ELSEVIER

Artificial Intelligence 129 (2001) 35–62

Artificial Intelligence

www.elsevier.com/locate/artint

LAO*: A heuristic search algorithm that finds solutions with loops

Eric A. Hansen [a,*], Shlomo Zilberstein [b]

[a] *Computer Science Department, Mississippi State University, Mississippi State, MS 39762, USA*
[b] *Computer Science Department, University of Massachusetts, Amherst, MA 01002, USA*

Received 15 February 2000; received in revised form 8 March 2001

Abstract

Classic heuristic search algorithms can find solutions that take the form of a simple path (A*), a tree, or an acyclic graph (AO*). In this paper, we describe a novel generalization of heuristic search, called LAO*, that can find solutions with loops. We show that LAO* can be used to solve Markov decision problems and that it shares the advantage heuristic search has over dynamic programming for other classes of problems. Given a start state, it can find an optimal solution without evaluating the entire state space.

Keywords: Heuristic search; Dynamic programming; Markov decision problems

1. Introduction

One of the most widely-used frameworks for problem-solving in artificial intelligence is state-space search. A state-space search problem is defined by a set of states, a set of actions (or operators) that map states to successor states, a start state, and a set of goal states. The objective is to find a sequence of actions that transforms the start state into a goal state, and also optimizes some measure of the quality of the solution.

Two well-known heuristic search algorithms for state-space search problems are A* and AO* [18]. A* finds a solution that takes the form of a sequence of actions leading in a path from the start state to a goal state. AO* finds a solution that has a conditional structure and takes the form of a tree, or more generally, an acyclic graph. But no heuristic search

* Corresponding author.
E-mail address: hansen@cs.msstate.edu (E.A. Hansen).

0004-3702/01/$ – see front matter
PII: S0004-3702(01)00106-0

algorithm has been developed that can find a solution that takes the form of a cyclic graph, that is, a solution with loops.

For many problems that can be formalized in the state-space search model, it does not make sense for a solution to contain loops. A loop in a solution to a theorem-proving problem represents circular reasoning. A loop in a solution to a problem-reduction task represents a failed reduction to primitive subproblems. However, there is an important class of problems for which it does make sense for a solution to contain loops. These problems can be formalized as Markov decision processes, a framework that is widely used in artificial intelligence for problems of planning and learning under uncertainty [1, 5]. A *Markov decision process* (MDP) models problems of sequential decision making that include actions that transform a state into one of several possible successor states, with each possible state transition occurring with some probability. A solution to an MDP takes the form of a mapping from states to actions called a *policy*. A policy is executed by observing the current state and taking the action prescribed for it. A solution represented in this way implicitly contains both branches and loops. Branching is present because the state that stochastically results from an action determines the next action. Looping is present because the same state may be revisited under a policy. (As an example of a plan with a conditional loop, consider an action that has its desired effect with probability less than one and otherwise has no effect. An appropriate plan might be to repeat the action until it "succeeds".)

An optimal policy can be found using a dynamic programming algorithm such as policy iteration or value iteration [2]. But a disadvantage of dynamic programming is that it evaluates the entire state space. In effect, it finds a policy for every possible starting state. By contrast, heuristic search algorithms solve a problem for a particular starting state and use an admissible heuristic to focus the search, and remove from consideration regions of the state space that can't be reached from the start state by an optimal solution. For problems with large state spaces, heuristic search has an advantage over dynamic programming because it can find an optimal solution for a start state without evaluating the entire state space.

This advantage is well-known for problems that can be solved by A* or AO*. In fact, an important theorem about the behavior of A* is that (under certain conditions) it evaluates the minimal number of states among all algorithms that find an optimal solution using the same heuristic [7]. A related result has also been established for AO* [3]. In this paper, we generalize the heuristic search approach to find solutions with loops and show that the resulting algorithm, which we call LAO*, can solve planning problems that are formalized as MDPs.

The paper is organized as follows. Section 2 reviews dynamic programming algorithms for MDPs and heuristic search algorithms for state-space search problems, and discusses their relationship. Section 3 introduces the LAO* algorithm and Section 4 describes some extensions of this algorithm that illustrate the relevance of heuristic search techniques to the problem of solving MDPs more efficiently. Section 5 describes the performance of LAO* on two test problems, and discusses some search control issues that affect its efficiency.

2. Background

We begin with a review of MDPs and dynamic programming algorithms for solving them. We then review the heuristic search algorithm AO*, and discuss the relationship between dynamic programming and heuristic search.

2.1. Markov decision processes

We consider an MDP with a finite set of states, S. For each state $i \in S$, let $A(i)$ denote a finite set of actions available in that state. Let $p_{ij}(a)$ denote the probability that taking action a in state i results in a transition to state j. Let $c_i(a)$ denote the expected immediate cost of taking action a in state i.

We focus on a special class of MDPs called *stochastic shortest-path problems* [2] because it generalizes traditional shortest-path problems for which artificial intelligence (AI) search algorithms have been developed. (The name "shortest-path" reflects an interpretation of action costs as arc lengths.) A stochastic shortest-path problem is an MDP with a set of terminal states, $T \subseteq S$. For every terminal state $i \in T$, no action can cause a transition out of this state (i.e., it is an absorbing state) and the immediate cost of any action taken in this state is zero. Formally, $p_{ii}(a) = 1$ and $c_i(a) = 0$ for all $i \in T$ and $a \in A(i)$. For all other states, immediate costs are assumed to be positive for every action, that is, $c_i(a) > 0$ for all states $i \notin T$ and actions $a \in A(i)$. The objective is to reach a terminal state while incurring minimum expected cost. Thus, we can think of terminal states as goal states and we will use the terms "terminal state" and "goal state" interchangeably from now on. Because the probabilistic outcomes of actions can create a nonzero probability of revisiting the same state, the worst-case number of steps needed to reach a goal state cannot be bounded. Hence, stochastic shortest-path problems are said to have an *indefinite horizon*.

A solution to an indefinite-horizon MDP can be represented as a stationary mapping from states to actions, $\pi : S \to A$, called a *policy*. A policy is said to be *proper* if, for every state, it ensures that a goal state is reached with probability 1.0. For a proper policy, the expected cost for reaching a goal state from each state $i \in S$ is finite and can be computed by solving the following system of $|S|$ linear equations in $|S|$ unknowns:

$$f^{\pi}(i) = \begin{cases} 0 & \text{if } i \text{ is a goal state,} \\ c_i(\pi(i)) + \sum_{j \in S} p_{ij}(\pi(i)) f^{\pi}(j) & \text{otherwise.} \end{cases} \tag{1}$$

We call f^{π} the *evaluation function* of policy π. We assume that the stochastic shortest-path problems we consider have at least one proper policy, and that every improper policy has infinite expected cost for at least one state. This assumption is made by Bertsekas [2] in developing the theory of stochastic shortest-path problems. It generalizes the assumption for deterministic shortest-path problems that there is a path to the goal state from every other state, and all cycles have positive cost.

A policy π is said to dominate a policy π' if $f^{\pi}(i) \leqslant f^{\pi'}(i)$ for every state i. An optimal policy, π^*, dominates every other policy, and its evaluation function, f^*, satisfies the

following system of $|S|$ nonlinear equations in $|S|$ unknowns, called the Bellman optimality equation:

$$f^*(i) = \begin{cases} 0 & \text{if } i \text{ is a goal state,} \\ \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f^*(j) \right] & \text{otherwise.} \end{cases} \tag{2}$$

Dynamic programming algorithms for MDPs find the evaluation function that satisfies the Bellman equation by successively improving an estimated evaluation function, f, by performing *backups*. For each state i, a backup takes the following form:

$$f(i) := \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right]. \tag{3}$$

Performing a backup for every state i in the state set is referred to as a *dynamic-programming update*. It is the core step of dynamic-programming algorithms for solving MDPs. The fact that a backup is performed for every state in the state set is characteristic of dynamic programming, which evaluates all problem states.

There are two related dynamic programming algorithms for indefinite-horizon MDPs: policy iteration and value iteration. Policy iteration is summarized in Table 1. It interleaves the dynamic-programming update, used for policy improvement, with policy evaluation. When the current policy π is not optimal, the dynamic-programming update finds a new policy π' such that $f^{\pi'}(i) \leqslant f^{\pi}(i)$ for every state i and $f^{\pi'}(i) < f^{\pi}(i)$ for at least one state i. Because the number of possible policies is finite and policy iteration improves the current policy each iteration, it converges to an optimal policy after a finite number of iterations.

Value iteration is summarized in Table 2. Each iteration, it improves an evaluation function f by performing a dynamic programming update. When the evaluation function

Table 1
Policy iteration

1. Start with an initial policy π.
2. *Evaluate policy*: Compute the evaluation function f^{π} for policy π by solving the following set of $|S|$ equations in $|S|$ unknowns,
$$f^{\pi}(i) = c_i(\pi(i)) + \sum_{j \in S} p_{ij}(\pi(i)) f^{\pi}(j).$$
3. *Improve policy*: For each state $i \in S$, let
$$\pi'(i) := \arg \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f^{\pi}(j) \right].$$
Resolve ties arbitrarily, but give preference to the currently selected action.
4. *Convergence test*: If π' is the same as π, go to step 5. Otherwise, set $\pi = \pi'$ and go to step 2.
5. Return an optimal policy.

Table 2
Value iteration

1. Start with an initial evaluation function f and parameter ε for detecting convergence to an ε-optimal evaluation function.
2. *Improve evaluation function*: For each state $i \in S$,
$$f'(i) := \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right].$$
3. *Convergence test*: If the error bound of the evaluation function is less than or equal to ε, go to step 4. Otherwise, set $f = f'$ and go to step 2.
4. Extract an ε-optimal policy from the evaluation function as follows. For each state $i \in S$,
$$\pi(i) := \arg \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right].$$

is sufficiently close to optimal, as measured by an error bound, a policy can be extracted from the evaluation function as follows:

$$\pi(i) := \arg \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right]. \tag{4}$$

For stochastic shortest-path problems, an error bound can be computed based on the *Bellman residual* of the dynamic programming update, which is

$$r = \max_{i \in S} \left| f'(i) - f(i) \right|, \tag{5}$$

and the mean first passage times for every state i under the current policy π. The *mean first passage time* for state i under policy π, denoted $\phi^{\pi}(i)$, is the expected number of time steps it takes to reach a goal state, beginning from state i and following policy π. Mean first passage times can be computed by solving the following system of linear equations for all $i \in S$:

$$\phi^{\pi}(i) = 1 + \sum_{j \in S} p_{ij}\big(\pi(i)\big) \phi^{\pi}(j). \tag{6}$$

Given the Bellman residual and mean first passage times, a lower bound, f^L, on the optimal evaluation function, f^*, is given by:

$$f^L(i) = f^{\pi}(i) - \phi^{\pi}(i)\ r, \tag{7}$$

where f^{π} is the evaluation function of policy π. For value iteration, an upper bound, f^U, on the optimal evaluation function is given by:

$$f^U(i) = f^{\pi}(i) + \phi^{\pi}(i)\ r. \tag{8}$$

For policy iteration, the current evaluation function is an upper bound on the optimal evaluation function. The maximum difference between the upper and lower bounds,

Table 3
Trial-based RTDP

1. Start with an admissible evaluation function f.
2. Repeat the following trial n times.

   Trial: Set the current state i to the start state and repeat the following steps m times or until a goal state is reached.

   (a) Improve the evaluation function by performing the following backup for the current state i,

   $$f(i) := \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right].$$

   (b) Take the action determined to be best for state i by the backup and change the current state i to the state that results from a stochastic transition caused by the action.
3. Extract a partial policy from evaluation function f as follows. For each state $i \in S$ that can be reached from the start state by selecting actions greedily,

   $$\pi(i) := \arg \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right].$$

defined as $\max_{i \in S} |f^U(i) - f^L(i)|$, gives a bound on the error of the current solution. A solution is ε-optimal if the error bound is less than ε. For any real number $\varepsilon > 0$, both policy iteration and value iteration converge to an ε-optimal solution after a finite number of iterations.

Real-time dynamic programming We are particularly interested in stochastic shortest-path problems that include a start state, $s \in S$, and for which the objective is to find a solution that minimizes the expected cost of reaching a goal state from the start state. Bertsekas [2] does not discuss this special case of a stochastic shortest-path problem. However, Barto et al. [1] do and note that it generalizes the classic AI search problem of finding a minimum-cost path from a start state to a goal state. Because finding a minimum-cost solution from a start state to a goal state is a special case of the problem of finding a minimum-cost solution from every state to a goal state, it can be solved using either policy iteration or value iteration. However, neither of these algorithms uses knowledge of the start state to focus computation on just those states that are reachable from the start state by following an optimal policy. Instead, both repeat an improvement step (the dynamic programming update) that updates the evaluation function for all states. In effect, both algorithms solve an MDP for all possible starting states.

Barto et al. [1] describe an algorithm called *real-time dynamic programming* (RTDP) that avoids exhaustive evaluation of the state set. Table 3 summarizes *trial-based RTDP*, which solves an MDP by organizing computation as a sequence of trials. Each trial consists of a sequence of steps. In each step, an action is selected based on lookahead search of

depth one or more, and the current state is updated based on the outcome of the action. In addition, the evaluation function is updated for a subset of states that includes the current state. At the beginning of each trial, the current state is set to the start state. A trial ends when the goal state is reached, or after a specified number of steps. An important feature of trial-based RTDP is that the evaluation function is only updated for states that are reached from the start state when actions are selected greedily based on the current evaluation function. Thus, RTDP may ignore large regions of the state space. Barto et al. [1] prove that under certain reasonable conditions, RTDP converges (asymptotically) to an optimal solution without necessarily evaluating the entire state space. They relate this result to heuristic search theory by showing that it generalizes the convergence theorem of Korf's learning real-time heuristic search algorithm (LRTA*) [11].

Dean et al. [5] describe a related algorithm that performs policy iteration on a subset of the states of an MDP, using various methods to identify the most relevant states and gradually increasing the subset until eventual convergence (or until the algorithm is stopped). The subset of states is called an *envelope* and a policy defined on this subset of states is called a *partial policy*. Adding states to an envelope is similar to expanding a partial solution in a search graph and the idea of using a heuristic to evaluate the fringe states of an envelope has been explored [6,21]. However, this algorithm is presented as a modification of policy iteration (and value iteration), rather than a generalization of heuristic search. In particular, the assumption is explicitly made that convergence to an optimal policy requires evaluating the entire state space.

Both RTDP and the related envelope algorithm are motivated by the problem of search (or planning) in real time. Both allow search to be interleaved with execution. The time constraint on search is usually the time before the next action needs to be executed, and both Dean et al. [5] and Tash and Russell [21] describe decision-theoretic approaches to optimizing the value of search in the interval between actions. The possibility of using real-time search to solve MDPs raises the following question: is there an off-line heuristic search algorithm that can solve MDPs? Before addressing this question directly, we review the framework of state-space heuristic search.

2.2. Heuristic search in AND/OR graphs

Like an MDP, a state-space search problem is defined by a set of states (including a start state and a set of goal states), a set of actions that cause state transitions, and a cost function that specifies costs for state transitions. The objective is to find a minimal-cost path from the start state to a goal state. It is customary to formalize a state-space search problem as a graph in which each node of the graph represents a problem state and each arc represents a state transition that results from taking an action. We use the same notation for state-space search problems as for MDPs. Let S denote the set of possible states, let $s \in S$ denote a start state that corresponds to the root of the graph, and let $T \subset S$ denote a set of goal (or terminal) states that occur at the leaves of the graph. Let A denote a finite set of actions and let $A(i)$ denote the set of actions applicable in state i.

The most general state-space search problem considered in the AI literature is AND/OR graph search. Following Martelli and Montanari [14] and Nilsson [18], we define an AND/OR graph as a hypergraph. Instead of arcs that connect pairs of states as in an

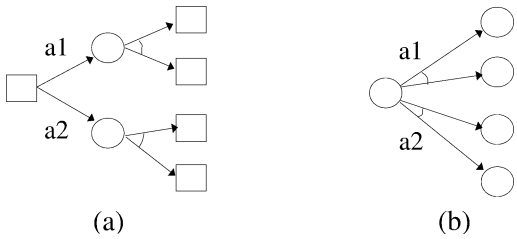

Fig. 1. (a) shows an OR node with two arcs leaving it, one for action $a1$ and one for action $a2$. Each arc leads to an AND node with two successor OR nodes, one for each possible successor state. (By convention, a square denotes an OR node and a circle denotes an AND node. In the terminology of decision analysis, a square corresponds to a choice node and a circle corresponds to a chance node.) (b) shows a state, indicated by a circle, with two 2-connectors leaving it, one for action $a1$ and one for action $a2$. Each 2-connector leads to two possible successor states. The representation on the right, using state nodes and k-connectors, is equivalent to the representation on the left, which uses OR and AND nodes.

ordinary graph, a hypergraph has *hyperarcs* or *k-connectors* that connect a state to a set of k successor states. Fig. 1 relates the concept of OR and AND nodes to the concept of a hyperarc or k-connector.

A k-connector can be interpreted in different ways. In problem-reduction search, it is interpreted as the transformation of a problem into k subproblems. We consider problems of planning under uncertainty in which it is interpreted as an action with an uncertain outcome. The action transforms a state into one of k possible successor states, with a probability attached to each successor. We let $p_{ij}(a)$ denote the probability that taking action a in state i results in a transition to state j. This interpretation of AND/OR graphs is made by Martelli and Montanari [14] and Pattipati and Alexandridis [19], among others.

In AND/OR graph search, a solution takes the form of an acyclic subgraph called a *solution graph*, which is defined as follows:

- the start state belongs to a solution graph;
- for every nongoal state in a solution graph, exactly one outgoing k-connector (corresponding to an action) is part of the solution graph and each of its successor states also belongs to the solution graph;
- every directed path in the solution graph terminates at a goal state.

A cost function assigns a cost to each k-connector. Let $c_i(a)$ denote the cost for the k-connector that corresponds to taking action a in state i. We assume each goal state has a cost of zero. A minimal-cost solution graph is found by solving the following system of equations,

$$f^*(i) = \begin{cases} 0 & \text{if } i \text{ is a goal state,} \\ \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f^*(j) \right] & \text{otherwise,} \end{cases} \tag{9}$$

where $f^*(i)$ denotes the cost of an optimal solution for state i and f^* is called the optimal *evaluation function*. Note that this is identical to the optimality equation defined for MDPs in Eq. (2), although we now assume the solution can be represented as an acyclic graph. This assumption means that a goal state can be reached from the start state after a bounded number of actions (where the bound is equal to the longest path from the start state to a goal state). By contrast, a stochastic shortest-path problem has an unbounded or indefinite

horizon. Thus the traditional framework of AND/OR graph search does not directly apply to stochastic shortest-path problems.

*AO** For state-space search problems that are formalized as AND/OR graphs, an optimal solution graph can be found using the heuristic search algorithm AO*. Nilsson [16,17] first described a version of AO* for searching AND/OR trees to find a solution in the form of a tree. Martelli and Montanari [13,14] generalized this algorithm for searching AND/OR graphs to find a solution in the form of an acyclic graph. Nilsson [18] also used the name AO* for this more general algorithm. Because any acyclic graph can be unfolded into an equivalent tree, the tree-search and graph-search versions of AO* solve the same class of problems. The graph-search version is more efficient when the same state can be reached along different paths because it avoids performing duplicate searches.

Like other heuristic search algorithms, AO* can find an optimal solution without considering every problem state. Therefore, a graph is not usually supplied explicitly to the search algorithm. An implicit graph, G, is specified implicitly by a start state s and a successor function. The search algorithm constructs an *explicit graph*, G', that initially consists only of the start state. A tip or leaf state of the explicit graph is said to be terminal if it is a goal state; otherwise, it is said to be nonterminal. A nonterminal tip state can be expanded by adding to the explicit graph its outgoing k-connectors (one for each action) and any successor states not already in the explicit graph.

AO* solves a state-space search problem by gradually building a solution graph, beginning from the start state. A *partial solution graph* is defined similarly to a solution graph, with the difference that the tip states of a partial solution graph may be nonterminal states of the implicit AND/OR graph. A partial solution graph is defined as follows:

- the start state belongs to a partial solution graph;
- for every nontip state in a partial solution graph, exactly one outgoing k-connector (corresponding to an action) is part of the partial solution graph and each of its successor states also belongs to the partial solution graph;
- every directed path in a partial solution graph terminates at a tip state of the explicit graph.

As with solution graphs, there are many possible partial solution graphs and an evaluation function can be used to rank them. The cost of a partial solution graph is defined similarly to the cost of a solution graph. The difference is that if a tip state of a partial solution graph is nonterminal, it does not have a cost that can be propagated backwards. Instead, we assume there is an admissible heuristic estimate $h(i)$ of the minimal-cost solution graph for state i. A heuristic evaluation function h is said to be *admissible* if $h(i) \leqslant f^*(i)$ for every state i. We can recursively calculate an admissible heuristic estimate $f(i)$ of the optimal cost of any state i in the explicit graph as follows:

$$f(i) = \begin{cases} 0 & \text{if } i \text{ is a goal state,} \\ h(i) & \text{if } i \text{ is a nonterminal tip state,} \\ \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right] & \text{otherwise.} \end{cases} \tag{10}$$

The best partial solution graph can be determined at any time by propagating heuristic estimates from the tip states of the explicit graph to the start state. If we mark the action

Table 4
AO*

1. The explicit graph G' initially consists of the start state s.
2. While the best solution graph has some nonterminal tip state:
   (a) *Expand best partial solution*: Expand some nonterminal tip state n of the best partial solution graph and add any new successor states to G'. For each new state i added to G' by expanding n, if i is a goal state then $f(i) := 0$; else $f(i) := h(i)$.
   (b) *Update state costs and mark best actions*:
      i. Create a set Z that contains the expanded state and all of its ancestors in the explicit graph along marked action arcs. (I.e., only include ancestor states from which the expanded state can be reached by following the current best solution.)
      ii. Repeat the following steps until Z is empty.
         A. Remove from Z a state i such that no descendent of i in G' occurs in Z.
         B. Set $f(i) := \min_{a \in A(i)} [c_i(a) + \sum_j p_{ij}(a) f(j)]$ and mark the best action for i. (When determining the best action resolve ties arbitrarily, but give preference to the currently marked action.)
3. Return an optimal solution graph.

that minimizes the value of each state, the best partial solution graph can be determined by starting at the root of the graph and selecting the best action for each reachable state.

Table 4 outlines the algorithm AO* for finding a minimal-cost solution graph in an acyclic AND/OR graph. It interleaves forward expansion of the best partial solution with a cost revision step that updates estimated state costs and the best partial solution. In the simplest version of AO*, the costs of the expanded state and all of its ancestor states in the explicit graph are updated [13]. However, Martelli and Montanari [14] and Nilsson [18] note that the only ancestor states that may have their costs change are those from which the expanded state can be reached by taking marked actions (i.e., by choosing the best action for each state). Thus, the parenthetical remark in step 2(b)i of Table 4 indicates that a parent i of state j is not added to Z unless both the estimated cost of state j has changed, and state j can be reached from state i by choosing the best action for state i.

To simplify exposition, we have omitted from our outline of AO* a solve-labeling procedure that is often included in the algorithm to improve efficiency. Briefly, a state is labeled solved if it is a goal state or if all of its successor states are labeled solved. Labeling states as solved can improve the efficiency of the solution expansion step because it is unnecessary to search below a solved state for nonterminal tip states.

We also note that the best partial solution graph may have many nonterminal tip states. AO* works correctly no matter which of these tip states is chosen for expansion. However, the efficiency of AO* can be improved by using a good selection function to choose which nonterminal tip state of the best partial solution graph to expand next. Possibilities include selecting the tip state with the least estimated cost, or selecting the tip state with greatest probability of being reached.

Although AND/OR graphs are usually assumed to be acyclic, the possibility of searching cyclic AND/OR graphs has been studied [10]. In previous work, however, the solution graph is assumed to be acyclic. In Section 3, we consider how to generalize AND/OR graph search to find cyclic solution graphs.

Dynamic programming and heuristic search It is possible to find an optimal solution to an AND/OR search problem using dynamic programming, instead of using AO*. Given a complete AND/OR graph containing every problem state, backwards induction can be performed to find the optimal action for every state. Starting from goal states that have a cost of zero, the optimality equation can be solved by performing a backup for each nonterminal state in backwards order from the leaves of the AND/OR graph toward the root. Given the optimal cost for every successor of a state, the optimal cost for a state can be determined by performing a single backup for that state. Because a backup is performed once for each state and each backup has constant-time complexity, the complexity of backwards induction is linear in the number of states, even though the number of possible solution graphs is exponential in the number of states.

Dynamic programming is said to be an *implicit-enumeration algorithm* because it finds an optimal solution without evaluating all possible solutions. Once an optimal solution for a state is found, Bellman's principle of optimality allows us to infer that an optimal solution that reaches this state must include the solution that is optimal for this state. This is the reason for solving the problem in a backwards order from the leaves of the tree to the root, recursively combining solutions to increasingly larger subproblems until the original problem is solved. Using Bellman's principle of optimality to avoid enumerating all possible solutions is sometimes called *pruning by dominance*.

Pruning by dominance, upon which dynamic programming relies, is a technique for pruning the space of solutions so that not every solution has to be evaluated in order to find the best one. It is not a technique for pruning the state space. However, it is also possible to prune states, and thus avoid generating the entire AND/OR graph, by using a technique called *pruning by bounds*. This approach is taken by branch-and-bound algorithms. To avoid evaluating the entire AND/OR graph, a branch-and-bound algorithm generates the graph incrementally in a forward direction, beginning from the start state at the root of the graph. It does not generate and evaluate subgraphs that can be pruned by comparing upper and lower bounds on their optimal cost. Given a lower-bound function (or equivalently, an admissible heuristic evaluation function), it uses the following rule to avoid generating and evaluating a subgraph of an AND/OR graph: if a lower bound on the optimal cost of one k-connector (corresponding to one possible action) is greater than an upper bound on the cost of the state (determined by evaluating a subgraph that begins at another k-connector), then the first k-connector cannot be part of an optimal solution and the subgraph below it does not need to be evaluated.

Like branch-and-bound, best-first heuristic search can find an optimal solution without evaluating every state in the AND/OR graph. It is called "best-first" because it uses the lower-bound function to determine which tip state of the explicit graph to expand next, as well as to prune the search space. Instead of expanding states in a depth-first (or breadth-first) order, the idea of best-first search is to expand states in an order that maximizes the effect of pruning. The objective is to find an optimal solution while evaluating as little of the AND/OR graph as possible. A "best-first" order for expanding an AND/OR graph is to expand the state that is most likely to be part of an optimal solution. This means expanding a tip state of the best partial solution graph.

So, in order to expand the graph in best-first order, AO* must identify the best partial solution graph in an explicit graph. To do so, AO* uses backwards induction to propagate

state costs from the leaves of the explicit graph to its root. In other words, AO* uses dynamic programming in its cost revision step. In its forward expansion step, it uses the start state and an admissible heuristic to focus computation on the part of the AND/OR graph where an optimal solution is likely to be. In summary, AO* uses branch-and-bound in its forward expansion step and dynamic programming in its cost revision step. Integrating these two techniques makes it possible to find an optimal solution as efficiently as possible, without evaluating the entire state space. It also suggests a similar strategy for solving MDPs using heuristic search.

3. LAO*

We now describe LAO*, a simple generalization of AO* that can find solutions with loops. (The "L" in the name LAO* stands for "loop".) LAO* is a heuristic search algorithm that can find optimal solutions for MDPs without evaluating the entire state space. Thus, it provides a useful alternative to dynamic programming algorithms for MDPs such as value iteration and policy iteration.

It is straightforward to represent an indefinite-horizon MDP as an AND/OR graph. We let each state of the AND/OR graph correspond to a state of the MDP and we let each k-connector correspond to an action with k possible outcomes. In reviewing MDPs and AND/OR graph search, we used the same notation for their transition functions, cost functions, and optimality equations, to help clarify the close relationship between these two frameworks. A difference is that a solution to an indefinite-horizon MDP contains loops (i.e., it allows the same state to be visited more than once). Loops express indefinite-horizon behavior.

The classic AO* algorithm can only solve problems that have acyclic solutions because the backwards induction algorithm used in its cost revision step assumes an acyclic solution. The key step in generalizing AO* to create LAO* is to recognize that the cost revision step of AO* is a dynamic programming algorithm, and to generalize this step appropriately. Instead of using backwards induction, state costs can be updated by using a dynamic programming algorithm for indefinite-horizon MDPs, such as policy iteration or value iteration. This simple generalization creates the algorithm LAO* summarized in Table 5.

As in AO*, the cost revision step of LAO* is only performed on the set of states that includes the expanded state and the states in the explicit graph from which the expanded state can be reached by taking marked actions (i.e., by choosing the best action for each state). This is the set Z in step 2(b) of Table 5. The estimated cost of states that are not in this subset are unaffected by any change in the cost of the expanded state or its ancestors.

The cost revision step of LAO* can be performed using either policy iteration or value iteration. An advantage of using policy iteration is that it can compute an exact cost for each state of the explicit graph after a finite number of iterations, based on the heuristic estimates at the tip states. When value iteration is used, convergence to exact state costs is asymptotic. However, this disadvantage is usually offset by the improved efficiency of value iteration for larger problems.

Table 5
LAO*

1. The explicit graph G' initially consists of the start state s.
2. While the best solution graph has some nonterminal tip state:
   (a) *Expand best partial solution*: Expand some nonterminal tip state n of the best partial solution graph and add any new successor states to G'. For each new state i added to G' by expanding n, if i is a goal state then $f(i) := 0$; else $f(i) := h(i)$.
   (b) *Update state costs and mark best actions*:
      i. Create a set Z that contains the expanded state and all of its ancestors in the explicit graph along marked action arcs. (I.e., only include ancestor states from which the expanded state can be reached by following the current best solution.)
      ii. Perform dynamic programming (policy iteration or value iteration) on the states in set Z to update state costs and determine the best action for each state.
3. *Convergence test*: If policy iteration was used in step 2(b)ii, go to step 4. Else perform value iteration on the states in the best solution graph. Continue until one of the following two conditions is met. (i) If the error bound falls below ε, go to step 4. (ii) If the best current solution graph changes so that it has an unexpanded tip state, go to step 2.
4. Return an optimal (or ε-optimal) solution graph.

Because LAO* allows a solution graph to contain loops, forward search from the start state to identify the best partial solution graph (and the states on its fringe) terminates at any state that represents a loop back to an already expanded state of the best partial solution graph, as well as at a goal state or nonterminal tip state.

In both AO* and LAO*, the fringe of the best partial solution graph may contain many unexpanded states and the choice of which state to expand next is nondeterministic. That is, both AO* and LAO* work correctly no matter what heuristic is used to select which nonterminal tip state of the best partial solution graph to expand next. A well-chosen state selection heuristic can improve performance, however. Possibilities include expanding the state with the highest probability of being reached from the start state or expanding the state with the least cost, as already discussed for AO*.

Example Fig. 2 shows a simple example of a stochastic shortest-path problem and the steps that LAO* takes to solve it. The "gridworld" in which a path is to be found is shown in (a). The states of the problem are the cells of the grid, which are numbered from 1 to 8. The problem is to find the shortest path from the start state, labeled 1, to an absorbing goal state, labeled 4. There are three available actions in each state. One action is an attempt to move one cell north (N), one action is an attempt to move one cell east (E), and one action is an attempt to move one cell south (S). Each action is successful in moving to the intended cell with probability 0.5, if there is a cell in the direction of movement. Otherwise, there is no move and the state remains unchanged. Hence, state transitions are stochastic. There is a cost of one for each action taken in a nongoal state. An admissible heuristic estimate of the optimal cost of each state is the length of a deterministic path from the state to the goal. The steps that LAO* takes to solve this problem are shown in (b) through (e) of Fig. 2. LAO* terminates when a solution graph is found, without needing to consider states 5, 6, or 7.

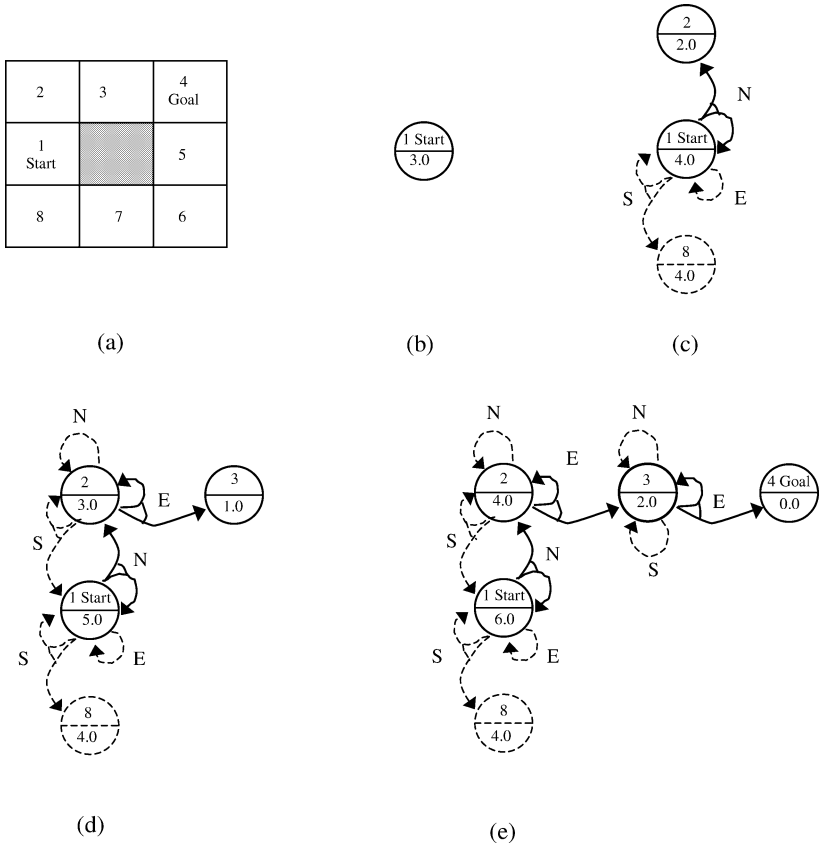

Fig. 2. (a) shows the grid for the stochastic shortest-path problem. (b) shows the root node of the explicit graph (corresponding to the start state) at the beginning of the search. (c) shows the result of expanding the start state. (d) shows the result of expanding state 2, which is the only state on the fringe of the best partial solution graph. (e) shows the result of expanding state 3. Solid lines show the best partial solution graph, and dashed lines show the rest of the explicit graph. The estimated cost of each state is shown in the lower half of the node.

Admissibility LAO* shares the properties of AO* and other heuristic search algorithms. Given an admissible heuristic evaluation function, all state costs in the explicit graph are admissible after each step and LAO* converges to an optimal or ε-optimal solution without (necessarily) evaluating all problem states. We first prove this for the case in which policy iteration is used in the cost revision step. Then we prove it for the case in which value iteration is used in the cost revision step.

Theorem 1. *If the heuristic evaluation function h is admissible and policy iteration is used to perform the cost revision step of LAO*, then*:

(1) $f(i) \leqslant f^*(i)$ *for every state i, after each step of LAO**;

(2) $f(i) = f^*(i)$ *for every state i of the best solution graph, when LAO* terminates*;

(3) *LAO* terminates after a finite number of iterations.*

Proof. (1) The proof is by induction. Every state $i \in G$ is assigned an initial heuristic cost estimate and $h(i) \leqslant f^*(i)$ by the admissibility of the heuristic evaluation function. The solution expansion step does not change the cost of any states and so it is sufficient to consider the cost revision step. We make the inductive assumption that at the beginning of this step, $f(i) \leqslant f^*(i)$ for every state $i \in G$. If all the tip states of G' have optimal costs, then all the nontip states in G' must converge to their optimal costs when policy iteration is performed on them by the convergence proof for policy iteration. But by the induction hypothesis, all the tip states of G' have admissible costs. It follows that the nontip states in G' must converge to costs that are as good or better than optimal when policy iteration is performed on them.

(2) The search algorithm terminates when the best solution graph for s is complete, that is, has no unexpanded states. For every state i in this solution graph, it is contradictory to suppose $f(i) < f^*(i)$ since that implies a complete solution that is better than optimal. By (1) we know that $f(i) \leqslant f^*(i)$ for every state in G'. Therefore $f(i) = f^*(i)$.

(3) It is obvious that LAO* terminates after a finite number of iterations if the implicit graph G is finite, or equivalently, the number of states in the MDP is finite. (When the state set is not finite, it may still converge in some cases.) □

Theorem 2. *If the heuristic evaluation function h is admissible and value iteration is used to perform the cost revision step of LAO*, then*:

(1) *$f(i) \leqslant f^*(i)$ for every state i at every point in the algorithm*;

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

Proof. (1) The proof is by induction. Every state $i \in G$ is assigned an initial heuristic cost estimate and $f(i) = h(i) \leqslant f^*(i)$ by the admissibility of the heuristic evaluation function. We make the inductive hypothesis that at some point in the algorithm, $f(i) \leqslant f^*(i)$ for every state $i \in G$. If a backup is performed for any state i,

$$\begin{aligned} f(i) &= \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right] \\ &\leqslant \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f^*(j) \right] = f^*(i), \end{aligned}$$

where the last equality restates the Bellman optimality equation.

(2) Because the graph is finite, LAO* must eventually find a solution graph that has no nonterminal tip states. Performing value iteration on the states in this solution graph makes the error bound of the solution arbitrarily small after a finite number of iterations, by the convergence proof of value iteration for stochastic shortest-path problems [2]. □

Convergence test If the cost revision step of LAO* computes exact state costs (for example, using policy iteration), LAO* converges when the best solution graph does not have any nonterminal tip states. This is the same convergence test used by AO*. It guarantees that the solution to which LAO* converges is optimal. When state costs are updated approximately (for example, using value iteration), then when LAO* finds a

solution graph that does not have any nonterminal tip states, this solution is not necessarily optimal. In this case, an additional convergence test can be used to determine when the solution is ε-optimal.

The convergence test is simple and is summarized in step 3 of Table 5. It consists of performing value iteration on the set of states visited by the current best solution graph. Because value iteration may change the best action for some state, the best solution graph may change from one iteration to the next. If the best solution graph changes to that it includes a nonterminal tip state, control is passed from the convergence test back to the main algorithm so that the best partial solution can be expanded and re-evaluated. When the best solution graph does not contain any nonterminal tip states, its error bound can be computed. The convergence test is repeated until the error bound falls below ε. When it does, LAO* terminates with an ε-optimal solution.

The error bound is the difference between an upper and lower bound on the optimal cost of the start state of the solution. (This is different from the error bound for dynamic programming, which is the maximum difference over all states, not just the start state.) Because the current estimated cost of the start state is admissible, it serves as a lower bound. An upper bound on the optimal cost of the start state is equal to $f^{\pi}(s) + \phi^{\pi}(s)r$, where $f^{\pi}(s)$ is the estimated cost of the start state under the current policy π, $\phi^{\pi}(s)$ is the mean first passage time from the start state to any goal state under this policy, and $r = \max_{i \in SG} |f'(i) - f(i)|$, is the Bellman residual for this iteration of the convergence test, where $SG \subseteq S$ denotes the set of states visited by the current best solution graph. Because the estimated cost of every state in the explicit graph is admissible, the Bellman residual only needs to be computed for the states visited by the best solution in order to determine the error bound of the solution. Similarly, first passage times only need to be computed for states in the solution graph.

We note that while Barto et al. [1] prove that RTDP converges to an optimal solution, they do not describe a convergence test for RTDP or a method for computing an error bound. The convergence test and error bound described here may also be adapted for use by RTDP.

Relationship to RTDP Both AO* and LAO*, as summarized in Tables 4 and 5, update the evaluation function for all ancestor states (along marked arcs) of an expanded state, whenever the best partial solution graph is expanded. They do so in order to maintain accurate estimates of state costs for every state in the explicit graph. Updating the evaluation function for all states in the explicit graph can be time consuming, however, especially for LAO*, which uses a more complex dynamic programming algorithm. Therefore, it is useful to point out that LAO* only needs to update the evaluation function for the ancestor states of an expanded state that are in the best partial solution graph, in order to ensure eventual convergence to an optimal or ε-optimal solution. To make this change to LAO*, as summarized in Table 5, we simply change the phrase "explicit graph" to "best partial solution graph" in step 2(b)i of the algorithm. We described LAO* as we originally did to emphasize its similarity to AO*. (But if we made this same change to AO*, it would also still converge to an optimal solution.)

With this modification, the previous convergence theorems for LAO* still hold, by the following reasoning. Consider any state i in the explicit graph that is not also in the best

partial solution graph. If $f(i)$ is not updated, its value will be less than it would be if it were updated—and therefore, it will still be admissible. Since $f(i)$ is less than it would be if it were updated, it is more likely that it will be selected for inclusion in the best partial solution graph. If it is not, it would not have been part of the best partial solution graph even if it had been updated. If it is selected for inclusion in the best partial solution graph, then $f(i)$ will be updated.

The observation that LAO* only needs to update f for states in the current best solution graph, in order to converge to an optimal solution, relates LAO* more closely to RTDP, which only updates f for states that can be reached from the start state by following the current best solution.

4. Extensions

While RTDP is a real-time search algorithm and LAO* is an off-line search algorithm, both solve the same class of MDPs and both converge to an optimal solution without evaluating all problem states. An advantage of deriving LAO* from the heuristic search algorithm AO* is that it makes it easier to generalize refinements of classic heuristic search algorithms for use in solving MDPs more efficiently. To illustrate this, we describe some extensions of LAO* that are suggested by its derivation from AO* (and ultimately, from A*). We conclude by briefly discussing the extension of LAO* to other MDPs besides stochastic shortest-path problems.

4.1. Pathmax

In A* search, a heuristic evaluation function h is said to be *consistent* if it is both admissible and $h(i) \leqslant c_i(a) + h(j)$ for every state $i \in S$ and action $a \in A(i)$, where j is the successor state of state i when action a is taken. Consistency is a desirable property because it ensures that state costs increase monotonically as the algorithm converges. If a heuristic evaluation function is admissible but not consistent, state costs need not increase monotonically. However, they can be made to do so by adding a *pathmax* operation to the update step of the algorithm, as first suggested by Mérõ [15]. This common improvement of A* and AO* can also be made to LAO*.

For problems with stochastic state transitions, we generalize the definition of consistency as follows: a heuristic evaluation function h is said to be *consistent* if it is both admissible and $h(i) \leqslant c_i(a) + \sum_j p_{ij}(a)h(j)$ for every state $i \in S$ and action $a \in A(i)$, where j is a possible successor state of state i when action a is taken. A pathmax operation is added to LAO* by changing the formula used to update state costs as follows:

$$f(i) := \max\left[f(i), \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right] \right]. \tag{10}$$

As it does for A* and AO*, use of the pathmax operation in LAO* can reduce the number of states that are expanded to find an optimal solution, when the heuristic is not consistent.

4.2. Evaluation function decomposition

The A* search algorithm relies on a familiar $f = g + h$ decomposition of the evaluation function. Chakrabarti et al. [3] have shown that a similar decomposition of the evaluation function is possible for AO* and that it supports a weighted heuristic version of AO*. We first show that this $f = g + h$ decomposition of the evaluation function can be extended to LAO*. Then we describe a similar weighted heuristic version of LAO*.

Each time a state i is generated by A*, a new estimate $f(i)$ of the optimal cost from the start state to a goal state is computed by adding $g(i)$, the cost-to-arrive from the start state to i, to $h(i)$, the estimated cost-to-go from i to a goal state. Each time a state i is generated by AO* or LAO*, the optimal cost from the start state to a goal state is also re-estimated. In this case, it is re-estimated by updating the estimated cost of every ancestor state of state i, including the start state. For the start state s, the estimated cost-to-go, denoted $f(s)$, is decomposed into $g(s)$, the cost-to-arrive from the start state to a state on the fringe of the best partial solution, and $h(s)$, the estimated cost-to-go from the fringe of the best partial solution to a goal state. In other words, $g(s)$ represents the part of the solution cost that has been explicitly computed so far and $h(s)$ represents the part of the solution cost that is still only estimated. The same decomposition applies to every other state in the explicit graph.

For each state i, the values $f(i)$, $g(i)$, and $h(i)$ are maintained, with $f(i) = g(i) + h(i)$. After a state i is generated by AO* or LAO*, and before it is expanded, $g(i)$ equals zero and $f(i)$ equals $h(i)$. After state i is expanded, $f(i)$ is updated in the same manner as before:

$$f(i) := \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right]. \tag{11}$$

Having determined the action a that optimizes $f(i)$, then $g(i)$ and $h(i)$ are updated as follows:

$$g(i) := c_i(a) + \sum_{j \in S} p_{ij}(a) g(j), \tag{12}$$

$$h(i) := \sum_{j \in S} p_{ij}(a) h(j). \tag{13}$$

Once LAO* converges to a complete solution, note that $h(i) = 0$ and $f(i) = g(i)$ for every state i in the solution.

The possibility of decomposing the evaluation function of an MDP in this way elegantly reflects the interpretation of an MDP as a state-space search problem that can be solved by heuristic search. It also makes it possible to create a weighted heuristic that can improve the efficiency of LAO* in exchange for a bounded decrease in the optimality of the solution it finds.

4.3. Weighted heuristic

Pohl [20] first described how to create a weighted heuristic and showed that it can improve the efficiency of A* in exchange for a bounded decrease in solution quality. Given

the familiar decomposition of the evaluation function, $f(i) = g(i) + h(i)$, a weight w, with $0.5 \leqslant w \leqslant 1$, is used to create a weighted heuristic, $f(i) = (1 - w)g(i) + wh(i)$. Use of this heuristic guarantees that solutions found by A* are no more than a factor of $w/(1-w)$ worse than optimal [4]. Given a similar $f = g + h$ decomposition of the evaluation function computed by LAO*, it is straightforward to create a weighted version of LAO* that can find an ε-optimal solution by evaluating a fraction of the states that LAO* might have to evaluate to find an optimal solution.

Note that a $f = g + h$ decomposition of the evaluation function and a weighted heuristic can also be used with RTDP. In fact, the idea of weighting a heuristic in order to find a bounded-optimal solution is explored by Ishida and Shimbo [9] for Korf's LRTA* algorithm [11], which can be regarded as a special case of RTDP for solving deterministic search problems. They create a weighted heuristic by multiplying the initial heuristic estimate of each state's value by a weight. (This approach is less flexible than one that relies on the $f = g + h$ decomposition we have just described, however, because it does not allow the weight to be adjusted in the course of the search.) They find that a weighted heuristic can significantly reduce the number of states explored (expanded) by LRTA* without significantly decreasing the quality of the solution it finds.

4.4. Heuristic accuracy and search efficiency

In all heuristic search algorithms, three sets of states can be distinguished. The *implicit graph* contains all problem states. The *explicit graph* contains those states that are evaluated in the course of the search. The *solution graph* contains those states that are reachable from the start state when the best solution is followed. The objective of a best-first heuristic search algorithm is to find an optimal solution graph while generating as small an explicit graph as possible.

Like all heuristic search algorithms, the efficiency of LAO* depends crucially on the heuristic evaluation function that guides the search. The more accurate the heuristic, the fewer states need to be evaluated to find an optimal solution, that is, the smaller the explicit graph generated by the search algorithm. For A*, the relationship between heuristic accuracy and search efficiency has been made precise. Given two heuristic functions, h_1 and h_2, such that $h_1(i) \leqslant h_2(i) \leqslant f^*(i)$ for all states i, the set of states expanded by A* when guided by h_2 is a subset of the set of states expanded by A* when guided by h_1 [18].

A result this strong does not hold for AO*, or by extension, for LAO*. The reason it does not is that the selection of the next state to expand on the fringe of the best partial solution graph is nondeterministic. Although AO* and LAO* work correctly no matter which state on the fringe of the best partial solution is expanded next, a particular choice may result in some states being expanded that would not be if the choice were different. Nevertheless, Chakrabarti et al. [3] show that a weaker result does hold for AO* and it is straightforward to extend this result to LAO*.

Chakrabarti et al. [3] consider the worst-case set of states expanded by a search algorithm. Adapting their analysis, let W denote the worst-case set of states expanded by LAO*, defined as follows:

- the start state s is in W;

- a state i is in W if there exists a partial solution graph p with an evaluation function f^p such that:
  - $f^p(s) \leqslant f^*(s)$;
  - every nontip state of p is in W;
  - i is a nonterminal tip state of p;
- no other states are in W.

Given this definition, we have the following theorem.

Theorem 3. *Given two heuristic functions, h_1 and h_2, such that $h_1(i) \leqslant h_2(i) \leqslant f^*(i)$ for all states i, the worst-case set of states expanded by LAO* when guided by h_2 is a subset of the worst-case set of states expanded by LAO* when guided by h_1.*

Proof. For any partial solution graph for start state s, we have

$$f_1(s) = g(s) + h_1(s),$$
$$f_2(s) = g(s) + h_2(s).$$

Since $h_1(s) \leqslant h_2(s)$, we have $f_1(s) \leqslant f_2(s)$. Thus, if $f_2(s) \leqslant f^*(s)$, we also have $f_1(s) \leqslant f^*(s)$. It follows that the worst-case set of states expanded by LAO* when guided by h_2 must be a subset of the worst-case set of states expanded by LAO* when guided by h_1. □

In other words, a more accurate heuristic does not necessarily make LAO* more efficient. However, it makes it more efficient in the worst case. This result holds for a "pure" version of LAO* that updates state costs exactly in the dynamic-programming step. If LAO* updates state costs approximately, for example, by using value iteration with a relaxed criterion for convergence, the effect on the set of states expanded in the worst case is not as clear.

4.5. Infinite-horizon problems

We have presented the results of this paper for stochastic shortest-path problems. Many MDPs do not include terminal (or goal) states, or if they do, there is no guarantee that a terminal state can be reached from every other state. Problems that may never terminate are called infinite-horizon MDPs. For an infinite-horizon problem, one way to ensure that every state has a finite expected cost is to discount future costs by a factor β, with $0 \leqslant \beta < 1$. For problems for which discounting is not reasonable, other optimality criteria, such as average cost per transition, can be adopted. Variations of policy and value iteration algorithms for these other optimality criteria have been developed [2]. By incorporating them into LAO*, we claim that the results of this paper can be extended to infinite-horizon problems in a straightforward way. Hansen and Zilberstein [8] consider LAO* for discounted infinite-horizon MDPs and give convergence proofs.

5. Performance

We now examine the performance of LAO* on two test problems. The first is the race track problem used by Barto et al. [1] to illustrate the behavior of RTDP. The second is a simple example of a supply-chain management problem that has been studied extensively in operations research [12]. While further work is needed to fully evaluate LAO*, the two problems allow us to observe some important aspects of its behavior and to improve its implementation.

Race track problem To illustrate the behavior of RTDP, Barto et al. [1] describe a test problem that involves a simple simulation of a race car on a track. The race track has any length and shape, and includes a starting line at one end and a finish line at the other. The track is discretized into a grid of square cells, where each cell represents a possible location of the car. Beginning at the starting line, the car attempts to move along the track to the finish line. The state of the MDP is determined by the location of the car and its two-dimensional velocity. The car can change its velocity by ± 1 in each dimension, for a total of nine actions. The actual acceleration/deceleration that results from each action is zero with some probability, in order to make state transitions stochastic. It is as if an attempt to accelerate or decelerate sometimes fails because of an unpredictable slip on the track. If the car hits the track boundary, it is moved back to the start state. For a full description of the problem, we refer to the paper by Barto et al. [1].

We test LAO* on an instance of this problem that has 21,371 states. When the race car it driven optimally, it avoids large parts of the track as well as dangerous velocities. Thus, only 2,248 states are visited by an optimal policy that starts at the beginning of the race track. Table 6 summarizes the performance of LAO* on this problem. It shows the number of states evaluated by LAO* as a function of two different admissible heuristics. The "zero heuristic" sets the initial heuristic cost of each state to zero. The "shortest-path heuristic" is computed by beginning from the set of goal states and determining the smallest possible number of state transitions needed to reach a goal state, for each state. This more informed

Table 6
Results for race track problem with 21,371 states. Optimal solution visits only 2,248 states (Policy iteration is much slower than value iteration because exact policy evaluation has cubic complexity in the number of states, compared to the quadratic complexity of value iteration.)

Algorithm	Solution quality	States evaluated	Time to converge
Policy iteration	Optimal	21,371	$>$ 10 minutes
Value iteration	Optimal	21,371	15.7 sec
LAO* w/ zero heuristic	Optimal	18,195	10.7 sec
LAO* w/ shortest-path heuristic	Optimal	14,252	4.7 sec
LAO* w/ weight $= 0.6$	Optimal	8,951	3.1 sec
LAO* w/ weight $= 0.67$	$+4\%$	4,508	1.8 sec

heuristic enables LAO* to find an optimal solution by expanding fewer states. Weighting the shortest-path heuristic allows a solution to be found by evaluating even fewer states, although the solution is not necessarily optimal. These results illustrate the importance of heuristic accuracy in limiting the number of states evaluated by LAO*.

The number of states evaluated by LAO* is only one factor that influences its efficiency. Our experiments quickly revealed that a naive implementation of LAO* can be very inefficient, even if it evaluates a fraction of the state space. The reason for this is that the running time of LAO* is proportional to the total number of states evaluated multiplied by the average number of times these states are evaluated. In a naive implementation of LAO*, the fact that fewer states are evaluated can be offset by an increase in the average number of evaluations of these states. Using the race track problem as an example, expanding just one state on the fringe of the best partial solution graph at a time, and performing several iterations of value iteration between each state expansion, results in many states being evaluated tens of thousands of times before convergence.

We found that the performance of LAO* can be improved by using various techniques to limit the number of times that states are evaluated. One technique is to expand multiple states on the fringe of the best partial solution graph before performing the cost revision step. (Nilsson [18] makes a similar suggestion for AO*.) For this problem, expanding all states on the solution fringe before performing the cost revision step worked better than expanding any subset of the fringe states (although we do not claim that this is the best strategy in general.) We also found it helpful to limit the number of iterations of value iteration in each cost revision step, and to limit the number of ancestor states on which value iteration is performed. Because value and policy iteration are much more computationally expensive than expanding the best partial solution graph, finding ways to minimize the expense of the cost revision step usually improves the performance of LAO*.

With this in mind, Table 7 summarizes a more efficient implementation of LAO* that limits the number of state backups by integrating them into the solution expansion step. Beginning from the start state, LAO* searches the best partial solution graph in a depth-first fashion. As each state is visited, including any newly expanded states on the fringe of the solution graph, the state is pushed on the stack used to control the depth-first search. (The state is also marked in order to avoid revisiting the same state during the depth-first search.) As states are popped off the stack, a backup is performed to re-evaluate them. It

Table 7
An efficient version of LAO* that combines backups with solution expansion

1. The explicit graph G' initially consists of the start state s.
2. *Expand best partial solution, update state costs, and mark best actions*: While the best solution graph has some nonterminal tip state, perform a depth-first search of the best partial solution graph. For each visited state i, in postorder traversal:
   (a) If state i is not expanded, expand it.
   (b) Set $f(i) := \min_{a \in A(i)} [c_i(a) + \sum_j p_{ij}(a) f(j)]$ and mark the best action for i. (When determining the best action resolve ties arbitrarily, but give preference to the currently marked action.)
3. *Convergence test*: Perform value iteration on the states in the best solution graph. Continue until one of the following two conditions is met. (i) If the error bound falls below ε, go to step 4. (ii) If the best solution graph changes so that it has an unexpanded tip state, go to step 2.
4. Return an ε-optimal solution graph.

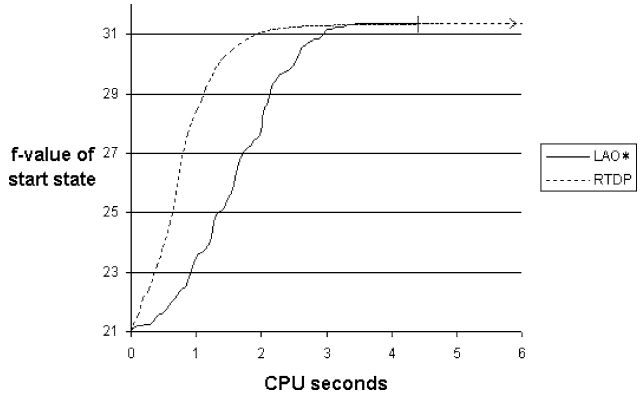

Fig. 3. Comparison of the convergence behavior of RTDP and LAO* on the race track problem. Both algorithms use the shortest-path heuristic, which initializes the start state to 21.0. Although RTDP improves the solution more quickly, LAO* converges to an ε-optimal solution after four and a half seconds, indicated by the vertical mark, whereas RTDP continues to run for more than three minutes without converging to the same value.

is possible to view this version of LAO* as a kind of "focused value iteration" because it works the same as value iteration except that instead of updating all states each iteration, it only updates states that are part of the best (partial) solution graph. LAO* expands the best partial solution graph each iteration until a complete and optimal solution is found. The last column of Table 6 shows convergence times for this more efficient implementation of LAO*. (All timing results in this section are on a 300 MHz UltraSparc II.)

On the race track problem, the performance of this version of LAO* is competitive with RTDP, as shown by Fig. 3. Trial-based RTDP improves the solution faster because it focuses on high-probability paths through the state space, while LAO* systematically explores all paths, including low-probability paths. However, LAO* converges to an ε-optimal solution after only four and a half seconds (with $\varepsilon = 0.000001$), whereas RTDP does not converge to a solution within 0.001 of this value after three minutes. The reason for this is that RTDP tends to avoid updating states on very low-probability paths, and thus is slow in propagating the low-probability state values needed for convergence.

This comparison suggests a combined strategy that focuses search on high-probability paths first, to improve the solution quickly, and then searches more systematically as the solution nears convergence. LAO* allows flexibility in how state expansions are mixed with state backups, and more sophisticated search control strategies that exploit this flexibility may further improve its performance.

Supply-chain problem Our second test problem is a simple supply-chain management problem. A supply chain is a network of facilities (companies or departments) that are responsible for procurement of materials, intermediate and finished product assembly, and distribution of finished products to customers. Supply chains, with various degrees of complexity, exist in both service and manufacturing. Supply-chain management requires making decisions about the acquisition of raw materials; the manufacture, distribution, and sale of a product; and optimal levels of inventory at various stages of the chain

of production. Such problems require sequential decision making under uncertainty (especially uncertainty about customer demand), and typically have very large state spaces [12].

We consider a small example of such a problem. To manufacture a product, say a lamp, we are dependent on supply of a part, say a flexible neck. Manufacture of the lamp depends on other parts as well, such as a lamp shade, bulb, basement, and cable, but we consider only supply of one part for this example. Demand for the product is stochastic. The problem is to determine how many units of the product to produce, what price to sell the product for, how many units of the part to order, and what inventory to keep on hand. A complicating factor is that the cost of parts goes down if they are ordered in advance. Each order has a guaranteed delivery time. The longer the period between ordering and delivery, the lower the cost. This reflects the fact that advance notice allows the part manufacturer to increase its own efficiency. We first specify the parameters of the problem. Then we describe the performance of LAO*.

The problem state is defined by a tuple of state variables,

$$(\mathit{ordAmt}, \mathit{ordTime}, \mathit{invProd}, \mathit{invPart}, \mathit{demand}, \mathit{price}),$$

where *ordAmt* is the quantity of the part on order for each of the next *ordTime* time periods, *invProd* is the current inventory of the product, *invPart* is the current inventory of the part, *demand* defines a demand curve that gives the current demand for the product as a function of price, and *price* is the current price of the product. We assume that each of these state variables has a finite number of possible values, beginning from a minimum value and increasing by a fixed step size to a maximum possible value. The size of the state space is the cardinality of the cross-product of the domains of the state variables.

An action consists of two decisions. The first is to (possibly) change the price of the final product. The second is to (possibly) place a new order for parts. Each time period, a maximal number of new products is produced based on the available parts and the production capacity. Therefore, control of production is performed indirectly by limiting the order of new parts. Once parts are available, we assume that it is beneficial to transform them as soon as possible into complete products. To relax this assumption, one could add to each action a third component that controls the production level.

To limit the number of possible actions (in a reasonable way), we allow the price of the product to be increased or decreased by only the fixed step size each time period. The number of actions and states is limited by the way we represent future orders. The most general representation is an arbitrary vector of orders $[\mathit{ordAmt}_1, \mathit{ordAmt}_2, \mathit{ordAmt}_3, \ldots, \mathit{ordAmt}_n]$, where ordAmt_i represents the number of parts on order for delivery after i time periods. Instead, we assume the same number of parts, *ordAmt*, is ordered for delivery in each of the next *ordTime* time periods. We also assume that existing orders form a commitment that cannot be canceled or changed. An action can only increase *ordAmt* or *ordTime*, or both, up to some maximum. If no more parts are ordered, *ordTime* decreases by one each time period until it reaches zero. Then it is possible to order any number of parts for any number of periods.

As for the state transition function, we assume that actions have deterministic effects, but that demand is stochastic. Demand for the product is given by the formula

$$\mathit{demand}(\mathit{price}) = \mathit{demand} \cdot \mathrm{e}^{\alpha \cdot \mathit{price}},$$

where α is a constant that characterizes this product, and *demand* is a state variable that changes stochastically each time period. With probability 0.3, it increases by one unit; with probability 0.3, it decreases by one unit; and with probability 0.4, it remains unchanged. The transition probabilities change when the demand variable is at its maximum or minimum value. At its maximum value, it decreases with probability 0.4 and otherwise remains unchanged. At its minimum value, it increases with probability 0.4 and otherwise remains unchanged.

We treat this as a reward-maximization problem (which is a simple modification of a cost-minimization problem). The overall reward is the difference between the income from selling products and the associated costs. In each action cycle, the immediate income is the number of products sold times the price of a product. The immediate cost has three components: production cost, inventory storage cost, and the cost of ordering parts. The former two are linear functions defined by the unit production cost, *CostProd*, and the unit storage costs for inventory, *InvCostPart* and *InvCostProd*. The cost of an order depends on a variable unit cost for a part, which is a monotonically decreasing function of time the part is ordered in advance. The minimum cost, *MinCostPart*, applies if the part is ordered at least t_{min} time periods in advance. The maximum cost, *MaxCostPart*, applies if the part if ordered less than t_{max} time periods in advance. The cost changes linearly between t_{max} and t_{min}.

The objective is to maximize the discounted sum of rewards over an infinite horizon. For discounted infinite-horizon problems, a simple admissible heuristic can be constructed by assuming the maximum one-step reward is received each time period. We consider an instance of the problem in which the variables can have the values shown in Table 8, and the constants are: $InvCostProd = 1.0$; $InvCostPart = 0.2$; $ProductionCost = 1.80$; $ProductionCapacity = 1500$; $t_{max} = 1$; $t_{min} = 4$; and $\alpha = 0.207944$. The discount factor is 0.95.

This instance of the supply-chain problem has 36,000 states and 27 actions. We solve the problem for the following start state: $ordAmt = 100$; $ordTime = 2$; $price = 10.00$; $demand = 4000$; $invProd = 600$; and $invPart = 0$. An optimal policy visits only 680 states and is found by expanding 14,229 states after 48 seconds. By comparison, value iteration takes 238 seconds to solve the same problem.

Table 8
Variables for the supply-chain problem

Description	Variable name	Min value	Max value	Increment step
Order level	*ordAmt*	0	1800	600
Order periods	*ordTim*	1	4	1
Product price	*price*	5	17.50	1.25
Demand	*demand*	1000	16000	1500
Prod inventory	*invProd*	0	100	2000
Part inventory	*invPart*	0	400	200
Cost of part	*costPart*	1.40	2.00	N/A

We have experimented with additional problem instances created by varying the parameters of this problem and found that LAO* consistently outperforms value iteration. It is well-suited for this supply-chain problem because, regardless of the starting state, an optimal policy is to try to enter a certain steady state of production and orders, and this steady state involves a small fraction of the state space (less than 2% in the above example). While it is difficult to generalize these results to other problems, or even to more complex supply-chain problems, they improve our intuition about the performance of LAO* and show that it can be a useful algorithm.

For what types of problem is LAO useful?* Given a start state and an admissible heuristic to guide forward expansion of a partial solution, LAO* can find an optimal solution without evaluating the entire state space. However, it only enjoys this advantage for problems for which an optimal solution visits a fraction of the state space, beginning from the start state. Many problems have this property, including our two test problems. But many do not. It would be useful to have a way of identifying problems for which LAO* is effective. There does not appear to be an easy answer to this question. Nevertheless, we make a few remarks in this direction.

One relevant factor appears to be how much overlap there is between the sets of successor states for different actions, taken in the same state. If there is little overlap, a policy can more easily constrain the set of reachable states. If there is much (or even complete) overlap, it cannot. For example, each action in the race track problem has two possible successor states. The action of accelerating or decelerating either succeeds or fails. (It fails due to a slip on the track. This is the stochastic element of the problem.) Thus, the overlap between the successor states for different actions is limited to the case of failure. Otherwise, each action has a distinct successor state. Now, consider the following slight modification of the race track problem. With some small probability, each action has the effect of any other action. In this case, every action has the same set of possible successor states and the choice of one action instead of another does not affect the set of reachable states. With this change to the race track problem, every policy visits all the states of the race track problem.

Even for problems for which an optimal policy visits all (or most) of the state space, LAO* may provide a useful framework for finding partial solutions under time constraints. Instead of finding a complete policy, it can focus computation on finding a policy for a subset of states that is most likely to be reached from the starting state. This approach to solving MDPs approximately has been explored by Dean et al. [5]. The framework of LAO* relates this approach more closely to classic heuristic search algorithms.

6. Conclusion

We have presented a generalization of AO*, called LAO*, that can find solutions with loops, and we have shown that it can be used to solve indefinite-horizon MDPs. Like other heuristic search algorithms, LAO* can find an optimal solution for a start state without evaluating the entire state space.

Although real-time search has been applied to MDPs before [1,5,6,21], LAO* is the first off-line heuristic search algorithm for MDPs. The derivation of LAO* from the classic heuristic search algorithm AO* provides a foundation for a heuristic search approach to MDPs. It also makes it easier to generalize enhancements of classic search algorithms for use in solving MDPs more efficiently. The results presented in this paper illustrate the relevance of the rich body of AI research on heuristic search to the problem of solving MDPs more efficiently. They also help to clarify the relationship between classic AI search techniques and dynamic programming algorithms developed in operations research.

Acknowledgements

We thank Dietrich Eisenhardt for implementing the supply chain example and running the experiments on it. We thank the anonymous reviewers for helpful comments. Support for this work was provided in part by the National Science Foundation under grant IIS-9984952 to the first author and grants IRI-9624992 and IRI-9634938 to the second author.

References

[1] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1995) 81–138.
[2] D. Bertsekas, Dynamic Programming and Optimal Control, Athena Scientific, Belmont, MA, 1995.
[3] P.P. Chakrabarti, S. Ghose, S.C. DeSarkar, Admissibility of AO* when heuristics overestimate, Artificial Intelligence 34 (1988) 97–113.
[4] H.W. Davis, A. Bramanti-Gregor, J. Wang, The advantages of using depth and breadth components in heuristic search, in: Z.W. Ras, L. Saitta (Eds.), Methodologies for Intelligent Systems Vol. 3, North-Holland, Amsterdam, 1989, pp. 19–28.
[5] T. Dean, L.P. Kaelbling, J. Kirman, A. Nicholson, Planning under time constraints in stochastic domains, Artificial Intelligence 76 (1995) 35–74.
[6] R. Dearden, C. Boutilier, Integrating planning and execution in stochastic domains, in: Proc. Tenth Conference on Uncertainty in Artificial Intelligence, Washington, DC, 1994, pp. 162–169.
[7] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (1985) 505–536.
[8] E.A. Hansen, S. Zilberstein, Heuristic search in cyclic AND/OR graphs, in: Proc. AAAI-98, Madison, WI, 1998, pp. 412–418.
[9] T. Ishida, M. Shimbo, Improving the learning efficiencies of realtime search, in: Proc. AAAI-96, Portland, OR, 1996, pp. 305–310.
[10] P. Jiménez, C. Torras, An efficient algorithm for searching implicit AND/OR graphs with cycles, Artificial Intelligence 124 (2000) 1–30.
[11] R. Korf, Real-time heuristic search, Artificial Intelligence 42 (1990) 189–211.
[12] Z. Lian, A. Deshmukh, J. Wang, Optimal frozen period in a periodic review inventory model with updateable demand forecasts, in: Proc. 15th International Conference on Production Research, 1999.
[13] A. Martelli, U. Montanari, Additive AND/OR graphs, in: Proc. IJCAI-73, Stanford, CA, 1973, pp. 1–11.
[14] A. Martelli, U. Montanari, Optimizing decision trees through heuristically guided search, Comm. ACM 21 (12) (1978) 1025–1039.
[15] L. Mérõ, A heuristic search algorithm with modifiable estimate, Artificial Intelligence 23 (1) (1984) 13–27.
[16] N.J. Nilsson, Search problem-solving and game-playing trees for minimal cost solutions, in: A. Morrell (Ed.), Information Processing 68, Vol. 2, North-Holland, Amsterdam, 1969, pp. 1556–1562.
[17] N.J. Nilsson, Problem Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

[18] N.J. Nilsson, Principles of Artificial Intelligence, Tioga Publishing, Palo Alto, CA, 1980.
[19] K.R. Pattipati, M.G. Alexandridis, Application of heuristic search and information theory to sequential fault diagnosis, IEEE Trans. Systems Man Cybernet. 20 (4) (1990) 872–887.
[20] I. Pohl, First results on the effect of error in heuristic search, Machine Intelligence 5 (1973) 219–236.
[21] J. Tash, S.J. Russell, Control strategies for a stochastic planner, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 1079–1085.