Heuristic Search in Cyclic AND/OR Graphs

Eric A. Hansen and Shlomo Zilberstein
Computer Science Department
University of Massachusetts
Amherst, MA 01003
{hansen,shlomo}@cs.umass.edu

Abstract

Heuristic search algorithms can find solutions that take the form of a simple path (A*), a tree or an acyclic graph (AO*). We present a novel generalization of heuristic search (called LAO*) that can find solutions with loops, that is, solutions that take the form of a cyclic graph. We show that it can be used to solve Markov decision problems without evaluating the entire state space, giving it an advantage over dynamic-programming algorithms such as policy iteration and value iteration as an approach to stochastic planning.

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 operators that map states to successor states, a start state, and a set of goal states. The objective is to find a sequence of operators that transforms the start state into a goal state and also optimizes some measure of the cost, or merit, of the solution.

Two well-known heuristic search algorithms for solving state-space search problems are A* and AO* (Nilsson 1980). A* finds a solution that takes the form of a sequence of operators leading from a 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. However no heuristic search 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. For example, a loop in a solution to a theorem-proving problem represents circular reasoning. A loop in a solution to a problem-reduction problem represents a failure to reduce it to primitive subproblems. However there are some problems for which it does make sense for a solution to contain loops. These include problems that can be formalized as Markov decision processes (MDPs), a framework widely used for stochastic planning in artificial intelligence (Dean *et al.* 1995; Barto *et al.* 1985; Tash and Russell 1994; Dearden and Boutilier 1997). A stochastic planning problem includes operators (or actions) that transform a state into one of several possible successor states, with each possible state transition occurring with some probability. A solution is usually cast in 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 operator 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.")

A policy for an MDP can be found using a dynamic programming algorithm such as policy iteration or value iteration. 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 finds a policy for a particular starting state and uses 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 particular 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 (Dechter and Pearl 1985) and a related result has been established for AO* (Chakrabarti *et al.* 1988). In this paper, we generalize heuristic search to find solutions with loops and show that the resulting algorithm can solve stochastic planning problems that are formalized as MDPs without evaluating the entire state space.

Background

We begin by reviewing AND/OR graphs and the heuristic search algorithm AO* for solving problems formalized as acyclic AND/OR graphs. We then briefly review MDPs and show that they can be formalized as cyclic AND/OR graphs.

AND/OR graphs

We formalize a state-space search problem as a graph in which each node represents a problem state and each arc represents the application of an operator to a state. Let S denote the set of all possible states; in this paper, we assume it is finite. Let $s \in S$ denote a start state that corresponds to the root of the graph and let $S^G \subseteq S$ denote a set of goal states that occur at the leaves of the graph. Let A denote a finite set of operators (or actions) and let $A(i)$ denote the set of operators applicable to state i.

Following Martelli and Montanari (1978) and Nilsson (1980), we view an AND/OR graph as a hypergraph. Instead of arcs that connect pairs of nodes as in an ordinary graph, a hypergraph has *hyperarcs* or *k-connectors* that connect a node to a set of k successor nodes. 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. Here we interpret a k-connector as a stochastic operator that transforms a state into one of k possible successor states. Let $p_{ij}(a)$ denote the probability that applying operator a to state i results in a transition to state j. A similar interpretation of AND/OR graphs is made by Martelli and Montanari (1978) and Pattipati and Alexandridis (1990), among others.

In AND/OR graph search, a "solution" is a generalization of the concept of a path in an ordinary graph. Starting from the start node, it selects exactly one operator (outgoing connector) for each node. Because a connector can have multiple successor nodes, a solution can also be viewed as a subgraph called a *solution graph.* Every directed path in the solution graph terminates at a goal node.

We assume a cost function assigns a cost to each hyperarc; let $c_i(a)$ denote the cost for the hyperarc that corresponds to applying operator a to state i. We also assume each goal state has a cost of zero. The cost of a solution graph for a given state is defined recursively as the sum of the cost of applying the operator prescribed for that state and the weighted sum of the cost of the solution graphs for each of its successor states, where the weight is the probability of each state transition. A minimal-cost solution graph is found by solving the following system of recursive equations,

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

where f^* denotes the optimal *cost-to-go function* and $f^*(i)$ is the optimal cost for state i. For an acyclic AND/OR graph, a special dynamic programming algorithm called backwards induction solves these equations efficiently by evaluating each state exactly once in a backwards order from the leaves to the root.

AO*

Unlike dynamic programming, heuristic search can find an optimal solution graph without evaluating the entire state space. Therefore a graph is not usually supplied explicitly to a search algorithm. We refer to G as the *implicit graph*; it is specified implicitly by a start node s and a successor function. The search algorithm works on an *explicit graph*, G', that initially consists only of the start node. A tip or leaf node of the explicit graph is said to be terminal if it is a goal node and nonterminal otherwise. A nonterminal tip node can be expanded by adding to the explicit graph its outgoing connectors and any successor nodes not already in the explicit graph.

Heuristic search works by repeatedly expanding the best partial solution until a complete solution is found. A *partial solution graph* is a subgraph of the explicit graph that starts at s and selects exactly one hyperarc for each node. It is defined similarly to a solution graph, except that a directed path may end at a nonterminal tip node. For every nonterminal tip node i of a partial solution graph, we assume there is an admissible heuristic estimate $h(i)$ of the minimal-cost solution graph for it. A heuristic evaluation function h is said to be *admissible* if $h(i) \leq f^*(i)$ for every node i. We can recursively calculate an admissible heuristic estimate $f(i)$ of the optimal cost of any node i in the explicit graph as follows:

$$f(i) = \begin{cases} 0 \text{ if } i \text{ is a goal node} \\ h(i) \text{ if } i \text{ is a nonterminal tip node} \\ \text{else } min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right] \end{cases}$$

```
1. The explicit graph $G'$ initially consists of the start node $s$.

2. Forward search: Expand the best partial solution graph as follows:
   (a) Identify the best partial solution graph and its nonterminal tip nodes by searching forward from the start state and following the marked action for each state.
   (b) If the best partial solution graph has no nonterminal tip nodes, goto 4.
   (c) Else expand some nonterminal tip node $n$ and add any new successor nodes to $G'$. For each new tip node $i$ added to $G'$ by expanding $n$, if $i$ is a goal node then $f(i) = 0$; else $f(i) = h(i)$.

3. Dynamic programming: Update state costs as follows:
   (a) Identify the ancestors in the explicit graph of expanded node $n$ and create a set $Z$ that contains the expanded node and all its ancestors.
   (b) Perform backwards induction on the nodes in $Z$ by repeating the following steps until $Z$ is empty.
       i. Remove from $Z$ a node $i$ such that no descendent of $i$ in $G'$ occurs in $Z$.
       ii. Set $f(i) := \min_{a \in A(i)} \left[c_i(a) + \sum_j p_{ij}(a) f(j)\right]$ and mark the best action for $i$. (When determining the best action resolve ties arbitrarily, but give preference to the currently marked action.)
   (c) Goto 2.

4. Return the solution graph.
```

Figure 1: AO*

Figure 1 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 dynamic programming step that uses backwards induction. As with all heuristic search algorithms, three classes of states can be distinguished. The implicit graph contains all possible states. The explicit graph contains all states that are generated and evaluated at some point in the course of the search. The solution graph contains those states that are reachable from the start state when a optimal solution is followed.

The version of AO* we have outlined is described by Martelli and Montanari (1973). Others have described slightly different versions of AO* (Martelli and Montanari 1978; Nilsson 1980; Bagchi and Mahanti 1983). One difference is to use a pathmax operation in step (3bii), as follows:

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

If the heuristic is admissible but not consistent, this ensures that state costs increase monotonically. Another difference is to try to limit the number of ancestors on which dynamic programming is performed by not considering the ancestors of a node unless the cost of the node has changed and the node can be reached by marked connectors. To simplify exposition, we have also omitted from our outline of AO* a solve-labeling procedure that is usually included to improve efficiency. Briefly, a node is labeled solved if it is a goal node or if all of its successor nodes are labeled solved. Labeling nodes as solved improves the efficiency of the forward search step of AO* because it is unnecessary to search below a solved node for nonterminal tip nodes.

Markov decision processes

MDPs are widely used in artificial intelligence as a framework for decision-theoretic planning and reinforcement learning. Here we note that an infinite-horizon MDP can be formalized in a straightforward way as a cyclic AND/OR graph. The cycles in the graph make infinite-horizon behavior possible. Let each node of the graph correspond to a state of the MDP and let each k-connector correspond to an action with k possible outcomes. The transition probability function and cost function defined earlier are the same as those for MDPs. A solution to an MDP generally takes the form of a mapping from states to actions, δ, called a policy.

Closely related to heuristic search problems are a class of infinite-horizon MDPs called *stochastic shortest-path problems* (Bertsekas 1995). (The name reflects an interpretation of costs as arc lengths.) Stochastic shortest-path problems have a start state and a set of absorbing states that can be used to model goal states. A policy is said to be *proper* if it ensures the goal state is reached from any state with probability 1.0. For a proper policy, the undiscounted infinite-horizon cost for each state i is finite and can be computed by solving the following system of $|S|$ equations in $|S|$ unknowns:

$$f^{\delta}(i) = c_i(\delta(s)) + \sum_{j \in S} p_{ij}(\delta(s)) f^{\delta}(j). \qquad (1)$$

In the rest of this paper we make the simplifying assumption that all possible policies are proper. The results of this paper do not depend on this assumption. When it cannot be made, other optimality criteria – such as discounted cost over an infinite horizon or average cost per transition – can be adopted to ensure every state has a finite expected cost under every policy (Bertsekas 1995).

A policy δ is said to dominate a policy δ' if $f^{\delta}(i) \leq f^{\delta'}(i)$ for every state i. An optimal policy dominates every other policy and its cost-to-go function, f^*, satisfies the following Bellman optimality equation:

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

Policy iteration is a well-known method for solving infinite-horizon MDPs. After evaluating a policy using equation (1), it improves it by performing the following operation for each state i:

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

Policy evaluation and policy improvement are repeated until the policy cannot be improved, which signifies that it is optimal. Another algorithm for solving MDPs is value iteration. Each iteration, it improves the estimated cost-to-go function f by performing the following operation for each state i,

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

However policy iteration and value iteration must evaluate all states to find an optimal policy. Therefore they can be computationally prohibitive for MDPs with large state sets. We try to overcome this limitation by using heuristic search to limit the number of states that must be evaluated.

LAO*

LAO* is a simple generalization of AO* that can find solutions with loops. Like AO*, it has two principal steps: a forward search step and a dynamic programming step. The forward search step is the same as in AO* except that it allows a solution graph to contain loops. Forward search of a partial solution graph now terminates at a goal node, a nonterminal tip node, or a loop back to an already expanded node of the current partial solution graph.

The problem with allowing a solution graph to contain loops is that the backwards induction algorithm of the dynamic programming step of AO* can no longer be applied. However dynamic programming can still be performed by using policy iteration or value iteration algorithms for infinite-horizon MDPs. This simple generalization of AO* creates the algorithm LAO* that is summarized in Figure 2. In the rest of this section we discuss some of the issues that must be considered to implement it efficiently.

1. The explicit graph G' initially consists of the start node s.
2. *Forward search:* Expand the best partial solution graph as follow:.
   (a) Identify the best partial solution graph and its nonterminal tip nodes by searching forward from the start state and following the marked action for each state.
   (b) If the best partial solution graph has no nonterminal tip nodes, goto 4.
   (c) Else expand some nonterminal tip node n and add any new successor nodes to G'. For each new tip node i added to G' by expanding n, if i is a goal node then $f(i) = 0$; else $f(i) = h(i)$.
3. *Dynamic programming:* Update state costs as follows:
   (a) Identify the ancestors in the explicit graph of expanded node n and create a set Z that contains the expanded node and all its ancestors.
   (b) Perform policy iteration on the nodes in set Z until convergence or else perform value iteration on the nodes in set Z for one or more iterations. Mark the best action for each state. (When determining the best action resolve ties arbitrarily, but give preference to the currently marked action.)
   (c) Goto 2.
4. Return the solution graph.

Figure 2: LAO*

Policy iteration

We begin by considering the use of policy iteration to perform the dynamic programming step of LAO*. The advantage of using policy iteration is that it computes an exact cost for each node of the explicit graph, based on the heuristic estimates at the tip nodes.

Policy iteration is performed on the set of nodes that includes the expanded node and all of its ancestors in the explicit graph. Some of these nodes may have successor nodes that are not in this set of nodes but are still part of the explicit graph; in other words, policy iteration is not necessarily (or usually) performed on the entire explicit graph. The costs of these successor nodes can be treated as constants in the dynamic programming step because they cannot be affected by any change in the cost of the expanded node or its ancestors. The dynamic programming step of AO* exploits this reasoning as well.

Performing policy iteration on this set of nodes may change the best action for some states and, by doing so, change the best partial solution graph; the backwards induction algorithm of AO* can have the same effect. Because multiple iterations of policy iteration may be necessary to converge, it is important to stress that policy iteration must be performed on all of the nodes in this set until convergence. This is necessary to ensure that all nodes in the explicit graph have exact, admissible costs, including those that are no longer part of the best partial solution graph.

It is straightforward to show that 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 policy without (necessarily) evaluating the entire state space.

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

1. *$f(i) \leq f^*(i)$ 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 node $i \in G$ is assigned an initial heuristic cost estimate and $h(i) \leq f^*(i)$ by the admissibility of the heuristic evaluation function. The forward search step expands the best partial solution graph and does not change the cost of any nodes and so it is sufficient to consider the dynamic programming step. We make the inductive assumption that at the beginning of this step, $f(i) \leq f^*(i)$ for every node $i \in G$. If all the tip nodes of G' have optimal costs, then all the nontip nodes 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 nodes of G' have admissible costs. It follows that the nontip nodes in G' must converge to costs that are as good or better than optimal when policy iteration is performed on them only.

(2) The search algorithm terminates when the best solution graph for s is complete, that is, has no unexpanded nodes. 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 $f(i) \leq f^*(i)$ for every node 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.) □

Because policy iteration is initialized with the current state costs, it may converge quickly. Nevertheless it is a much more time-consuming algorithm than the backward induction algorithm used by AO*. The backwards induction algorithm of AO* has only linear complexity in the size of the set of nodes on which dynamic programming is performed. Each iteration of policy iteration has cubic complexity in the size of this set of nodes and more than one iteration may be needed for policy iteration to converge.

Value iteration

An alternative is to use value iteration in the dynamic programming step of AO*. A single iteration of value iteration is computationally equivalent to the backwards induction algorithm of AO* and states can be evaluated in a backwards order from the expanded node to the root of the graph to maximize improvement. However the presence of loops means that state costs are not exact after value iteration. Therefore LAO* is no longer guaranteed to identify the best partial solution graph or to expand nodes in a best-first order. This disadvantage may be offset by the improved efficiency of the dynamic programming step, however, and it is straightforward to show that state costs remain admissible and converge in the limit to optimality.

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

1. *$f(i) \leq f^*(i)$ for every node i at every point in the algorithm*
2. *$f(i)$ converges to $f^*(i)$ in the limit, for every node i of the best solution graph*

Proof: (1) The proof is by induction. Every node $i \in G$ is assigned an initial heuristic cost estimate and $f(i) = h(i) \leq f^*(i)$ by the admissibility of the heuristic evaluation function. We make the inductive hypothesis that at some point in the algorithm, $f(i) \leq f^*(i)$

for every node $i \in G$. If a value iteration update is performed for any node i,

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

where the last equality restates the Bellman optimality equation.

(2) Because the graph is finite, LAO* must eventually find a complete solution graph. In the limit, the nodes is this solution graph must converge to their exact costs by the convergence proof for value iteration. The solution graph must be optimal by the admissibility of the costs of all the nodes in the explicit graph. □

When value iteration is used, convergence to optimal state costs is asymptotic. If bounds on optimal state costs are available, however, it may be possible to detect convergence to an optimal solution after a finite number of steps by using the bounds to prune actions that can be proved suboptimal.

Forward search

We briefly mention some ways in which the efficiency of LAO* can be affected by the forward search step.

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

It is also possible to expand several nodes at a time in the forward search step. This risks expanding some nodes unnecessarily but can improve performance when the dynamic programming step is more expensive than the forward search step.

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. Dearden and Boutilier (1997) describe a form of abstraction for MDPs that can create admissible heuristics of varying degrees of accuracy.

An ϵ-admissible version of AO* has been described that increases the speed of AO* in exchange for a bounded decrease in solution quality (Chakrabarti *et al.* 1988). An analogous ϵ-admissible version of LAO* may find an ϵ-optimal solution by evaluating a fraction of the states that LAO* would have to evaluate to find an optimal solution.

For some problems it may be possible to store all nodes visited by the best solution graph in memory, but impossible to store the entire explicit graph in memory. For such problems, it may be useful to create a memory-bounded version of LAO* modeled after memory-bounded versions of AO* (Chakrabarti *et al.* 1989).

Related Work

LAO* closely resembles some recently developed algorithms for solving stochastic planning problems formalized as MDPs.

Barto, Bradtke, and Singh (1995) describe an algorithm called real-time dynamic programming (RTDP) that generalizes Korf's learning real-time heuristic search algorithm (LRTA*) to MDPs (Korf 1990). They show that under certain conditions, RTDP converges (asymptotically) to an optimal solution without evaluating the entire state space. This parallels the principal result of this paper and LAO* and RTDP solve the same class of problems. The difference is that RTDP relies on trial-based exploration – a concept adopted from reinforcement learning – to explore the state space and determine the order in which to update state costs. By contrast, LAO* finds a solution by systematically expanding a search graph in the manner of heuristic search algorithms such as A* and AO*.

Dean *et al.* (1995) 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 very 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 also been explored (Tash and Russell 1994; Dearden and Boutilier 1997). 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 of these algorithms are motivated by the problem of search (or planning) in real-time and both allow it to be interleaved with execution; the time constraint on search is often the time before the next action needs to be executed. Both Dean *et al.* (1995) and Tash and Russell (1994) describe decision-theoretic approaches to optimizing the value of search in the interval between actions. These algorithms can be viewed as real-time counterparts of LAO*. In fact, the relationship between LAO*, the envelope approach to policy and value iteration, and RTDP mirrors (closely, if not exactly) the relationship between A*, RTA*, and LRTA* (Korf 1990). Thus LAO* fills a gap in the taxonomy of search algorithms.

RTDP and the related envelope approach to policy and value iteration represent a solution as a mapping from states to actions, albeit an incomplete mapping called a partial policy; this reflects their derivation from dynamic programming. LAO* represents a solution as a cyclic graph (or equivalently, a finite-state controller), a representation that generalizes the graphical representations of a solution used by search algorithms like A* (a simple path) and AO* (an acyclic graph); this reflects its derivation from heuristic search. The advantage of representing a solution in the form of a graph is that it exhibits reachability among states explicitly and makes analysis of reachability easier.

Conclusion

We have presented a simple generalization of AO*, called LAO*, that can find solutions with loops. It can be used to solve state-space search problems that are formalized as cyclic AND/OR graphs, a class of problems that includes MDPs as an important case. Like other heuristic search algorithms, LAO* can find an optimal solution for a given start state without evaluating the entire state space.

LAO* has been implemented and tested on several small MDPs. Future work will study the factors that affect its efficiency by testing it on various large MDPs. The principal contribution of this paper is conceptual. It provides a foundation for recent work on how to solve MDPs more efficiently by focusing computation on a subset of states reachable from a start state. Our derivation of LAO* from AO* clarifies the relationship of this work to heuristic search. It also suggests that a rich body of results about heuristic search may be generalized in an interesting way for use in solving MDPs more efficiently.

Acknowledgments.

Support for this work was provided in part by the National Science Foundation under grants IRI-9624992, IRI-9634938 and INT-9612092.

References

Bagchi, A. and Mahanti, A. 1983. Admissible Heuristic Search in AND/OR Graphs. *Theoretical Computer Science* 24:207–219.

Barto, A.G.; Bradtke, S.J.; and Singh, S.P. 1995. Learn to Act using Real-Time Dynamic Programming. *Artificial Intelligence* 72:81–138.

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

Chakrabarti, P.P.; Ghosh, S.; & DeSarkar, S.C. 1988. Admissibility of AO* When Heuristics Overestimate. *Artificial Intelligence* 34:97-113.

Chakrabarti, P.P; Ghosh, S.; Acharya, A.; & DeSarkar, S.C. 1989. Heuristic Search in Restricted Memory. *Artificial Intelligence* 47:197-221.

Dean, T.; Kaelbling, L.P.; Kirman, J.; and Nicholson, A. 1995. Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence* 76:35–74.

Dearden, R and Boutilier, C. 1997. Abstraction and Approximate Decision-Theoretic Planning. *Artificial Intelligence* 89:219–283.

Dechter, R. and Pearl, J. 1985. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM* 32:505–536.

Korf, R. 1990. Real-Time Heuristic Search. *Artificial Intelligence* 42:189–211.

Martelli, A. and Montanari, U. 1973. Additive AND/OR Graphs. In Proceedings of the Third International Joint Conference on Artificial Intelligence, 1–11. Stanford, CA.

Martelli, A. and Montanari, U. 1978. Optimizing Decision Trees Through Heuristically Guided Search. *Communications of the ACM* 21(12):1025–1039.

Nilsson, N.J. 1980. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga Publishing Company.

Pattipati, K.R. and Alexandridis, M.G. 1990. Application of Heuristic Search and Information Theory to Sequential Fault Diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics* 20(4):872–887.

Tash, J. and Russell, S. 1994. Control Strategies for a Stochastic Planner. In Proceedings of the Twelth National Conference on Artificial Intelligence, 1079–1085. Seattle, WA.