AO* Revisited

Valentina Bayer Zubek [*], Thomas G. Dietterich

School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331-3102, USA

Abstract

This paper revisits the AO* algorithm introduced by Martelli and Montanari [1] and made popular by Nilsson [2]. The paper's main contributions are: (1) proving that the value of a node monotonically increases as the AO* search progresses, (2) proving that selective updates of the value function in the AO* algorithm are correct, and (3) providing guidance to researchers interested in the AO* implementation. (1) and (2) are proven under the assumption that the heuristic used by AO* is consistent. The paper also reviews the use of AO* for solving Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs).

Key words: Heuristic search, AO* implementation, AND/OR graph, admissible heuristic, consistent heuristic, MDP, POMDP

1 Introduction

The AO* algorithm introduced by Martelli and Montanari[1], and presented in more detail by Nilsson [2], is a well-known technique for solving AND/OR graphs. AO* has been widely applied to solve problems such as symbolic integration [3], theorem proving [2, chapter 6], game analysis, learning diagnostic policies [4]. This paper introduces a systematic framework for describing the AO* algorithm, using notations inspired from the MDP framework. We prove that AO* converges to the optimal solution graph of the AND/OR graph, if provided with an admissible heuristic. The proof is not new, however, being first presented by Martelli and Montanari in [1]. That paper also observed that given an additional assumption of consistency (or monotonicity) for the

[*] Corresponding author.
Email addresses: bayer@cs.orst.edu (Valentina Bayer Zubek), tgd@cs.orst.edu (Thomas G. Dietterich).

heuristic, the value of each node in the graph increases with more iterations, and the algorithm can perform more efficient updates. These claims (that also appear in [2] and in all further papers about AO* implementation) were never proved before, and this paper fills the gap by proving them. This paper is also aimed to guide researchers interested in an efficient AO* implementation.

Section 2 introduces the definitions and notations for AND/OR graphs. We give an overview of the AO* algorithm in Section 3, and we define the notions of admissible and consistent heuristics, along with the properties they induce on the value function in the graph. We also detail the implementation of the AO* algorithm. Section 4 proves that AO* with an admissible heuristic converges to the optimal solution graph, and that AO* with a consistent heuristic also converges but performs more efficient updates to the value function. We conclude by reviewing the AO* literature in Section 5, including the application of the AO* algorithm to solving MDPs and POMDPs with a given start state.

2 AND/OR Graphs

An AND/OR graph alternates between two types of nodes, OR nodes and AND nodes. We restrict our attention to acyclic graphs (in fact, DAGs), where a node is either an OR node or an AND node. Each OR node has an arc (or 1-connector) to each of its successor AND nodes. Each AND node has a k-connector to the set of its k successor OR nodes. The names come from the fact that an OR node solution requires the selection of only one of its successors, while an AND node solution requires the solution of all of its successors.

2.1 Examples of AND/OR Graphs

AND/OR graphs were studied in the early 70s to represent the problem-reduction approach to problem-solving, in applications such as symbolic integration, theorem proving, and game analysis. In problem-reduction search, the original problem is transformed into several subproblems. A problem corresponding to an OR node is solved when at least one of its successor subproblems is solved. A problem corresponding to an AND node is solved when all of its successor subproblems are solved. For example, to find a counterfeit coin from a set of 4 coins, in the corresponding OR node we could choose to weigh either a pair of coins (one coin in each pan of the scale), or all 4 coins (two in each pan). To solve one of these two AND nodes we need to solve all 3 OR nodes corresponding to the weighing outcomes: balance tips left, balance tips right, and balance is steady.

A Markov Decision Process (MDP) is a mathematical model describing the interaction of an agent with an environment. An MDP is defined by a set of states S, a set of (stochastic) actions A, the transition probabilities $P_{tr}(s'|s,a)$ of moving from state s to state s' after executing action a, and the costs associated with these transitions. An AND/OR graph for an MDP has OR nodes corresponding to states s, and AND nodes corresponding to state-action pairs (s, a).

2.2 Definitions and Notations

We will introduce notations and definitions for an arbitrary AND/OR graph G. The AO* algorithm constructs an AND/OR graph incrementally, starting from the root node; this graph is called the *explicit graph*, and is a subgraph of the entire search graph (the *implicit graph*). In other words, during the search process AO* can find the optimal solution without doing exhaustive search. In the following, G is the entire graph (implicit graph).

By definition, the (implicit) finite AND/OR graph G consists of a *root* OR node (start node) and a *successor function*. There are a total of N OR nodes. The successor function for an OR node n is one AND node (n, a), corresponding to the arc (1-connector) denoted by a. We will refer to these arcs as *actions*. The set of all actions is called A, and the set of actions valid in OR node n is denoted $A(n)$. If there are m actions in $A(n)$, then OR node n has m AND node successors. The successor function for an AND node (n, a) is given by a k-connector which points to the set of k OR nodes n'.

Terminal nodes are OR nodes corresponding to goal nodes. We assume there is a predicate $P(n)$ that returns true when node n is a goal node. There are no valid actions in a terminal node, so $A(n) = \emptyset$.

Figure 1 presents a simple example of an AND/OR graph.

During the AO* description, it will be useful to ignore AND nodes and view the AND/OR graph as a DAG of OR nodes. For this purpose, we will say that OR node n_1 is a *parent* of OR node n_2 if n_2 can be reached from n_1 by traversing exactly one AND node. Abusing notation, we will also call n_2 a *successor* of n_1. If at least one AND node must be traversed to get from n_1 to n_2, then n_2 is a *descendant* of n_1, and n_1 is an *ancestor* of n_2.

At the very beginning, the explicit graph consists of the root OR node and its *unexpanded AND nodes* (that is, the successors of these AND nodes were not yet generated). The explicit graph is grown by selecting one of these AND nodes for expansion, and generating all its successors. The k-connector and the successor OR nodes (along with their unexpanded AND nodes) are added to

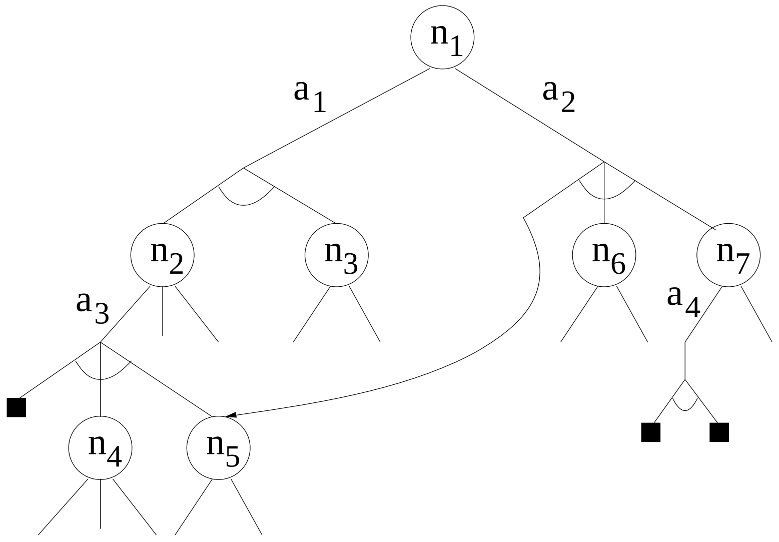

Fig. 1. A simple example of an AND/OR graph. OR nodes are enclosed in circles (e.g., $n_1, n_2, \ldots, n_7$). AND nodes are marked by arcs (e.g., (n_1, a_1), (n_2, a_3), (n_7, a_4)) and specify the outcomes of actions. OR nodes and AND nodes succeed each others in the graph. The terminal nodes are marked by solid squares. OR node n_5 can be reached twice from the root n_1, by following the path of AND nodes (n_1, a_1), (n_2, a_3), and also from the AND node (n_1, a_2).

the graph. The implicit graph is generated after all AND nodes are expanded, therefore its leaves are terminal OR nodes.

The OR node whose unexpanded AND node the AO* algorithm will select for expansion is called a *fringe* node. Note that an OR node may have both expanded and unexpanded AND nodes.

Abusing the notation, we will sometimes say "expanding the action a in node n" instead of "expanding AND node (n, a)" (which generates the successor OR nodes of (n, a)).

As the graph grows, some of the successors OR nodes of AND nodes may have been generated before, and therefore the corresponding branch of the k-connector only needs to be pointed toward these existing nodes (see Figure 1). This makes G a graph, instead of a tree. For example, in the problem of learning diagnostic policies (see Section 5.3), multiple paths from the root lead to the same OR node by changing the order of the tests.

The AND/OR graph is *acyclic*, by definition: starting from any OR node, there is no sequence of actions that leads back to the node.

By definition, a terminal OR node (one on which the goal predicate is true) is *solved*. We can recursively define an AND node to be solved, when all its successors OR nodes are solved. A (nonterminal) OR node is solved if at least one of its successor AND nodes is solved.

A *solution graph* of an AND/OR graph G is a subgraph of G, such that: the root OR node of G belongs to the solution graph; each OR node in the solution graph has only one of its successor AND nodes in the solution graph; each AND node in the solution graph has all its successor OR nodes in the solution graph. For brevity of notation, we will call a solution graph a *policy*, denoted π. For an OR node n, the policy specifies the action a to be taken, $\pi(n) = a$. The policy is undefined in a terminal OR node. For an AND node (n, a), the policy specifies all its successors OR nodes. A policy is finite, and all its leaves are terminal OR nodes; such a policy is *complete* (all its AND nodes are expanded). Before the AO* algorithm terminates, the policy may be *incomplete* or *partial*, if it contains unexpanded AND nodes. The leaves of an incomplete policy are OR nodes n; some leaves may be terminal nodes, but there is at least one non-terminal node, with unexpanded AND node $(n, \pi(n))$. In general, a policy is a graph, not a tree.

A *positive cost* $C(n, a)$ is associated with each AND node (n, a), and a *positive weight* $w(n, a, n')$ is associated with the branch of the connector from the AND node to its successor OR node n'. Special cases for the weights are: all weights are 1.0, for problem-solving, i.e. $w(n, a, n') = 1, \forall n, a, n'$; weights correspond to transition probabilities for MDPs, i.e., $w(n, a, n') = P_{tr}(s'|s, a)$, with $\sum_{s'} P_{tr}(s'|s, a) = 1, \forall s, a$, where node n corresponds to MDP state s and node n' corresponds to one of the resulting MDP states s' after executing action a in state s.

We assume that the weights and costs are given.

Solving an AND/OR graph means finding the policy of minimum cost. Using notations inspired from MDPs, we call the minimum cost function of an OR node n its value function $V^*(n)$, and the minimum cost function of an AND node (n, a) its Q-function $Q^*(n, a)$. These cost functions are recursively defined as: $V^*(n) = \min_{a \in A(n)} Q^*(n, a)$, $Q^*(n, a) = C(n, a) + \sum_{n'} w(n, a, n') \times V^*(n')$, where n' are the successor OR nodes of AND node (n, a). If OR node n is a terminal node, its value function is zero, $V^*(n) = 0$. The minimum cost function of the root node is also called the *optimal value function* V^* of the AND/OR graph. There could be more than one policy whose cost is V^*; such a policy is called an *optimal policy* π^*; in an OR node, an optimal policy selects an action $\pi^*(n) = \text{argmin}_{a \in A(n)} Q^*(n, a)$.

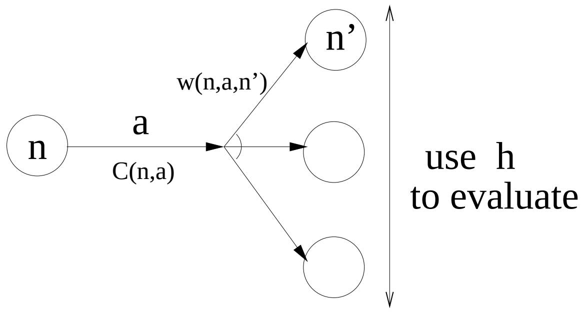

Fig. 2. The admissible heuristic $Q(n,a)$ for unexpanded AND node (n,a) is computed using one-step lookahead and admissible heuristic h to evaluate its successor OR nodes n', $Q(n,a) = C(n,a) + \sum_{n'} w(n,a,n') \cdot h(n')$.

3 AO* Algorithm

3.1 Overview of the AO Algorithm*

The AO* algorithm computes an optimal solution graph (policy) of an AND/OR graph, given an admissible heuristic [1,2]. A heuristic is admissible if it never over-estimates the optimal cost of getting from a node to terminal nodes. If in addition the heuristic satisfies the consistency (monotonicity) property, then the algorithm becomes more efficient in terms of the number of node updates.

During its search, AO* considers incomplete policies in which not all AND nodes have been expanded. The AO* algorithm repeats the following steps: in the current policy π, it selects an AND node to expand; it expands it (by generating its successor OR nodes); and then it recomputes (bottom-up) the optimal value function and policy of the revised graph. The algorithm terminates when all leaf OR nodes of the policy are terminal (so there are no more unexpanded AND nodes in the policy). The complete policy computed upon convergence is an optimal policy π^*.

We now introduce the notion of admissible heuristic.

3.2 Admissible Heuristic

Definition 3.1 *Heuristic h is admissible if for any OR node n it under-estimates the cost of the optimal policy under this node, i.e. $h(n) \leq V^*(n)$.*

The admissible heuristic is positive, $h(n) \geq 0$, and since terminal nodes have $V^*(n) = 0$, it implies that the heuristic estimates for terminal nodes is also zero, $h(n) = 0$.

We can extend the admissible heuristic to provide an under-estimate, $Q(n,a)$, of the optimal cost of an unexpanded AND node (n,a). This extension is based on a one-step lookahead (see Figure 2), and it computes $Q(n,a) = C(n,a) + \sum_{n'} w(n,a,n') \cdot h(n')$, where n' iterates over the successor OR nodes of AND node (n,a). Because $h(n') \leq V^*(n')$, it follows that $Q(n,a) \leq Q^*(n,a)$, so the heuristic for unexpanded AND nodes is admissible. Because the weights w and the admissible heuristic h are positive, we also have that $Q(n,a) \geq C(n,a)$.

The definition of the admissible heuristic $Q(n,a)$ can be extended to apply to all AND nodes as follows:

$$Q(n,a) = \begin{cases} C(n,a) + \sum_{n'} w(n,a,n') \cdot h(n'), \text{ if } (n,a) \text{ is unexpanded} \\ C(n,a) + \sum_{n'} w(n,a,n') \cdot V(n'), \text{ if } (n,a) \text{ is expanded} \end{cases}$$

where $V(n) \overset{\text{def}}{=} \min_{a \in A(n)} Q(n,a)$ is the value function of OR node n, with $V(n) = 0$ if n is terminal. This defines value functions $V(n)$ and $Q(n,a)$, based on the admissible heuristic h, for every OR node n and every AND node (n,a) in the (explicit) graph. We can also define the policy whose value function is $V(n)$, as $\pi(n) = \operatorname{argmin}_{a \in A(n)} Q(n,a)$.

Theorem 3.1 proves that Q and V form an admissible heuristic. Their values are less than the optimal value functions for all iterations of AO* (in the following, we did not burden the notation with an index for iteration).

Theorem 3.1 *If h is an admissible heuristic, then for all nodes n and all actions $a \in A(n)$, $Q(n,a) \leq Q^*(n,a)$, and $V(n) \leq V^*(n)$.*

Proof by induction:

Base case for Q:
If AND node (n,a) is unexpanded, then $Q(n,a) = C(n,a) + \sum_{n'} w(n,a,n') \cdot h(n') \leq C(n,a) + \sum_{n'} w(n,a,n') \cdot V^*(n') = Q^*(n,a)$, because $h(n') \leq V^*(n')$.

Base case for V:
For a terminal OR node, $V(n) = V^*(n) = 0$.

Let n_u be any OR node all of whose AND nodes (n_u, a) are unexpanded. Then $V(n_u) \leq V^*(n_u)$ follows from the base case of Q.

Any other OR node n_e in the graph has at least one AND node (n_e, a) previously expanded. Let n' be any of the successor OR nodes of (n_e, a).

Induction hypothesis:
If $V(n') \leq V^*(n'), \forall n'$, then $Q(n_e, a) \leq Q^*(n_e, a)$, $\forall a \in A(n_e)$, and $V(n_e) \leq V^*(n_e)$.

We only need to consider the case of an expanded action $a \in A(n_e)$, because the other situations are covered by the base case of Q. By definition, $Q(n_e, a) = C(n_e, a) + \sum_{n'} w(n_e, a, n') \cdot V(n')$. Because n' is one of the successor OR nodes of (n_e, a), we can apply the induction hypothesis for n', so $V(n') \leq V^*(n')$, hence $Q(n_e, a) \leq Q^*(n_e, a)$. It follows that $Q(n_e, a) \leq Q^*(n_e, a), \forall a \in A(n_e)$, and $V(n_e) \leq V^*(n_e)$. **Q.E.D.**

We now introduce the notion of consistent heuristic.

3.3 Consistent Heuristic

Definition 3.2 *Heuristic h is consistent (or monotonic) if for any OR node n and its successor OR node n',*

$$h(n) \leq C(n, a) + \sum_{n'} w(n, a, n') \times h(n'), \forall a \in A(n)$$

Trivial examples of a consistent heuristic $h(n)$ are the optimal value function $V^*(n)$ (by definition, $V^*(n) = min_{a \in A(n)} C(n, a) + \sum_{n'} w(n, a, n') \times V^*(n')$), and the zero heuristic, $h(n) \equiv 0$.

Theorem 3.2 *If heuristic h is consistent, then for every OR node n in the graph, $h(n) \leq V(n)$, where $V(n)$ is the value of node n for an arbitrary iteration of AO^*.*

Proof by induction:

Base case:

If OR node n is terminal, $h(n) = V(n) = 0$.

Induction hypothesis:
Let n' be any successor OR node of n. If $h(n') \leq V(n')$, then $h(n) \leq V(n)$.

Because h is consistent, $h(n) \leq C(n, a) + \sum_{n'} w(n, a, n') \times h(n'), \forall a \in A(n)$. We can apply the induction hypothesis for nodes n', so $h(n') \leq V(n')$. Therefore $h(n) \leq C(n, a) + \sum_{n'} w(n, a, n') \times h(n') \leq C(n, a) + \sum_{n'} w(n, a, n') \times V(n') = Q(n, a), \forall a \in A(n)$, which implies $h(n) \leq Q(n, a), \forall a \in A(n)$. Finally, $h(n) \leq min_{a \in A(n)} Q(n, a) = V(n)$. **Q.E.D.**

The following theorem establishes that a consistent heuristic is also admissible.

Theorem 3.3 *If heuristic h is consistent, then it is admissible (i.e., for every node n in the graph, $h(n) \leq V^*(n)$).*

Proof: Similar to the proof of Theorem 3.2, replacing V with V^*.

Corollary 3.4 *If heuristic h is consistent, then for every node n in the graph, $h(n) \leq V(n) \leq V^*(n)$.*

Proof: Because h is consistent, Theorem 3.2 provides that $h(n) \leq V(n)$. Theorem 3.3 guarantees that a consistent heuristic is admissible, so we can apply Theorem 3.1 for h admissible which says that $V(n) \leq V^*(n)$. It follows that $h(n) \leq V(n) \leq V^*(n)$. **Q.E.D.**

3.4 Pseudocode and Implementation Details for the AO* Algorithm

In our AO* implementation, we store more information than necessary, but this makes the description and the implementation clearer. We store Q, V, π, though it would be enough to store just the Q function, because the policy π and the V function can be computed from Q.

3.4.1 Data Structures

An OR node n stores

- the current policy, $\pi(n)$, and its value, $V(n)$,
- a flag that marks if this node is solved,
- a list of successor AND nodes (n, a), for all actions $a \in A(n)$
- a list of parent OR nodes, along with markers set to 1 if n is reached by $\pi(parent)$.

Because our graph is a DAG and we do not want to generate the same OR node multiple times, the OR nodes are stored in a hash table. Before generating a new OR node, we double check that such an OR node does not exist already. If it does, we only update the list of its parents.

An AND node (n, a) stores

- a flag that marks this node as expanded or not
- the Q-value, $Q(n, a)$,
- a list of its successor OR nodes n', and one weight $w(n, a, n')$ for each successor.

3.4.2 Pseudocode

Table 1 gives the pseudocode for the AO* algorithm. We describe each step of it. Step (1) is described in Table 2, step (4) in Table 3 and step (5) in Table 4.

Table 1
Pseudocode for the AO* algorithm.

```
function AO* returns a complete policy.
iteration $i = 0$;
create hash-table;
(1) create-OR-node(root, (OR Node) 0, hash-table);
(2) while (root not solved){
        // iteration $i + 1$
(3)     in current $\pi$, select fringe OR node $n$ with
            AND node $(n, a)$ to expand $(\pi(n) = a)$.
(4)     expand AND node $(n, a)$ by creating successor OR nodes $n'$.
(5)     do bottom-up updates of $Q, V, \pi$ for $n$ and its ancestors.
i++;
}
return last $\pi$.
```

Table 2
Creating a new OR node. If $n =$ root, there is no parent.

```
function create-OR-node(OR node $n$, OR node parent, hash-table).
if $n$ is terminal
    set $A(n) = \emptyset$, $solved = 1$, $V(n) = 0$, $\pi(n)$ to be undefined.
    no need to store any parents of $n$, nor to store $n$ in the hash table.
    create OR node $n$.
else
    for every action $a \in A(n)$
        compute $Q(n,a) = C(n,a) + \sum_{n'} w(n,a,n') \cdot h(n')$
        mark $(n,a)$ as unexpanded
        create AND node $(n,a)$
    $\pi(n) = \arg\min_{a \in A(n)} Q(n,a)$, $V(n) = \min_{a \in A(n)} Q(n,a)$.
    solved = 0;
    if (parent)
        add parent to the list of parents.
    create OR node $n$.
    store OR node $n$ in the hash-table.
```

Step(1): Creating a New OR Node

First we take care of the case when OR node n is terminal. Otherwise, for every action $a \in A(n)$ we compute $Q(n, a)$ using the heuristic h in the successor nodes n', and we create the unexpanded AND node (n, a). Then we compute the policy $\pi(n)$ and value function $V(n)$. The node n is not solved.

If the call to create-OR-node() specifies a parent, then we add this parent to the list of parents of the new OR node.

We finally create the new OR node n with policy $\pi(n)$, value $V(n)$, solved, list of successor AND nodes (if any), and list of parents (if any).

The root gets added to the hash-table as the first OR node created. For all other nonterminal OR nodes, we first check that they are not already in the hash-table, before storing them.

Step(2): OR Node Becomes Solved

In the following, we explain what it means for an OR node to be solved. This is a general concept, though in step (2) of Table 1 it is applied for the root node. An OR node becomes *solved* when all its successor OR nodes, reached by π, are solved. The base case for this recursive definition is a terminal node.

Definition 3.3 *An OR node is solved if it is a terminal OR node, or if it is an internal OR node and all its successor OR nodes, reached by π, are solved.*

Once an OR node n becomes solved, it stays solved, because its choice for π will not be changed, as we see in the code for selecting the AND node to expand. The policy under a solved node is a complete policy (because its leaves are terminal nodes), the best one in the subgraph rooted at this OR node. With a proof similar to Theorem 4.1, it can be shown by induction that for a solved OR node n, $V(n) = V^*(n)$, starting with the terminal nodes whose value is zero.

Step(3): Select AND Node to Expand

We traverse the current policy π (at iteration $i+1$, this is policy π_i) until we find a suitable leaf OR node n, with unexpanded $\pi(n)$. Node n is called a *fringe* node. For example, we could traverse the policy depth-first, keeping track of the leaf OR node n with unexpanded $\pi(n)$ having the largest value, $\max_n V(n)$; this is done in problem solving. In MDPs, one may choose the node with $\max_n V(n) \times P(n)$, where $P(n)$ is the probability of reaching node n from the root. This heuristic for node selection can be interpreted as expanding the AND node $(n, \pi(n))$ with the largest impact on the root.

We can stop the search down a branch of π if we find a solved OR node, because the policy under that node is complete (all its leaf OR nodes are terminal), therefore there are no unexpanded AND nodes in it. This proves that once an OR node becomes solved, it stays solved. It also makes the implementation more efficient.

It is also possible to expand more AND nodes per iteration.

Table 3
Expanding AND node (n, a) of fringe OR node n.

```
**function** expand-AND-node(fringe OR node $n$, hash-table).
a = $\pi(n)$.
for every successor OR node $n'$ of AND node $(n, a)$
  if (node $n'$ is already in the hash-table)
    add $n$ to the list of parent OR nodes of $n'$.
    if ($h$ is consistent)
      mark the link $n - n'$ as being part of $\pi$.
  else
    create-OR-node($n'$, $n$, hash-table).
  add OR node $n'$ to the list of successors for AND node $(n, a)$.
mark AND node $(n, a)$ as expanded.
```

We are interested in the leaf OR node n, though we expand its AND node $(n, \pi(n))$; the reason for this will become apparent in the description of the next steps of the AO* algorithm.

Since the root is not solved at the current iteration, there must be at least one unexpanded AND node in π, so the fringe node exists.

In general, note that an OR node n is a leaf with respect to a particular policy π, in the sense that the AND node $(n, \pi(n))$ is unexpanded, though in the graph G this OR node may have other successor AND nodes, already expanded.

Step(4): Expand AND Node

Table 3 shows the details for expanding an AND node (n, a). If the successor OR node n' was already created, then it is stored in the hash-table, and its parent fringe node is added to the list of parents for n'. If the heuristic is consistent, then we mark the connector from fringe to n' as being part of the policy π (this is useful for future updates of values and policy). If n' was not generated before, we need to create a new OR node n'. In either case, the OR node n' is added as a successor of the AND node (n, a). Finally we mark the AND node as expanded.

Step(5): Bottom-up Updates of Values and Policy

Let $V_i(n)$ be the value for OR node n at the end of iteration i. All nodes n in the graph whose values are not modified in iteration $i+1$ have $V_{i+1}(n) = V_i(n)$. The action set $A(n)$ of valid actions in any node n does not change from one iteration of AO* to another.

Table 4
Updating V, Q and π at iteration $i+1$ after the expansion of π in fringe OR node.

```
 **function** update(OR node fringe).
push fringe OR node onto the queue.
**while** (queue not empty){
   pop OR node $n$ from the queue.
   recompute $Q(n,a)$ for all expanded AND nodes $(n,a)$,
      $Q_{i+1}(n,a) := C(n,a) + \sum_{n'} w(n,a,n') \cdot V_{i+1}(n')$.
   for unexpanded AND nodes $(n,a)$,
      $Q_{i+1}(n,a) := Q_i(n,a)$.
   $\pi_{i+1}(n) := \arg\min_{a \in A(n)} Q_{i+1}(n,a)$, $V_{i+1}(n) := \min_{a \in A(n)} Q_{i+1}(n,a)$.
   **if** (AND node $(n, \pi_{i+1}(n))$ is expanded)
      **if** (all successor OR nodes of $(n, \pi_{i+1}(n))$ are solved)
         label OR node $n$ as solved.
      if ($h$ is consistent)
         mark all its successor OR nodes as being reachable by $\pi_{i+1}$ from node $n$.
   **if** $(\pi_i(n)) \neq \pi_{i+1}(n)))$
      if ($h$ is consistent)
         mark all the successor OR nodes of $(n, \pi_i(n))$ as unreachable by $\pi_{i+1}$
         from node $n$.
   **if** ($h$ is consistent)
      **if** ((OR node $n$ solved) or $(V_{i+1}(n) > V_i(n)))$
         push onto the queue all marked parents of OR node $n$.
   **else** // $h$ is admissible but not consistent
      **if** ((OR node $n$ solved) or $(V_{i+1}(n) \neq V_i(n)))$
         push onto the queue all parents of OR node $n$.
}
```

Table 4 details the updates of values and policies. A change in the graph is initiated by a change in the fringe node. Starting with the fringe node, we propagate changes in the values upward in the graph by pushing OR nodes, which can be affected by these changes, onto a queue. The changes propagate from bottom-up, so when updating the value of node n we already computed the value V_{i+1} of its successor OR nodes n'. Because the graph is a DAG, there are no loops in this bottom-up propagation.

Only the Q values for expanded AND nodes need to be updated, based on the already computed V values of their successors OR nodes. The Q values for unexpanded AND nodes were already computed, based on h for their successor OR nodes, and these Q values stay the same. For every OR node in the queue, it is necessary to perform the updates for all its expanded AND nodes, to take into account changes in value in their successors.

After updating the Q values, we recompute π and V. The current OR node n popped from the queue becomes solved when its successor OR nodes through

π are solved.

The next section will prove the correctness of AO* with selective updates, provided that the heuristic h is consistent. If h is admissible, but not consistent, we do not need to mark links, and we will have to update all ancestors of fringe node if the fringe became solved or if there is a change in its value. The following description (and also the markings between OR nodes in Tables 3 and 4) are assuming that h is consistent.

We first mark the successor OR nodes of $(n, \pi(n))$ as being reachable by π. If the policy has changed, we need to mark the successor OR nodes of n through the old policy π as being unreachable by the new π. It is important to have a marker between a successor OR node and its parent OR node, if the successor is reached from the parent by following π. These markers, which are similar to reversed links, will be useful after successor OR nodes update their V values and these changes need to be propagated to their marked parents, which will be pushed onto the queue.

If an OR node becomes solved, or its V value changes (which for consistent heuristic h can only be an increase in value, as proved in the next section, Theorem 4.4), then all its marked parents get pushed onto the queue. The connectors to the parent OR nodes were marked as being part of π in previous iterations of AO*. These parents, when their turn to be updated comes, will push their marked parents onto the queue, and this continues until the queue is empty.

A node can be pushed on the queue multiple times. AO* without selective updates will push all ancestors of a fringe node on the queue. Even with selective updates, a node can appear multiple times on the queue. Each time a node n is removed from the queue, we do a full Belmann update of its value function, that is, we recompute all $Q(n, a), \forall a \in A(n)$, and set $V(n) = min_{a \in A(n)} Q(n, a) = min_{a \in A(n)} C(n, a) + \sum_{n'} w(n, a, n') \times V(n')$. If node n appears $k > 1$ times on the queue, at iteration $i + 1$, it is possible that the l^{th} update of its value function, with $l < k$, uses value V_i for some of the successors n', if nodes n' did not have their values V_{i+1} computed already when node n is removed from the queue for the l^{th} time. However, at the end of iteration $i + 1$, the value of node n, $V_{i+1}(n)$, will be correctly computed using $V_{i+1}(n')$.

Figure 3 depicts a case where a node is pushed several times on the queue. The figure depicts the policy π_i at iteration $i + 1$. Node n_2 was first generated by performing the action E in node n_1, but later on, after performing action S in node n_3, n_2 was also one of the successors of n_3. Assume that at iteration $i+1$, the node (n_2, W) was chosen for expansion. Then after computing $V_{i+1}(n_2)$, the fringe node n_2 pushes on the queue its marked parents n_1 and n_3. Node

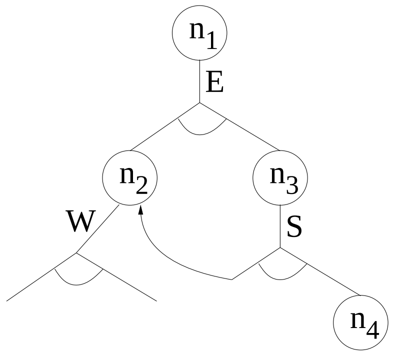

Fig. 3. A policy in an AND/OR graph, which will cause node n_1 to appear twice on the queue. The fringe node is n_2, and the queue will hold nodes n_2, n_1, n_3, n_1.

n_3 will also push node n_1 on the queue. Therefore at iteration $i+1$ the queue will hold nodes n_2, n_1, n_3, n_1, in this order. First time node n_1 is removed from the queue, it will compute $Q_{i+1}(n_1, E) = C(n1, E) + w(n_1, E, n_2) \cdot V_{i+1}(n_2) + w(n_1, E, n_3) \cdot V_i(n_3)$, using the value function from iteration i for node n_3, because the value of n_3 was not yet updated at iteration $i+1$. The second time n_1 is removed from the queue, node n_3 was already removed from the queue and it had its value $V_{i+1}(n_3)$ computed, so $Q_{i+1}(n_1, E) = C(n1, E) + w(n_1, E, n_2) \cdot V_{i+1}(n_2) + w(n_1, E, n_3) \cdot V_{i+1}(n_3)$.

4 Proofs

This section first proves that AO* with an admissible heuristic converges to the optimal policy π^* of the AND/OR graph. Next we show that AO* with a consistent heuristic also converges to π^* but performs more efficient updates of the value function. In order to prove this result, we first prove an invariant (at the end of each iteration i of AO* with efficient updates, every node n in the graph has the correct value function $V_i(n)$), and we prove that the value function of a node $V_i(n)$ increases with future iterations.

Theorem 4.1 *AO* with an admissible heuristic h converges to the optimal policy π^*, and $V(n) = V^*(n)$, for every OR node n reached by the optimal policy π^* from the root.*

In the worst case, AO* does exhaustive search in the implicit graph, which is a finite DAG, until its policy π becomes a complete policy. Because the leaves of π are terminal OR nodes, which are solved, it results that all OR nodes n reachable by π are also solved (from the definition of solved), so the root is also solved. Therefore we exit the while loop in Table 1, and the policy π returned by AO* upon convergence is complete.

In the following, by $V(n)$ we denote the values, when the algorithm terminates, of the OR nodes n reachable by π from the root of the graph.

Proof by induction:

<u>Base case</u>

The leaf OR nodes n of π are terminal, so $V(n) = V^*(n) = 0$.

<u>Induction hypothesis</u>

Let n be any internal (non-terminal) OR node of π, such that all its successor OR nodes n' through π have $V(n') = V^*(n')$. Then $V(n) = V^*(n)$.

Let $\pi(n) = a$, therefore $V(n) = Q(n, a)$. Because n is an internal node of π, the AND node (n, a) was expanded, and by definition $Q(n, a) = C(n, a) + \sum_{n'} w(n, a, n') \times V(n')$. Because n' are the successor OR nodes of n through π, we can apply the induction hypothesis for n', so $V(n') = V^*(n')$, and we have $Q(n, a) = Q^*(n, a)$.

Since $V(n) = Q(n, a)$, and $V(n) = \min_{a' \in A(n)} Q(n, a')$, it results that $Q(n, a) \leq Q(n, a'), \forall a' \in A(n)$. Because $Q(n, a') \leq Q^*(n, a'), \forall a' \in A(n)$, according to Theorem 3.1, it follows that $Q^*(n, a) = Q(n, a) \leq Q(n, a') \leq Q^*(n, a'), \forall a' \in A(n)$. Therefore $Q^*(n, a) \leq Q^*(n, a'), \forall a' \in A(n)$, so $V^*(n) = \min_{a' \in A(n)} Q^*(n, a') = Q^*(n, a)$. But $V(n) = Q(n, a) = Q^*(n, a) = V^*(n)$, therefore $V(n) = V^*(n)$.

We proved that for all nodes n reached by the complete policy π when AO* terminates, $V(n) = V^*(n)$. Therefore this policy is an optimal policy π^*. **Q.E.D.**

Let i be the current iteration of the AO* algorithm. G_i is the explicit graph at the end of iteration i. V_i and π_i are the value function and policy at the end of iteration i. Note that $A(n)$ does not change with the iteration. The fringe node is the OR node whose policy π_i is expanded at iteration $i+1$. We define the Q value function for an expanded AND node (n, a) to be $Q_i(n, a) = C(n, a) + \sum_{n'} w(n, a, n') \cdot V_i(n')$, that is, it is based on one-step lookahead and it uses V_i for its successors. Theorem 4.3 will prove that at the end of AO* iteration i, every OR node n in the graph G_i has the correct $V_i(n)$ and $\pi_i(n)$. However, the Q values stored in the graph may not be the most recent ones; but since the V values are the correct ones, when the Q values are needed, a one-step lookahead will compute their most up-to-date values. That is why we define $Q_i(n, a)$ to be the updated Q value based on V_i, and not the actual Q value stored in the AND node.

The following theorem establishes that the AO* algorithm, provided with a consistent heuristic, converges to the optimal policy even if we only update the values of the marked ancestors of the fringe node (we call this *selective updates*), instead of updating the values of all ancestors of the fringe node.

Theorem 4.2 *AO^* with a consistent heuristic and with selective updates of the value function converges to the optimal policy π^*.*

In order to prove this theorem, we will first prove the following invariant:

Theorem 4.3 Invariant: *At the end of every iteration i of AO^* with a consistent heuristic and selective updates, every OR node n in the graph G_i has the correct $V_i(n)$ and $\pi_i(n)$.*

Proof of Theorem 4.2: After we prove the invariant, we only need to replace i with the iteration $final$ at which AO* with selective updates converges, and because the policy π_{final} is complete, we can employ a proof similar to Theorem 4.1's to show that $V_{final} = V^*$ and $\pi_{final} = \pi^*$, which proves Theorem 4.2.

Proof of Theorem 4.3 (the Invariant), by induction over iteration i:

Base case (before the while loop in Table 1):

If $i = 0$, the graph G_0 consists only of the *root* node. Let us assume that the root node is not terminal. Then all its actions $a \in A(root)$ are unexpanded, therefore $Q_0(root, a) = C(root, a) + \sum_{n'} w(root, a, n') \cdot h(n')$. So $V_0(root) = \min_{a \in A(root)} Q_0(root, a)$ and $\pi_0(root) = \text{argmin}_{a \in A(root)} Q_0(root, a)$ are correct.

Induction hypothesis:
If G_i (at the end of iteration i of AO* with a consistent heuristic and selective updates) has the correct V_i and π_i, then graph G_{i+1} has the correct V_{i+1} and π_{i+1}.

To prove this induction hypothesis, we need to consider several cases for an arbitrary node n in the graph G_{i+1}.

Case (a): Node n belongs to G_{i+1} but not to G_i.
That means that node n was generated at iteration $i+1$, by expanding the policy π_i of the fringe node. If n is a terminal node, $V_{i+1}(n) = 0$ and $\pi_{i+1}(n)$ is not defined, which is correct. Otherwise, since n was not previously generated, all its actions $a \in A(n)$ are unexpanded, so $Q_{i+1}(n, a) = C(n, a) + \sum_{n'} w(n, a, n') \cdot h(n')$. So $V_{i+1}(n) = \min_{a \in A(n)} Q_{i+1}(n, a)$ and $\pi_{i+1}(n) = \text{argmin}_{a \in A(n)} Q_{i+1}(n, a)$ are correct. Note that $V_i(n)$ and $\pi_i(n)$ are not defined.

Case (b): Node n is a successor of the fringe node, and n belongs to G_i.
That means that n was generated before iteration $i + 1$. Because the graph is a DAG, n is not an ancestor of the fringe, so its value and policy do not change at iteration $i + 1$, $V_{i+1}(n) = V_i(n)$, and $\pi_{i+1}(n) = \pi_i(n)$.

Case (c): Node n is the fringe node whose policy $\pi_i(n) = a$ was expanded at iteration $i+1$.
Then each successor n' through a is either of type (a) or (b), that is, n' is either terminal, newly generated, or generated before (in graph G_i), and it has the correct value $V_{i+1}(n')$. It follows that $Q_{i+1}(n,a)$ is correctly updated based on $V_{i+1}(n')$. Any other action a' in the fringe node n is either: (1) unexpanded, so its $Q_{i+1} = Q_i$ value is computed based on h of its successors, and therefore does not need to be updated, or (2) was expanded before, but its value does not change $Q_{i+1}(n,a') = Q_i(n,a')$, because its successors n' do not change their value function $V_{i+1}(n') = V_i(n')$, as shown in case (b). Therefore the value $V_{i+1}(n)$ and policy $\pi_{i+1}(n)$ of the fringe node are updated correctly.

Note that in case (2) we need to update $Q_{i+1}(n,a') = C(n,a') + \sum_{n'} w(n,a,n') \cdot V_{i+1}(n')$, because it is possible that $Q_i(n,a')$ was not computed at iteration i, so we cannot use the stored Q value of (n,a').

We show next that the value of node n is nondecreasing. Because (n,a) was unexpanded at the end of iteration i, $Q_i(n,a) = C(n,a) + \sum_{n'} w(n,a,n') \cdot h(n')$. After expansion at iteration $i+1$, $Q_{i+1}(n,a) = C(n,a) + \sum_{n'} w(n,a,n') \cdot V_{i+1}(n')$. Since h is consistent, we can apply Theorem 3.2 for iteration $i+1$ and nodes n', so $h(n') \le V_{i+1}(n')$, therefore $Q_i(n,a) \le Q_{i+1}(n,a)$. Since $Q_i(n,a') = Q_{i+1}(n,a')$ for any other action $a' \ne a$, it follows that $V_i(n) = \min_{a \in A(n)} Q_i(n,a) \le \min_{a \in A(n)} Q_{i+1}(n,a) = V_{i+1}(n)$, therefore $V_i(n) \le V_{i+1}(n)$.

The fringe node becomes solved at iteration $i+1$ if all its successor OR nodes through $\pi_{i+1}(n)$ are solved (they are either terminal, or were marked solved in the graph G_i). A solved node will always stay solved, and its value is in fact the optimal value V^*.

The fringe node n was the first node to be pushed on the queue in iteration $i+1$. If either $V_i(n) < V_{i+1}(n)$ or n becomes solved, the fringe node triggers a change in the graph, and this needs to be propagated to its ancestors (we will actually prove that only the marked ancestors of n will be affected).
End of Case (c).

In the remaining cases, node n belongs to both graphs G_i and G_{i+1}.

Recall that, by definition, OR node n is an *ancestor* of fringe node if there is a directed path of AND nodes that can be followed from n to the fringe node (base case: any parent of fringe is an ancestor of fringe; any OR node who has at least one successor OR node which is an ancestor of fringe is an ancestor of fringe).

There are three remaining cases for node n, at the end of iteration i:

Case (d): Node n is *not an ancestor* of the fringe node.
By definition, there is no directed path of AND nodes in G_i that can be followed from n to the fringe node (base case: n is not a parent of fringe; none of the successors of n is an ancestor of fringe). Case (b), where n is a successor of the fringe node, is a special case of (d), that is why we called the path of AND nodes "directed".

Case (e): Node n is a *marked ancestor* of the fringe node.
Recursive definition: any parent n of the fringe node that reaches the fringe node through $\pi_i(n)$ is a marked ancestor of fringe. Any other ancestor of the fringe node in G_i is marked if it has at least one successor through $\pi_i(n)$ that is a marked ancestor of fringe. Note that not all marked ancestors of fringe may be reached by π_i from the root.

Case (f): Node n is an *unmarked ancestor* of the fringe node.
Recursive definition: any parent n of the fringe node that reaches the fringe node through an action $a \neq \pi_i(n)$ is an unmarked ancestor of fringe. Any other ancestor of the fringe node in G_i is unmarked if at least one of its successors n' is an ancestor of fringe, with the restriction that if n' is reached through $\pi_i(n)$, then n' must be unmarked.

Note: an ancestor of fringe may change its type between (e) and (f) depending on the policy, which in turn depends on the AO* iteration i, that is why the iteration i appears in the above definitions. Also, a non-ancestor can become an ancestor in future iterations (for example, in Figure 3, before expanding (n_3, S), node n_3 is not an ancestor of n_2).

A change in the fringe node (either an increase in its value function, or it becoming solved) may potentially trigger a change in all of its ancestors. The following theorem proves that only marked ancestors of fringe need to be considered for updates, while unmarked ancestors, and nodes that are not ancestors of fringe, do not need to update their value function.

Theorem 4.4 *Let n be any node in the graph G_{i+1}. If heuristic h is consistent, and (1) if n is not an ancestor of fringe, then $V_i(n) = V_{i+1}(n)$ (no update is necessary); (2) if n is the fringe node, or if n is a marked ancestor of fringe, then $V_i(n) \leq V_{i+1}(n)$ (an update is necessary); (3) if n is an unmarked ancestor of fringe, then $V_i(n) = V_{i+1}(n)$ (no update is necessary).*

We explain the main idea using a scenario of a trainer having to select the best performer from a team. The players' performance can stay the same or can worsen from day to day. If the performance of the best player today, called B, is the same tomorrow, then tomorrow the trainer does not need to consider the other players, because their performance will not be better than B's. But if the performance of B gets worse tomorrow, then the trainer needs to compare his performance with the other players'. The analogy is: the team is the node n, the players are the actions $A(n)$, "today"is iteration i, "tomorrow" is iteration "i+1", B is $\pi_i(n) = a$, performance of B today is $Q_i(n,a) = V_i(n)$, performance of B tomorrow is $Q_{i+1}(n,a)$. Let a' be any another action from $A(n)$ (another player). The performance of a is best today, so $Q_i(n,a) \leq Q_i(n,a')$. The performance of any player can stay the same or can get worse (the costs increase) with each day (iteration), so $Q_i(n,a) \leq Q_{i+1}(n,a)$, and $Q_i(n,a') \leq Q_{i+1}(n,a')$. If $Q_i(n,a) = Q_{i+i}(n,a)$ then $Q_{i+1}(n,a) \leq Q_{i+1}(n,a')$, so $V_{i+1}(n) = V_i(n) = Q_{i+1}(n,a)$ and $\pi_{i+1}(n) = a$. If $Q_i(n,a) < Q_{i+1}(n,a)$ then it is possible that $V_{i+1}(n) < Q_{i+1}(n,a)$ and $\pi_{i+1}(n) \neq a$.

Proof by induction:

The induction is done after the index of the node n in the graph G'_{i+1}. This graph has all the nodes from G_{i+1}, has fringe as root, and all its arcs are the reversed of the arcs in G_{i+1}. Because the graph G'_{i+1} is a DAG, we can attach an *index* to each node n such that $index(fringe) = 1$ and $index(n) > index(n'), \forall n'$, where n' is a successor of n in graph G_{i+1}.

Base case:

Node n is the fringe node, with $index(n) = 1$. Case (c) proved that $V_i(n) \leq V_{i+1}(n)$ so an update in the value function of fringe is necessary. We also need to update all Q values of expanded AND nodes.

Induction hypothesis:
Let n be any node (different from fringe) in the graph G_{i+1}, such that all its successors n' have the correct $V_{i+1}(n')$, with $V_{i+1}(n') = V_i(n')$ if n' is not an ancestor of fringe or if it is an unmarked ancestor of fringe, and $V_i(n') \leq V_{i+1}(n')$ if n' is a marked ancestor of fringe. Then node n will have the correct $V_{i+1}(n)$, needing no updates if n is not an ancestor of fringe or if it is an unmarked ancestor of fringe, and needing an update if n is a marked ancestor of fringe.

Node n is of type (d), not an ancestor of fringe.

Since any successor n' of n is not an ancestor of fringe, we can apply the induction hypothesis for n' so $V_{i+1}(n') = V_i(n')$, therefore $Q_{i+1}(n,a) = Q_i(n,a), \forall a \in A(n)$ and $V_{i+1}(n) = V_i(n)$. We do not need to update $V(n)$, nor $Q(n,a)$.

Intuitively, since there is no directed path of AND nodes that can be followed from n to the fringe node, any change in the fringe node will not influence node n. Therefore $V_i(n) = V_{i+1}(n)$ and $\pi_i(n) = \pi_{i+1}(n)$, and no updates are necessary for node n in iteration $i+1$ (so n will not be pushed on the queue).
End of case (d).

Node n is of type (e), a marked ancestor of fringe.

By definition, at least one of the successors of n through $\pi_i(n)$ is a marked ancestor of fringe, or is the fringe node. Denote this successor by n'_m. The induction hypothesis guarantees that $V_i(n'_m) \leq V_{i+1}(n'_m)$. The other successors n' of n (even through $\pi_i(n)$) can be of type (d), (e), or (f). We need to perform a full Bellman update for n, that is, we need to compute all $Q_{i+1}(n, a), \forall a \in A(n)$, because $V_i(n') \leq V_{i+1}(n')$ if n' is of type (e), or if it is the fringe node. Recomputing all $Q_{i+1}(n, a)$ takes the same time as it would take to check the type of each successor node n'. Applying the induction hypothesis for n', we obtain $Q_i(n, a) \leq Q_{i+1}(n, a)$ for all AND nodes (n, a) who have at least one successor of type (e) or who have the fringe node as successor, and we know there is at least one such AND node, $(n, \pi_i(n))$, and $Q_i(n, a) = Q_{i+1}(n, a)$ for all AND nodes (n, a) whose successors are of type (d) or (f). Therefore $V_i(n) \leq V_{i+1}(n)$.

Because all successors n' of n have the correct $V_{i+1}(n')$ values, node n will also have the correct $V_{i+1}(n)$ value.

Node n was pushed on the queue by each of its successors n' of type (e) (or by the fringe node) whose value strictly increased, $V_i(n) < V_{i+1}(n)$, or which has become solved. Node n will push its marked parents, that were actually marked in iterations previous to $i + 1$, on the queue if $V_i(n) < V_{i+1}(n)$ or if it became solved (node n becomes solved at iteration $i + 1$ if all its successor OR nodes through $\pi_{i+1}(n)$ are solved). In turn, if there are changes in these parents, they will push all their marked parents on the queue; the next case (f) completes the proof that the queue only contains marked ancestors of fringe.

Recall that a node n can be pushed several times on the queue, if the node has more than one successor n' through $\pi_i(n)$ that was a marked ancestor of fringe. Before the last update of node n (when it is removed for the last time from the queue), it is possible that not all the successors n' have computed their value $V_{i+1}(n')$ (instead, they still have the old value $V_i(n')$). However, at the time of the last update of n, all successors n' have the correct $V_{i+1}(n')$ values, and node n will also update its value to the correct $V_{i+1}(n)$. The above proof for case (e) is done considering that n is removed for the last time from the queue. End of case (e).

Node n is of type (f), an unmarked ancestor of fringe.

All successors n' of n through $\pi_i(n)$ are of type (d) or (f), and none is the fringe node or a marked ancestor of fringe. According to the induction hypothesis for nodes n' of type (d) or (f), $V_{i+1}(n') = V_i(n')$, so $Q_{i+1}(n, \pi_i(n)) = Q_i(n, \pi_i(n))$. The successors n'' through action $a \neq \pi_i(n)$ can be of type (d), (e), (f), or fringe, so if we were to update $Q(n, a)$, its value may increase, $Q_i(n, a) \leq Q_{i+1}(n, a)$ for AND nodes (n, a) that have successors of type (e) or fringe. However, we do not need to update $Q(n,a)$ for $a \neq \pi_i(n)$, because $Q_{i+1}(n,\pi_i(n)) = Q_i(n,\pi_i(n)) \leq Q_i(n,a) \leq Q_{i+1}(n,a), \forall a \in A(n)$, so $Q_{i+1}(n,\pi_i(n)) \leq Q_{i+1}(n,a), \forall a \in A(n)$, therefore $V_{i+1}(n) = V_i(n)$ and $\pi_{i+1}(n) = \pi_i(n)$. In conclusion, no updates are necessary for node n which is an unmarked ancestor of fringe, so n will not be pushed on the queue. End of case (f).

Q.E.D.

This completes the proof of Theorem 4.4, which also completes the proof of the Invariant(Theorem 4.3). Accordingly, we can view AO* as performing updates only in the subgraph of G'_{i+1} that has the fringe node as root and only contains nodes that are marked ancestors of the fringe.

Corollary 4.5 *If heuristic h is consistent, then for any node n in the graph G_i, subsequent iterations of AO* with selective updates increase its value: $V_i(n) \leq V_{i+1}(n)$.*

Proof: It follows from the proof of the Invariant (Theorem 4.3). We could extend the corollary to apply to all nodes n, even those that were not generated in G_i, by defining $V_i(n) = h(n)$.

Corollary 4.6 *If heuristic h is consistent, then for any AND node (n,a) in the graph G_i, subsequent iterations of AO* with selective updates increase its value: $Q_i(n,a) \leq Q_{i+1}(n,a)$.*

Proof: It follows from the proof of the Invariant (Theorem 4.3).

If the heuristic h is admissible but not consistent, it is possible that the value of a node n decreases in future iterations, $V_{i+1}(n) < V_i(n)$. If this happens for the fringe node (or any of its ancestors), then all its ancestors must be pushed on the queue, and no selective updates in AO* are possible. The reason is that a decrease in value function in a node causes a decrease in the Q value function in an ancestor, even for an action that was not the best policy, but that may now become the best policy.

5 Review of AO* Literature

The AO* algorithm has been studied extensively both in the Artificial Intelligence and the Operations Research communities. Horowitz and Sahni [5, pages 530-532] proved the NP completeness of the AND/OR decision problem (does graph G have a solution of cost at most k, for given k), by reduction from CNF-satisfiability.

Our definitions of AND and OR nodes are similar to those of Martelli and Montanari [1], Chakrabarti et al. [6], Pattipati and Alexandridis [7], Qi [8] and Hansen [9]. An OR node specifies the choice of an action. It is called an OR node because its solution involves the selection of only one of its successor AND nodes. An AND node specifies the outcomes of an action. It is called an AND node because in order to solve it, all its successor OR nodes must be solved. These definitions are the reverse of Nilsson's [2] in which the type of a node is determined by the relation to its parent.

There are several implementations of AO*: two by Martelli and Montanari [1,10], one by Nilsson [2], and one by Mahanti and Bagchi [11]. The first three implementations are practically identical. Martelli and Montanari are the first to recognize that dynamic programming techniques that discover common subproblems can be applied to search AND/OR graphs and to compute the optimal solution. Martelli and Montanari [1] show that the AO* algorithm with an admissible heuristic converges to an optimal policy π^*.

Our implementation follows the framework of Nilsson's. Our analysis is more complex than his, because he does not explicitly differentiate between AND nodes and OR nodes when describing the AO* algorithm for graphs, though he makes the distinction between the two nodes when discussing AND/OR trees. He calls AND/OR graphs hypergraphs, and their hyperarcs/hyperlinks connectors, so instead of arcs/links connecting pairs of nodes in ordinary graphs, connectors connect a parent node with a set of successor nodes. The complete solution (no longer a path, but a hyperpath), is represented by an AND/OR subgraph, called a solution graph (with our notation, this is a policy). These connectors require a new algorithm, AO*, for the AND/OR graphs, instead of the A* algorithm for ordinary graphs.

Nilsson does not refer to the nodes of an AND/OR graph as being AND nodes or OR nodes, because in general a node can be seen as both an OR node and an AND node. Pearl [12] notes as well that an AND link and an OR link can point to the same node. Nevertheless, in this paper we separate the two types of nodes, so a node is either a pure AND node or a pure OR node.

AND/OR graph search algorithms have been studied by many authors. At first, the algorithms worked on an *implicit* graph (the entire graph that can be generated), which was assumed to be acyclic [10,2,11,6]. An *explicit* graph is the part of the graph generated during the search process. Graphs with cycles were usually solved by unfolding the cycles. For a recent review of AO* algorithms for searching both explicit and implicit AND/OR graphs with cycles see [13].

Branch-and-bound algorithms use lower and upper bounds to prune non-optimal branches, without generating and evaluating the entire AND/OR graph. Kumar and Kanal [14] explain the relationship between branch-and-bound algorithms from Operations Research and heuristic search algorithms from Artificial Intelligence (including alpha-beta [15], AO*, B* [16], and SSS* [17]). Other relevant articles, which outline AO* as a branch-and-bound procedure are [18–20]. They show that AO* is a special case of a general branch-and-bound formulation.

5.2 Theoretical Results on AO*

5.2.1 Memory-bounded AO*

AO* may require memory exponential in the size of the optimal policy. Chakrabarti et al. [21] propose running AO* in restricted memory by pruning unmarked nodes (that are not part of some policy π_i) when the available memory is reached. This method still computes the optimal value function, trading-off space for time (if the pruned nodes are needed again, they must be generated again).

5.2.2 Inadmissible Heuristics

If the heuristic h is inadmissible, but within ϵ of the optimal value function V^*, then it is straightforward to compute a bound on the value of the suboptimal policy learned with h. Indeed, if $h(n) - V^*(n) \leq \epsilon, \forall n$, then the maximal error of the policy π computed by AO* with heuristic h is ϵ, $V^\pi - V^* \leq \epsilon$ (see [6] and [8], page 31).

Chakrabarti et al. [6] showed that the optimal policy can be computed with an inadmissible heuristic if if its weight is shrunk. If the heuristic function can be decomposed into $f = g + h$, where g is the cost incurred so far, and h is the heuristic estimating the remaining cost, then AO* with the weighted heuristic $(1 - w) \cdot g + w \cdot h$, where $0 \leq w \leq 1$, compute an optimal policy π^* even for an overestimating (inadmissible) heuristic h. The weight w is such that $w < \frac{1}{1+\epsilon}$, where ϵ is the maximum distance between the inadmissible heuristic h and V^*.

5.2.3 Influence of Heuristic on Nodes Expanded

Chakrabarti et al. [6] show that a more accurate admissible heuristic in AO* has a smaller worst-case set of nodes expanded. That is, if heuristic h_2 is closer to the optimal value function than h_1, $h_1 \leq h_2 \leq V^*$, then the largest set of nodes expanded by AO* with the h_2 heuristic is a subset of the largest set of nodes expanded by AO* with the h_1 heuristic.

5.3 MDPs

A diagnostic policy specifies what test to perform next, based on the results of previous tests, and when to stop and make a diagnosis. An optimal diagnostic policy minimizes the expected total cost of tests and incorrect diagnoses. We formulated the problem of learning diagnostic policies as an MDP, and showed how to apply the AO* algorithm to learn the optimal policy [22,23,4]. The MDP states correspond to all possible combinations of test results. Unlike other work on diagnosis which assumes a Bayesian network from which the MDP probabilities are inferred, we learn the MDP probabilities from data, as needed by the AO* search.

The test sequencing problem is a simpler version of learning diagnostic policies. The goal is to deterministically identify faulty states of an electronic system while minimizing expected test costs. Since the system states can be identified unambiguously, there are no misdiagnosis costs. Pattipati and Alexandridis [7] observe that the test sequencing problem is an MDP whose solution is an optimal AND/OR decision tree. An MDP state is a set of system states. To compute the optimal solution to the test sequencing problem, Pattipati and Alexandridis employ AO* with two admissible heuristics, one derived from Huffman coding and the other being based on the entropy of system states. These two heuristics require equal test costs in order to be admissible.

Hansen's LAO* algorithm [24] is a generalization of AO* that solves MDPs with cycles by using a dynamic programming method (either value iteration or policy iteration) for the bottom-up update of the value function and policy. This is necessary because in the general case of an MDP with cycles, we can not perform a single sweep of value iteration through the state space from the fringe node to the root node to update the value function. Bonet and Geffner [25,26] follow up with heuristic search algorithms for solving MDPs with cycles, which converge faster than value iteration, RTDP or Hansen's LAO*. RTDP [27] is a Real Time Dynamic Programming algorithm that bypasses full dynamic programming updates in MDPs by only updating the values of states reached from an initial state during repeated trials of executing a greedy policy; a heuristic is used to initialize the value function. If the heuristic is admissible, then RTDP converges in the limit to the optimal policy. RTDP extends Korf's Learning Real Time A* (LRTA*) algorithm [28] to asynchronous dynamic programming with stochastic actions. LRTA* can be seen as a real-time version of A*, and RTDP is the real-time version of AO* and LAO*. In terms of how they represent solutions, A* outputs a simple path (a sequence of actions), AO* outputs a directed acyclic graph, and LAO* outputs a cyclic graph (a finite-state controller).

5.4 POMDPs

Partially Observable MDPs (POMDPs) are MDPs in which the state of the world is not known with certainty. Instead, observations reveal information about the state of the world. The states of the POMDP are called *belief or information states*.

Heuristic search methods (either branch-and-bound or AO*) have been applied to approximately solve infinite-horizon POMDPs from a single initial belief state. Satia and Lave [29] proposed a branch-and-bound algorithm for finding optimal and ϵ-optimal POMDP policies; Larsen and Dyer [30] improve upon this work. Washington [31] used the value function of the underlying MDP to define lower and upper bounds in AO*. Hansen [9] developed a heuristic search algorithm that combines AO* with policy iteration for approximately solving infinite-horizon POMDPs (from a given start state) by searching in the policy space of finite-state controllers. For a recent review of approximation methods for solving POMDPs, which also includes a systematic presentation of lower and upper bounds for POMDPs, see Hauskrecht [32].

6 Conclusions

This paper presents a mathematical framework to solve acyclic AND/OR graphs using the AO* algorithm. Several notations, such as value functions and policy, are inspired from the MDP framework. The paper details the implementation of AO* using these notations, proves that AO* with an admissible heuristic converges to the optimal solution graph of the AND/OR graph, and fills a gap in the AO* literature by proving that, using a consistent heuristic in AO*, the value function of a node increases monotonically and more efficient updates of the value function can be performed.

References

[1] A. Martelli, U. Montanari, Additive AND/OR graphs, in: Proceedings of the Third International Joint Conference on Artificial Intelligence, 1973, pp. 1–11.

[2] N. Nilsson, Principles of Artificial Intelligence, Tioga Publishing Co., Palo Alto, CA, 1980.

[3] J. R. Slagle, A heuristic program that solves symbolic integration problems in freshman calculus, Journal of the Association for Computing Machinery 10(4) (1963) 507–520.

[4] V. Bayer-Zubek, Learning diagnostic policies from examples by systematic search, in: Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence (in press), Banff, Canada, 2004.

[5] E. Horowitz, S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Rockville, MD, 1978.

[6] P. Chakrabarti, S. Ghose, S. DeSarkar, Admissibility of AO* when heuristics overestimate, Artificial Intelligence 34 (1988) 97–113.

[7] K. R. Pattipati, M. G. Alexandridis, Application of heuristic search and information theory to sequential fault diagnosis, IEEE Transactions on Systems, Man and Cybernetics 20(4) (1990) 872–887.

[8] R. Qi, Decision graphs: Algorithms and applications to influence diagram evaluation and high-level path planning under uncertainty, Ph.D. thesis, University of British Columbia (1994).

[9] E. Hansen, Solving POMDPs by searching in policy space, in: Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann, San Francisco, 1998, pp. 211–219.

[10] A. Martelli, U. Montanari, Optimizing decision trees through heuristically guided search, Communications of the ACM 21(12) (1978) 1025–1039.

[11] A. Mahanti, A. Bagchi, AND/OR graph heuristic search methods, Journal of the ACM 32(1) (1985) 28–51.

[12] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Publishing Co., Massachusetts, 1984.

[13] P. Jimenez, C. Torras, An efficient algorithm for searching implicit AND/OR graphs with cycles, Artificial Intelligence 124 (2000) 1–30.

[14] V. Kumar, L. N. Kanal, A general branch and bound formulation for understanding and synthesizing AND/OR tree search procedures, Artificial Intelligence 21 (1983) 179–198.

[15] D. E. Knuth, R. W. Moore, An analysis of alpha-beta pruning, Artificial Intelligence 6(4) (1975) 293–326.

[16] H. Berliner, The B* tree search algorithm: a best-first proof procedure, Artificial Intelligence 12 (1979) 23–40.

[17] G. C. Stockman, A minimax algorithm better than alpha-beta?, Artificial Intelligence 12 (1979) 179–196.

[18] D. S. Nau, V. Kumar, L. N. Kanal, General branch and bound, and its relation to A* and AO*, Artificial Intelligence 23(1) (1984) 29–58.

[19] V. Kumar, L. N. Kanal, The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound, in: L. N. Kanal, V. Kumar (Eds.), Search in Artificial Intelligence, Springer-Verlag, Berlin, 1988, pp. 1–27.

[20] V. Kumar, L. N. Kanal, A general branch-and-bound formulation for AND/OR graph and game tree search, in: L. N. Kanal, V. Kumar (Eds.), Search in Artificial Intelligence, Springer-Verlag, Berlin, 1988, pp. 91–130.

[21] P. Chakrabarti, S. Ghose, A. Acharya, S. DeSarkar, Heuristic search in restricted memory, Artificial Intelligence 41 (1989) 197–221.

[22] V. Bayer-Zubek, T. Dietterich, Pruning improves heuristic search for cost-sensitive learning, in: Proceedings of the Nineteenth International Conference of Machine Learning, Morgan Kaufmann, Sydney, Australia, 2002, pp. 27–35.

[23] V. Bayer-Zubek, Learning cost-sensitive diagnostic policies from data, Ph.D. thesis, Department of Computer Science, Oregon State University, Corvallis, http://eecs.oregonstate.edu/library/?call=2003-13 (2003).

[24] E. Hansen, S. Zilberstein, A heuristic search algorithm that finds solutions with loops, Artificial Intelligence 129 (1–2) (2001) 35–62.

[25] B. Bonet, H. Geffner, Labeled RTDP: Improving the convergence of real-time dynamic programming, in: Proceedings of ICAPS-03, 2003.

[26] B. Bonet, H. Geffner, Faster heuristic search algorithms for planning with uncertainty and full feedback, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Francisco, 2003.

[27] A. Barto, S. Bradtke, S. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72(1) (1995) 81–138.

[28] R. Korf, Real-time heuristic search, Artificial Intelligence 42 (1990) 189–211.

[29] J. K. Satia, R. E. Lave, Markovian decision processes with probabilistic observation of states, Management Science 20 (1973) 1–13.

[30] J. B. Larsen, J. S. Dyer, Using extensive form analysis to solve partially observable Markov decision problems, Tech. rep., 90/91-3-1, University of Texas at Austin (1990).

[31] R. Washington, BI-POMDP: Bounded, incremental partially-observable Markov-model planning, in: Proceedings of the Fourth European Conference on Planning, 1997.

[32] M. Hauskrecht, Value-function approximations for partially observable Markov decision processes, Journal of Artificial Intelligence Research 12 (2000) 33–94.