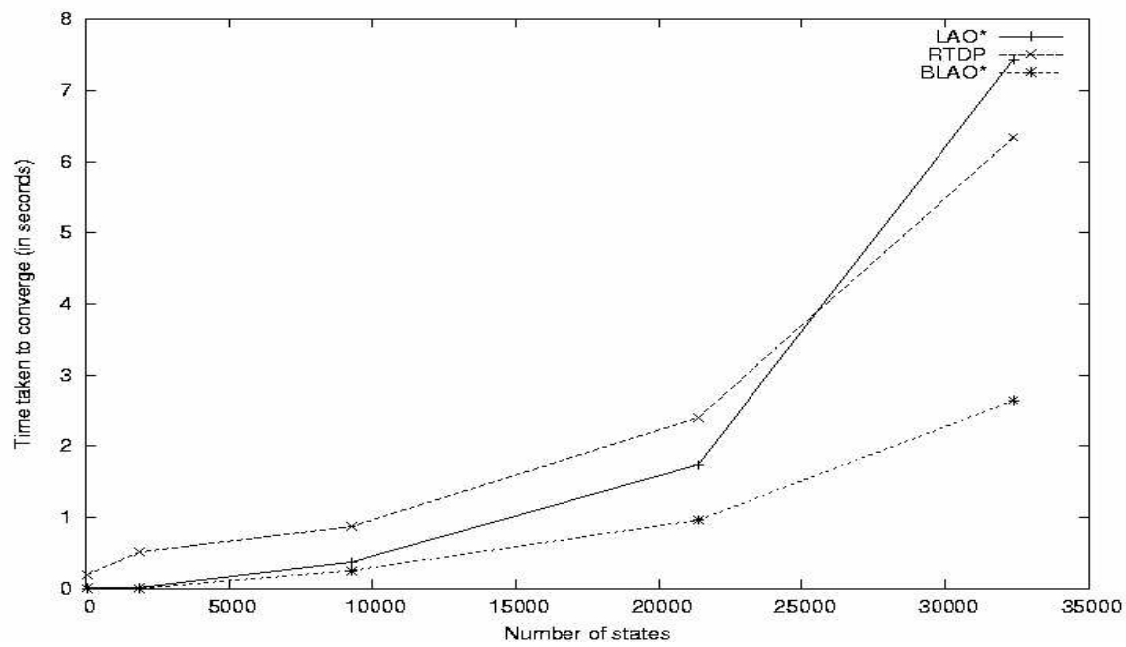

Figure 7.1: Comparison of time taken to converge by $BLAO^*$, LAO^* and $RTDP$ algorithms.

and $RTDP$.

Figure 7.2 compares the total number of states expanded by $BLAO^*$ and LAO^* to converge to an optimal solution for race-track examples. The number of states expanded by $BLAO^*$ is consistently less than the number expanded by LAO^*.

7.0.4 Random Markov Decision Processes

We generated these examples for evaluating the performance of $BLAO^*$ on MDPs with high out-degree. The description of the generation of these examples is given in Section 6. Table 7.3 summarizes the time taken for $RTDP$, LAO^* and $BLAO^*$ to converge for random MDPs. $RTDP$ with a maximum of 5,000,000 trials did not converge on large random MDPs (MDPs with 5,000, 10,000 and 25,000 states) and ran 320, 316 and 274 seconds respectively. The results show that $BLAO^*$ is faster than LAO^* and significantly faster than $RTDP$ on random MDPs.

Figure 7.3 compares the performance of $BLAO^*$ and LAO^* in solving random MDPs. Note that $BLAO^*$ converges to an optimal solution much faster than LAO^* and $RTDP$.

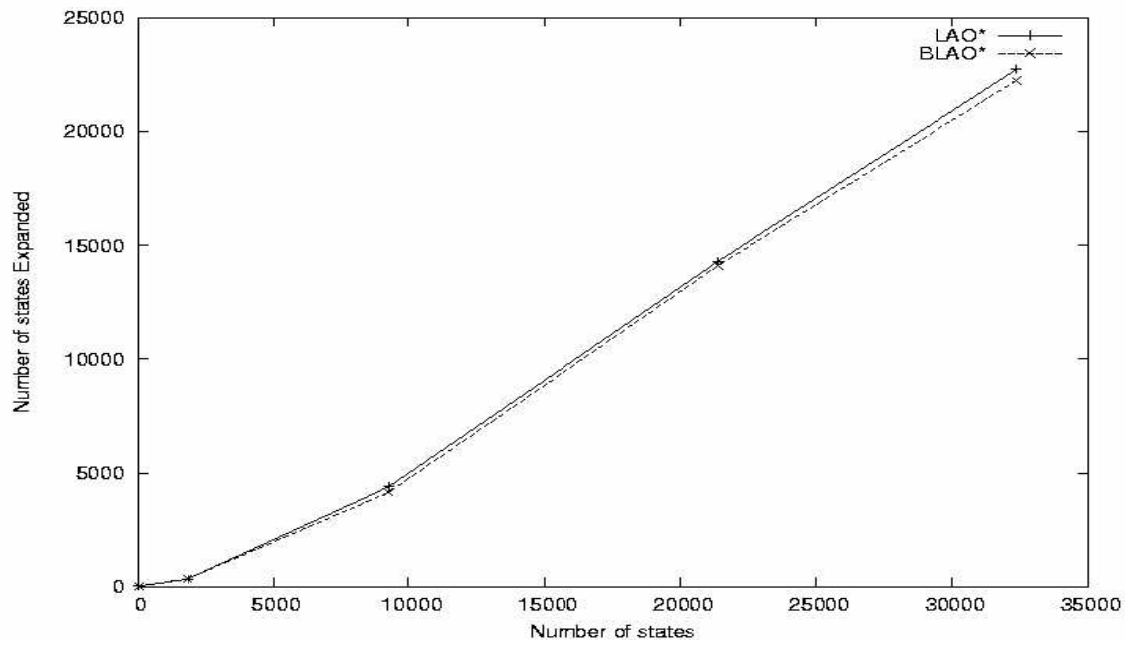

Figure 7.2: Number of states expanded by $BLAO^*$ and LAO^* for race-track examples.

Table 7.3: Comparison of convergence time for $RTDP$, LAO^* and $BLAO^*$ algorithms on randomly generated MDPs.

Average Convergence Time			
No. of states	RTDP	LAO*	BLAO*
1000	0.19	0.40	0.35
5000	–	3.26	3.05
10000	–	11.65	10.01
25000	–	24.36	22.81

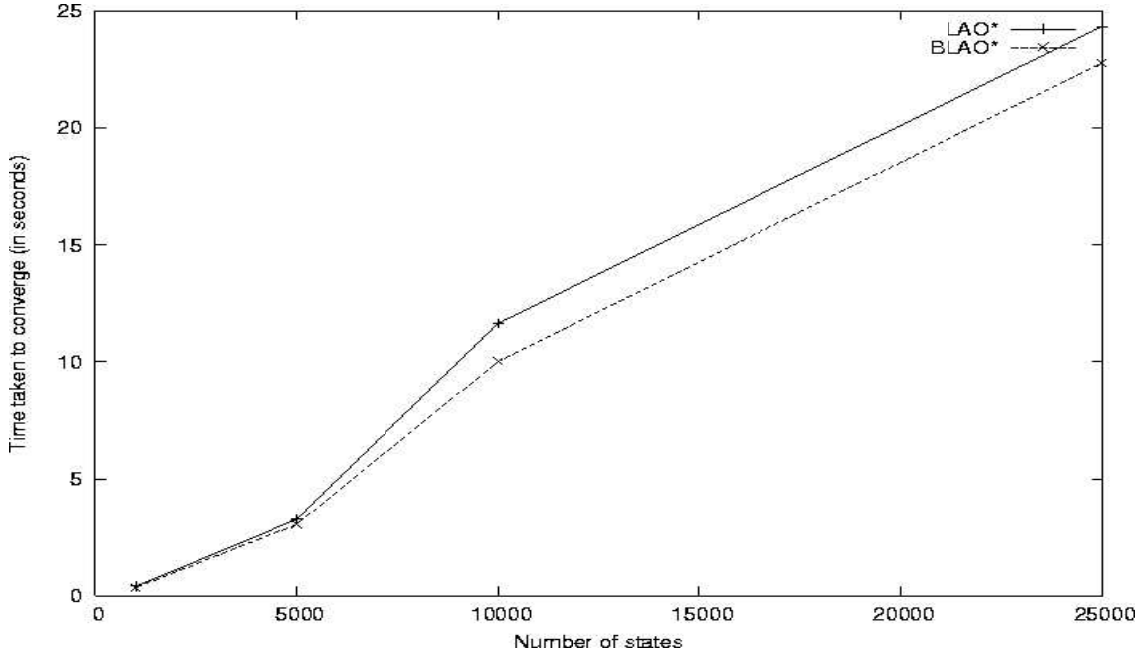

Figure 7.3: Comparison of time taken to converge by $BLAO^*$ and LAO^* on Random MDPs.

The speedup in performance of $BLAO^*$ over LAO^* for random MDPs is not significant, which we believe is a consequence of high out-degree and high randomness in the state transitions of random MDPs. In our random MDPs, a state can transition to more than one (up to 10) states. Hence, the probability of a transition to each successor state is small and does not guide the searches as well as higher-probability transitions do. Although the searches take the next most probable state of all the successor states, the probability of reaching a goal state from that state is small. Thus, both the searches have to explore many paths and states to choose an optimal path. It is important to note that $BLAO^*$ is faster than LAO^* in all our tests, even in the most random examples.

When the algorithm $BLAO^*$ is run, it forks off two processes, one for forward search and one for backward search. Because these processes run independently and their timing is affected by external factors, the searches may meet in different places each time the algorithm is run on a given example. Thus, the total number of states expanded by $BLAO^*$ may vary slightly each time it is run against the same example. This is so because the forward search and the backward search can meet at any state in the graph and the number of states expanded depends on how fast the two searches meet. In our experiments $BLAO^*$ was run an average of 10 times on each example. The

Table 7.4: Comparison of number of states expanded by LAO^* and $BLAO^*$ algorithms on random MDPs.

Number of states expanded		
No. of states	BLAO*	LAO*
1000	1565	1000
5000	6438	5000
10000	13783	10000
25000	31542	25000

values in the tables show the average over all runs for each example, and the average number of nodes expanded.

For the random MDPs, the number of states expanded by $BLAO^*$ in finding the optimal solution is apparently greater than the total number of states in an MDP. This is so because a state can be expanded by either of the searches (forward search or backward search) or by both. Whenever a state is expanded, it is counted once towards the total number of nodes expanded. So, whenever a state is expanded in both directions, it is counted twice towards the total number of states expanded by $BLAO^*$ algorithm. Due to the nature of random MDPs, many states are expanded in both directions and hence are counted twice. Thus, the total number of states expanded by $BLAO^*$ may be higher than the number of states in that MDP.

On each run of $BLAO^*$, the time taken to find an optimal solution is less than the time taken by any run of LAO^*. No matter where the searches meet, the solution given by $BLAO^*$ is identical to the one given by LAO^*. Hence, the solutions found by $BLAO^*$ are almost certainly all optimal.

Chapter 8

Discussion

We have introduced a new algorithm, Bidirectional LAO^*, for planning with MDPs. On the benchmark MDPs for LAO^*, race-track problems, our algorithm out-performs both LAO^* and *RTDP* and shows insignificant speedup on randomly generated MDPs. The algorithms are compared on MDPs which vary in the number of states, start and goal states.

MDPs can be represented compactly using ADDs. Symbolic LAO^* by Eric Hansen et al. [7], shows significant performance speedup on MDPs represented using MDPs. We are currently working on the implementation of $BLAO^*$ to MDPs represented using ADDs and are hoping to find similar speedups to our algorithm.

The convergence of *RTDP*, another method for solving MDPs, is very slow. A labeling procedure was introduced by Blai Bonet and Hector Geffner, which significantly improved the convergence of *RTDP* [5]. We have not yet directly compared $BLAO^*$ with *Labeled RTDP*. However, we are planning to work on the implementation of the labeling procedure to $BLAO^*$, hoping to find similar speedups to our algorithm.

Bibliography

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

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

[3] Dimitri Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, MA, 1995.

[4] Venkata Deepti Kiran Bhuma and Judy Goldsmith. Bidirectional LAO* algorithm. In *Proc. IICAI'03*, pages 980–992, 2003.

[5] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programing. In *Proc. ICAPS 2003*, pages 12–21, 2003.

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

[7] Eric A. Hansen, Zhengzhu Feng, and Shlomo Zilberstein. Symbolic generalization for on-line planning. In *Proc. Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI-03)*, 2003.

[8] Eric A. Hansen and Shlomo Zilberstein. LAO* : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

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

[10] Alberto Martelli and Ugo Montanari. Additive AND/OR graphs. In *Proc. IJCAI-73*, pages 1–11, 1973.

[11] Nils J. Nilsson. *Principles of Articifical Intelligence*. Tioga Publishing, Palo Alto, CA, 1980.

Vita

Date and Place of Birth: Cuddapah, India, June 7, 1981

Education: Bachelor of Technology in Computer Science & Engineering
KSRM College of Engineering, Cuddapah, 2002

Professional Positions: Graduate Research Assistant
Department of Computer Science
University of Kentucky
Lexington, Kentucky, 2003-2004

Honors: Kentcuky Graduate Scholarship
University of Kentucky, 2002-2003

Professional Publications:

"*BLAO** Algorithm" In *Proceedings of IICAI-03*, pages 980–992, 2003.

Venkata Deepti Kiran Bhuma