AO*

CHAPTER 3

SEARCH STRATEGIES FOR DECOMPOSABLE PRODUCTION SYSTEMS

In chapter 1, we introduced decomposable production systems and structures called AND/OR trees, for controlling their operation. In this chapter we describe some heuristic strategies for searching AND/OR trees and graphs. We also describe some search techniques for graphs used in game-playing systems.

3.1. SEARCHING AND/OR GRAPHS

Recall that the *AND* or the *OR* label given to a node in an AND/OR tree depends upon that node's relation to its parent. In one case, a parent node labeled by a compound database has a set of AND successor nodes, each labeling one of the component databases. In the other case, a parent node labeled by a component database has a set of OR successor nodes, each labeling the database resulting from the application of alternative rules to the component database.

We are generally concerned with AND/OR graphs rather than with the special case of trees, because different sequences of rule applications may generate identical databases. For example, a node could be labeled by a component database resulting both from having split a compound one and from having applied a rule to another one. In this case, it would be called an OR node with respect to one parent and an AND node with respect to the other parent. For this reason, we do not generally refer to the nodes of an AND/OR graph as being AND nodes or OR nodes;

instead, we introduce some more general notation, appropriate for graphs. We continue to call these structures AND/OR graphs, however, and use the terms AND nodes and OR nodes when discussing AND/OR trees.

We define AND/OR graphs here as *hypergraphs*. Instead of arcs connecting *pairs* of nodes, there are *hyperarcs* connecting a parent node with a *set* of successor nodes. These hyperarcs are called *connectors*. Each *k-connector* is directed from a *parent* node to a set of *k successor* nodes. (If all of the connectors are 1-connectors, we have the special case of an ordinary graph.)

In Figure 3.1, we show an example of an AND/OR graph. Note that node n_0 has a 1-connector directed to successor n_1 and a 2-connector directed to the set of successors $\{n_4, n_5\}$. For $k > 1$, k-connectors are denoted in our illustrations by a curved line joining the arcs from parent to elements of the successor set. (Using our earlier terminology, we could have regarded nodes n_4 and n_5 as a set of AND nodes, and we could have regarded node n_1 as an OR node, relative to their common parent n_0; but note that node n_8, for example, belongs to a set of AND nodes relative to its parent n_5 but is an OR node relative to its parent n_4.)

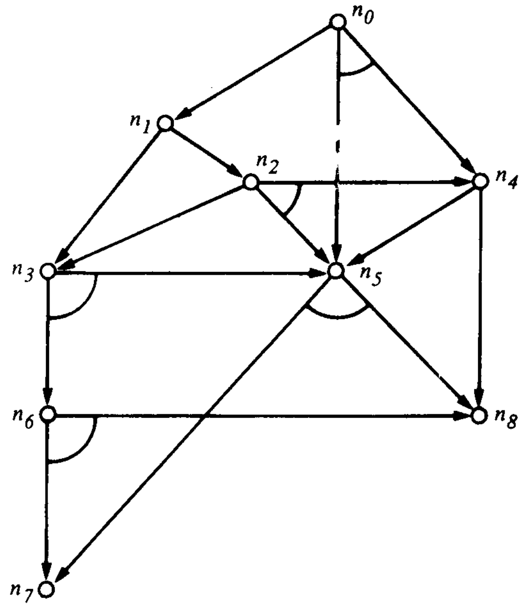

Fig. 3.1 An AND/OR graph.

In an AND/OR tree, each node has at most one parent. In trees and graphs we call a node without any parent a *root* node. In graphs, we call a node having no successors a *leaf node* (a *tip* node for trees).

A decomposable production system defines an implicit AND/OR graph. The initial database corresponds to a distinguished node in the graph called the *start node*. The start node has an outgoing connector to a set of successor nodes corresponding to the components of the initial database (if it can be decomposed). Each production rule corresponds to a connector in the implicit graph. The nodes to which such a connector is directed correspond to component databases resulting after rule application and decomposition into components. There is a set of *terminal* nodes in the implicit graph corresponding to databases satisfying the termination condition of the production system. The task of the production system can be regarded as finding a *solution graph* from the start node to the terminal nodes.

Roughly speaking, a solution graph from node n to node set N of an AND/OR graph is analogous to a path in an ordinary graph. It can be obtained by starting with node n and selecting exactly one outgoing connector. From each successor node to which this connector is directed, we continue to select one outgoing connector, and so on, until eventually every successor thus produced is an element of the set N. In Figure 3.2, we show two different solution graphs from node n_0 to $\{n_7, n_8\}$ in the graph of Figure 3.1.

We can give a precise recursive definition of a solution graph. The definition assumes that our AND/OR graphs contain no cycles, that is, it assumes that there is no node in the graph having a successor that is also its ancestor. The nodes thus form a partial order which guarantees termination of the recursive procedures we use. We henceforth make this assumption of acyclicity.

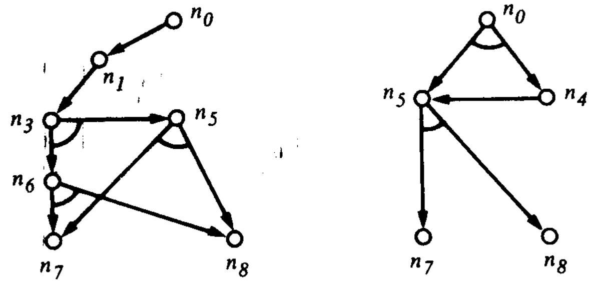

Fig. 3.2 Two solution graphs.

Let G' denote a solution graph from node n to a set N of nodes of an AND/OR graph G. G' is a subgraph of G.

If n is an element of N, G' consists of the single node n;

otherwise, if n has an outgoing connector, K, directed to nodes $\{n_1,\ldots,n_k\}$ such that there is a solution graph to N from each of n_i, where $i = 1,\ldots,k$, then G' consists of node n, the connector, K, the nodes $\{n_1,\ldots,n_k\}$, and the solution graphs to N from each of the nodes in $\{n_1,\ldots,n_k\}$;

otherwise, there is no solution graph from n to N.

Analogous to the use of arc costs in ordinary graphs, it is often useful to assign costs to connectors in AND/OR graphs. (These costs model the costs of rule applications; again we need to assume that each cost is greater than some small positive number, e.) The connector costs can then be used to calculate the cost of a solution graph. Let the cost of a solution graph from any node n to N be denoted by $k(n,N)$. The cost $k(n,N)$ can be recursively calculated as follows:

If n is an element of N, $k(n,N) = 0$.

Otherwise, n has an outgoing connector to a set of successor nodes $\{n_1,\ldots,n_i\}$ in the solution graph. Let the cost of this connector be c_n. Then,

$$k(n,N) = c_n + k(n_1,N) + \ldots + k(n_i,N)\,.$$

We see that the cost of a solution graph, G', from n to N is the cost of the outgoing connector from n (in G') plus the sum of the costs of the solution graphs from the successors of n (in G') to N. This recursive definition is satisfactory because we are assuming acyclic graphs.

Note that our definition of the cost of a solution graph might count the costs of some connectors in the solution graph more than once. In general, the cost of an outgoing connector from some node m is counted in the cost of a solution graph from n to N just as many times as there are paths from n to m in the solution graph. Thus, the costs of the two solution graphs in Figure 3.2 are 8 and 7 if the cost of each k-connector is k.

Beyond merely finding *any* solution graph from the start node to a set of terminal nodes, we may want to find one having minimal cost. We call such a solution graph an *optimal* solution graph. Let the cost of an optimal solution graph from n to a set of terminal nodes be denoted by the function $h^*(n)$.

3.2. AO*: A HEURISTIC SEARCH PROCEDURE FOR AND/OR GRAPHS

As with ordinary graphs, we define the process of *expanding* a node as the application of a successor operator that generates *all* of the successors of a node (through all outgoing connectors). We might now define a breadth-first search algorithm for searching implicit AND/OR graphs to find solution graphs. Again, since breadth-first procedures are uninformed about the problem domain, they are typically not sufficiently efficient for AI applications. We are naturally led to ask whether some search procedure using an evaluation function with a heuristic component can be devised for AND/OR graphs.

We now describe a search procedure that uses a heuristic function $h(n)$ that is an estimate of $h^*(n)$, the cost of an optimal solution graph from node n to a set of terminal nodes. Just as with **GRAPHSEARCH**, simplifications in the statement of the procedure are possible if h satisfies certain restrictions.

Let us impose a monotone restriction on h, that is, for every connector in the implicit graph directed from node n to successors $n_1,\ldots,n_k$, we assume:

$$h(n) \leq c + h(n_1) + \ldots + h(n_k),$$

where c is the cost of the connector. This restriction is analogous to the monotone restriction on heuristic functions for ordinary graphs. If $h(n) = 0$ for n in the set of terminal nodes, then the monotone restriction implies that h is a lower bound on h^*, that is, $h(n) \leq h^*(n)$ for all nodes n.

Our heuristic search procedure for AND/OR graphs can now be stated as follows:

```
Procedure AO*

1   Create a search graph, $G$, consisting solely of
        the start node, $s$. Associate with node
        $s$ a cost $q(s) = h(s)$.
        If $s$ is a terminal node, label $s$ SOLVED.

2   until $s$ is labeled SOLVED, do:

3     begin

4     Compute a partial solution graph, $G'$,
          in $G$ by tracing down the marked connectors
          in $G$ from $s$. (Connectors of $G$ will be
          marked in a subsequent step.)

5     select any nonterminal leaf node, $n$, of
          $G'$. (We discuss later how this
          selection might be made.)

6     Expand node $n$ generating all of its successors
          and install these in $G$ as successors of $n$.
          For each successor, $n_j$, not already
          occurring in $G$, associate the cost
          $q(n_j) = h(n_j)$.
          Label SOLVED any of these successors that are
          terminal nodes. (See text for discussion of what to
          do in case node $n$ has no successors.)

7     Create a singleton set of nodes, $S$, containing
          just node $n$.

8     until $S$ is empty, do:

9         begin

10        Remove from $S$ a node $m$ such that
              $m$ has no descendants in $G$ occurring
              in $S$.
```

```
11        Revise the cost $q(m)$ for $m$, as follows:
              for each connector directed from $m$ to a
              set of nodes $\{n_{1i}, \ldots, n_{ki}\}$
              compute $q_i(m) = c_i + q(n_{1i}) + \ldots$
              $+ q(n_{ki})$. [The $q(n_{ji})$ have
              either just been computed in a
              previous pass through this inner loop
              or (if this is the first pass) they were
              computed in step 6.]
              Set $q(m)$ to the minimum over all
              outgoing connectors of $q_i(m)$ and
              mark the connector through which this
              minimum is achieved, erasing the previous
              marking if different. If all of the
              successor nodes through this connector
              are labeled SOLVED, then label node $m$
              SOLVED.

12        If $m$ has been marked SOLVED or if the
              revised cost of $m$ is different than its
              just previous cost, then add to $S$ all
              those parents of $m$ such that $m$ is one
              of their successors through a marked
              connector.

13        end

14    end
```

Shouldn't be
Sum
for us

Algorithm **AO*** can best be understood as a repetition of the following two major operations. First, a top-down, graph-growing operation (steps 4-6) finds the best partial solution graph by tracing down through the marked connectors. These (previously computed) marks indicate the current best partial solution graph from each node in the search graph. (Before the algorithm terminates, the best partial solution graph does not yet have all of its leaf nodes terminal, which is why it is called *partial.*) One of the nonterminal leaf nodes of this best partial solution graph is expanded, and a cost is assigned to its successors.

The second major operation in **AO*** is a bottom-up, cost-revising, connector-marking, *SOLVE*-labeling procedure (steps 7-12). Starting with the node just expanded, the procedure revises its cost (using the newly computed costs of its successors) and marks the outgoing connector on the estimated best "path" to terminal nodes. This revised cost estimate is propagated upward in the graph. (Acyclicity of our graphs guarantees no loops in this upward propagation.) The revised cost, $q(n)$, is an updated estimate of the cost of an optimal solution graph from n to a set of terminal nodes. Only the ancestors of nodes having their costs revised can possibly have their costs revised, so only these need be considered. Because we are assuming the monotone restriction on h, cost revisions can only be cost increases. Therefore, not all ancestors need have cost revisions, but only those ancestors having best partial solution graphs containing descendants with revised costs (hence step 12).

When the AND/OR graph is an AND/OR tree, the bottom-up operation can be simplified somewhat (because then each node has only one parent).

To avoid making algorithm **AO*** appear more complex than it already does, we ignored the possibility (in step 6) that the node selected for expansion might not have any successors. This case is easily handled in step 11 by associating a very high q value cost with any node, m, having no successors (or, more generally, any node recognized as not belonging to any solution graph). The bottom-up operation will then propagate this high cost upward, which eliminates any chance that a graph containing this node might be selected as an estimated best solution graph.

Suppose some node n has a finite number of descendants in the implicit AND/OR graph and that these do not comprise a solution graph from n to a set of terminal nodes. Then, eventually, the revised cost, $q(n)$, for node n will have a very high value. The assignment of a very high value, $q(s)$, to the start node can therefore be taken to signal that there is no solution graph from the start node.

What is h computable

It is possible to prove that if there is a solution graph from a given node to a set of terminal nodes, and if $h(n) \leq h^*(n)$ for all nodes, and if h satisfies the monotone restriction, ~~then algorithm~~ **AO*** will terminate in an optimal solution graph. (This optimal solution graph can be obtained by tracing down from s through the marked connectors at termination. The cost of this optimal solution graph is equal to the q value of s at termination.) Thus, we can say that algorithm **AO*** with these restrictions is admissible. We omit the proof of this result here; the interested reader is referred to Martelli and Montanari (1973).

A breadth-first algorithm can be obtained from **AO*** by using $h \equiv 0$. Because such an h function satisfies the monotone restriction (and is a lower bound on h^*), the breadth-first algorithm using it is admissible.

As an example of the use of **AO***, let us consider again the graph of Figure 3.1. Suppose that the following estimates are available:

$h(n_0) = 0, h(n_1) = 2, h(n_2) = 4, h(n_3) = 4,$

$h(n_4) = 1, h(n_5) = 1, h(n_6) = 2, h(n_7) = 0,$

$h(n_8) = 0.$

Let nodes n_7 and n_8 be terminal nodes, and let the cost of each k-connector be k. Note that our h function provides a lower bound on h^* and satisfies the monotone restriction.

The search graphs obtained after various cycles through the outer loop of **AO*** are shown in Figure 3.3. In each graph, the revised q values are shown next to each node; heavy arrows are used to mark connectors, and nodes labeled *SOLVED* are indicated by solid circles. During the first cycle, we expand node n_0; next we expand node n_1, then node n_5, and then node n_4. After node n_4 is expanded, node n_0 is labeled *SOLVED*. The solution graph (with minimal cost equal to 5) is obtained by tracing down through the marked connectors.

We have not yet discussed how **AO*** selects (in step 5) a nonterminal leaf node of the estimated best partial solution graph to expand. Perhaps it would be efficient to select that leaf node most likely to change the estimate of the best partial solution graph. If the estimate of the best partial solution graph never changes, **AO*** must eventually expand all of the nonterminal leaf nodes of this graph anyway. However, if the estimate is eventually going to change to some more nearly optimal graph, the sooner **AO*** makes this change, the better. Possibly the expansion of that leaf node having the *highest h* value would most likely result in a changed estimate. fail first!

As with algorithms **A** and **A*** for ordinary graphs, **AO*** may be modified in a variety of ways to render it more practical in special situations. First, rather than recompute a new estimated best partial solution graph after every node expansion, one might instead expand one

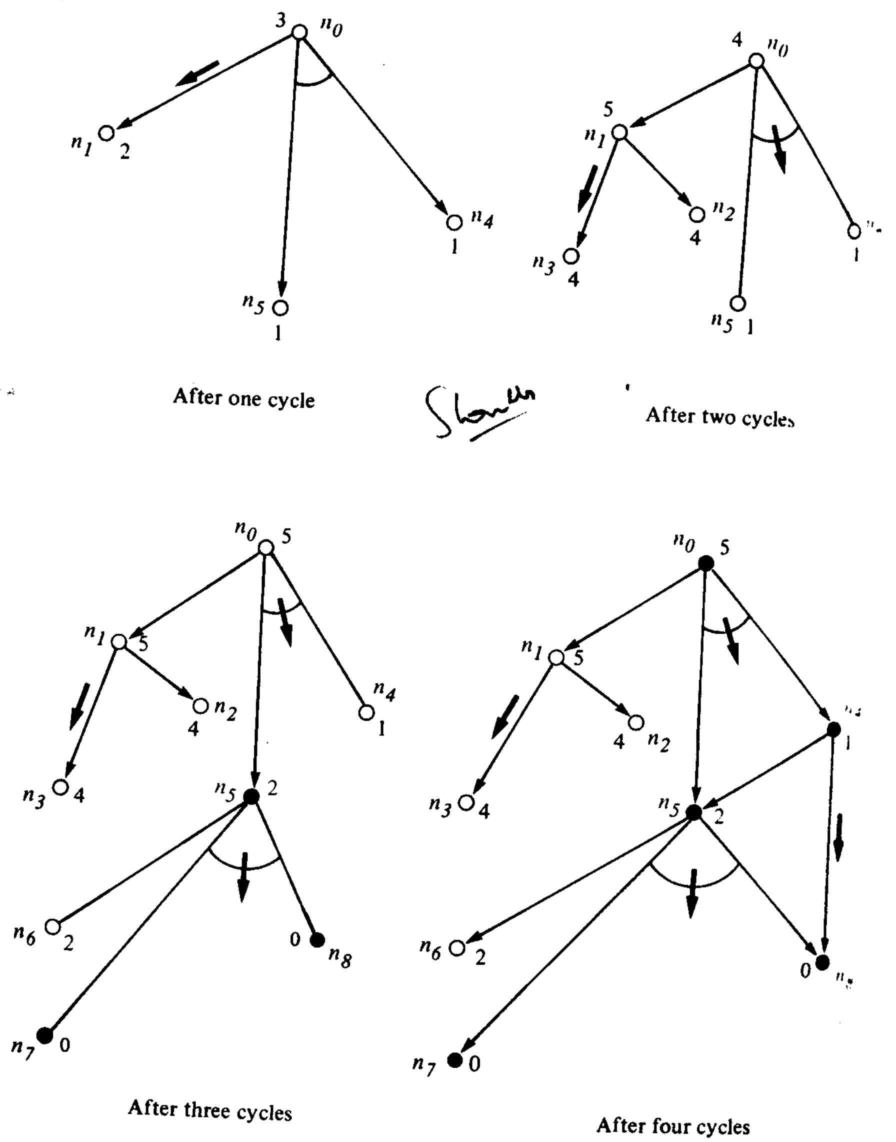

Fig. 3.3 Search graphs after various cycles of **AO***.

or more leaf nodes and some number of their descendants all at once, and then recompute an estimated best partial solution graph. This strategy reduces the overhead expense of frequent bottom-up operations but incurs the risk that some node expansions may not be on the best solution graph.

A staged-search strategy may also be used for AND/OR graphs. To employ it, one periodically reclaims needed storage space by discarding some of the AND/OR search graph. One might, for example, determine a few of those partial solution graphs within the entire search graph having the *largest* estimated costs. These can then be discarded periodically (with the risk, of course, of discarding one that might turn out to be the top of an optimal solution graph.)

3.3. SOME RELATIONSHIPS BETWEEN DECOMPOSABLE AND COMMUTATIVE SYSTEMS

In chapter 1 we mentioned that several problems could be solved by production systems working in either forward or backward directions. (Whether one chooses to call a given direction forward, or backward, is often arbitrary.) Here we illustrate that certain types of commutative forward systems are dual to decomposable backward ones.

Suppose that we have a production system based on the following rewrite rules:

R1: $T \rightarrow A, B$

R2: $T \rightarrow B, C$

R3: $A \rightarrow D$

R4: $B \rightarrow E, F$

R5: $B \rightarrow G$

R6: $C \rightarrow G$