Problem Decomposition

Chapter 6

The problem solving approach we have explored so far is to search for a sequence of moves *starting* from the given state in the state space, or a candidate solution in the solution space, till a desired state or solution is reached. This problem solving strategy is essentially forward looking trial and error. In many situations, one can instead reason backwards from the goal to determine what needs to be done. That is, the search algorithm reasons in a backward fashion from the goal, rather than in the forward direction from the given. Of course, in some problems this may not make a difference, for example in path finding in a city map where the two approaches are equivalent. But in some problems, backward reasoning can lead to breaking up the problem into smaller parts that can be tackled independently, leading to smaller search spaces.

Consider, for example, the task of designing a treat for your friend. The "moves" could be choosing from different activities, and the goal could be an evening plan acceptable to your friend. Let us say that the evening plan constitutes three phases. You start with some activity, followed by a movie and dinner. Let the options be,

Evening → Visit Mall | Visit Beach
Movie → The Matrix | Artificial Intelligence: AI | Bhuvan Shome | Seven Samurai
Dinner → Pizza Hut | Saravana Bhavan

where the above *productions* represent the choices for each phase. Let us say that the forward search program traverses the tree shown in Figure 6.1 before terminating. Your friend is happy with a walk on the beach followed by the movie 'The Matrix', and dinner at Saravana Bhavan, as shown by the leaf node in grey in the figure. One can inspect the tree in some order, but observe that it is fruitless to search in the left subtree, which has *Visit Mall* as the activity. But a depth first search will end up doing exactly that, spending time searching the *entire* subtree below *Visit Mall*, before moving to the right half of the search tree. When it fails in one subtree below, it backtracks to the *last* choice made and tries the next subtree. That is, it does *chronological* backtracking.

One problem with chronological backtracking that *DFS* does is that it simply goes back to the last choice point and tries the next option. If after trying the first combination (Visit Mall, The Matrix, Pizza Hut), the algorithm somehow knew that the culprit for failure was the Visit-Mall choice, it would backtrack directly to trying the next option at that level. We will visit this strategy, known as *dependency-directed backtracking* in Chapter 9. Meanwhile, let us focus on problem decomposition with backward reasoning.

The *DFS* search formulation above is a bottom-up approach, in which the algorithm synthesizes different combinations of primitive moves, and then tests for the goal being achieved. In this chapter, we explore an alternative approach in which we view problem solving as a top-down process. The main idea is that problems can be decomposed into subproblems. This decomposition process continues till we have problems that are trivial to solve. This could happen when we have a library of simple (primitive) problems and their solutions. This approach assumes that the problems can be decomposed into smaller problems that can be solved independently, in smaller search spaces.

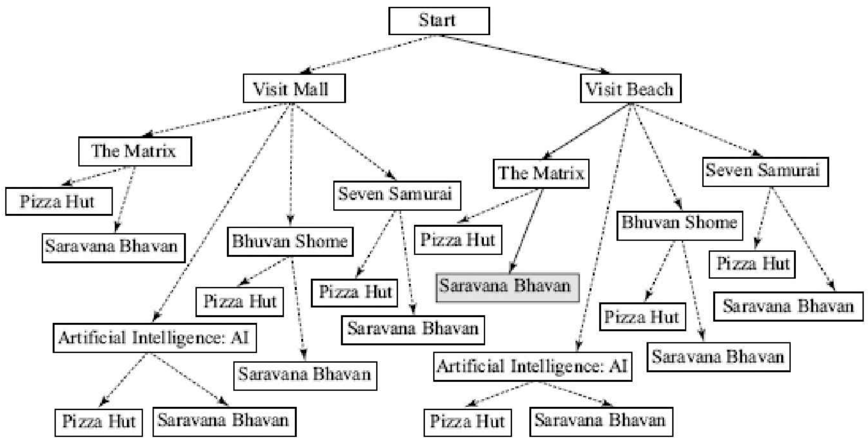


Figure 6.1 A search tree for planning an evening out. The desired plan is marked by the shaded node. Depth First Search will search through from left to right.

For the above problem of planning an evening out, we decompose the problem into three parts to be solved independently. They correspond to the three phases of the evening plan. The resulting search tree is depicted in Figure 6.2. Notice that the decomposition happens at the top level in this problem. This is indicated in the figure by connecting the three edges emanating from the root. In general, decomposition could also happen lower down as one breaks down a subproblem further.

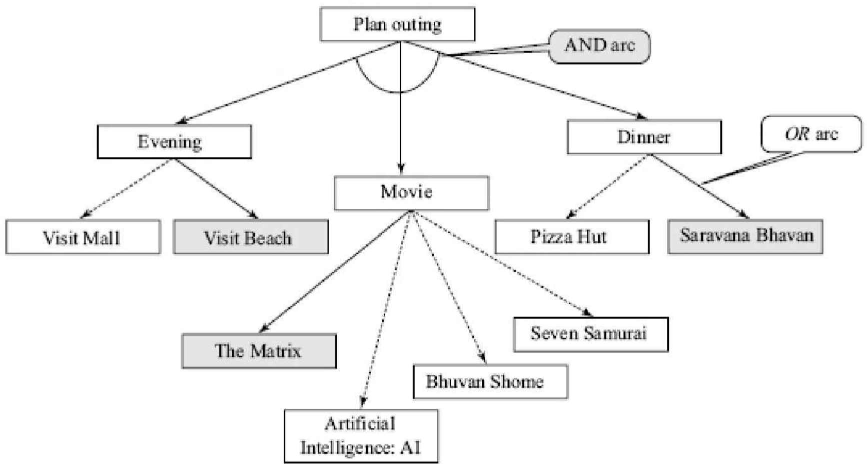


Figure 6.2 An *AND*-OR tree. And arcs represent subproblems to be solved individually. Notice that the solution is subtree rather than a path.

The semantics of an *AND* arc is that *all* the connecting edges have to be traversed. The three edges in the figure together can be thought of as a hyper-edge. Traditionally, such edges are known as *AND* edges, because all individual edges have to be traversed. In contrast, the other edges (the kind that we have been dealing with all along in the preceding chapters), are known as *OR* edges. This is because one could go down one edge, *or* the next one, *or* the next one. If a node has only an *AND* edge coming out of it, we will call it an *AND* node. Likewise, if it has only *OR* edges coming out, we will call it an *OR* node. In general, if we have a graph with both kinds of nodes, we can always convert them into graphs with pure *AND* and *OR* nodes by an addition of extra nodes, as illustrated in Figure 6.3. These search spaces are also known as *goal trees*, (Charniak and McDermott, 1985) because they break up a goal (problem) into subgoals (subproblems). We also call them *AND*/*OR* *trees/graphs* or *AO trees/graphs*.

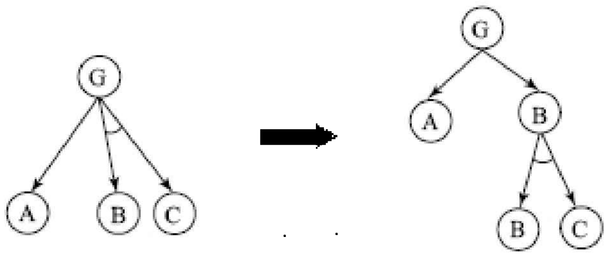


Figure 6.3 An AO graph with mixed nodes can be converted into a graph with pure *AND* and *OR* nodes.

Since one has to traverse all the edges at an *AND* node, the solution obtained will not be a path, but a subtree (or a subgraph) in the *AND*/*OR* search space. This is illustrated as solid arcs in Figure 6.2 for the *AND*/*OR* problem formulated above. Observe that one can still use a depth first search approach, with the modification that at each *AND* node, more than one search will be spawned. In the above example, these will be three searches. The first will search below the node labelled *Outing*, the second below the node labelled *Movie*, and the third below the node *Dinner*. But since they will search independently, they will not explore fruitless combinations as done by our first formulation. In general, of course, an *AND*/*OR* search space will be much larger, with many *AND* and *OR* levels, and we will still need to adopt a heuristic approach.

We look at a couple of examples where *AND/OR* graphs have been used for building problem solvers.

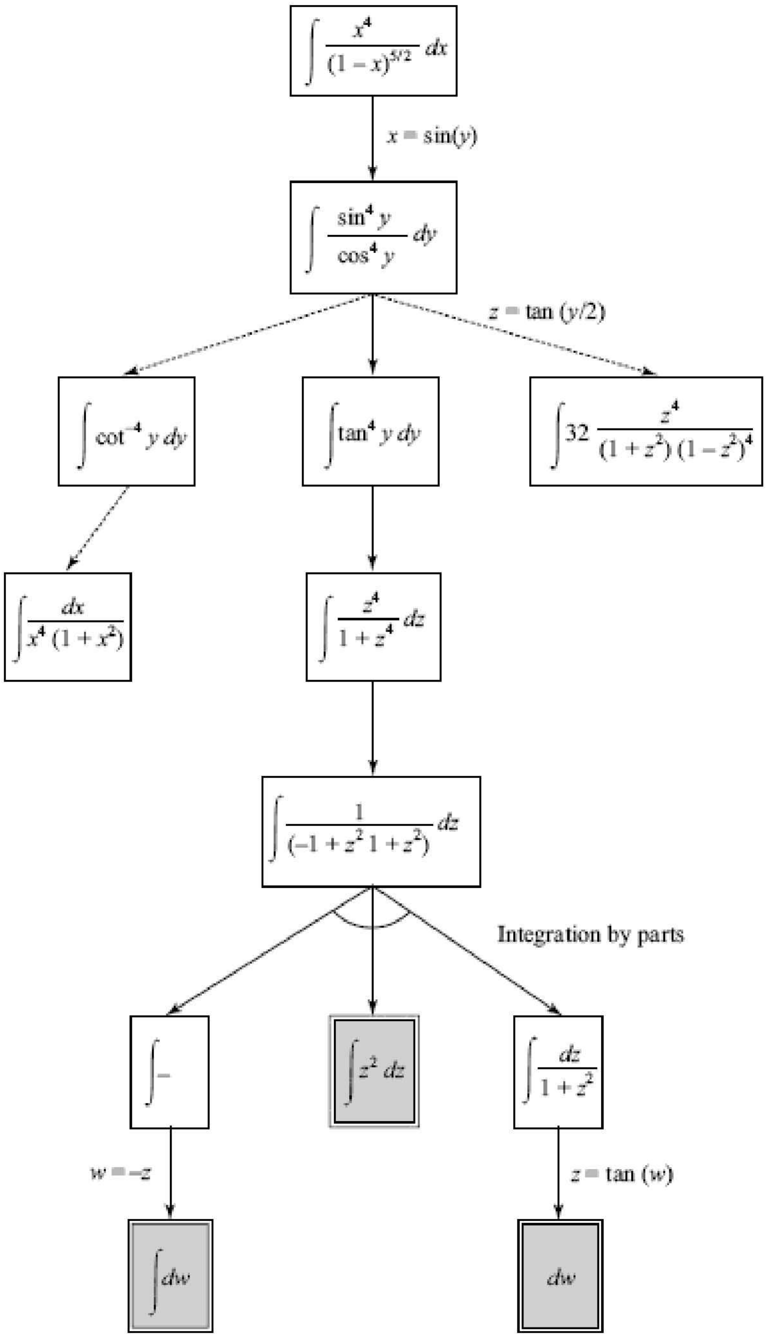
6.1 *SAINT*

One of the first AI systems that used *AO* graphs was *SAINT*, developed as part of his doctoral thesis by James Slagle (1961). *SAINT* was designed to solve symbolic problems in mathematics (Slagle, 1963), and was a precursor to many subsequent systems. We look at an example where *SAINT* solves an integral equation by searching for transformations and problem decomposition. The given problem is,

$$\int \frac{x^4}{(1-x^2)^{5/2}}\,dx$$

Through a series of transformations as shown in Figure 6.4, *SAINT* breaks it down to three simple problems that can be solved trivially. The reverse transformations are applied to the three solutions to finally get the solution,

$$\frac{1}{3}\tan^3(\arcsin x) - \tan(\arcsin x) \ + \arcsin x$$



$\int \frac{x^4}{(1-x)^{5/2}}\,dx$
$x = \sin(y)$
$\int \frac{\sin^4 y}{\cos^4 y}\,dy$
$z = \tan(y/2)$
$\int \cot^{-4} y\,dy$
$\int \tan^4 y\,dy$
$\int 32 \frac{z^4}{(1+z^2)(1-z^2)^4}$
$\int \frac{dx}{x^4(1+x^2)}$
$\int \frac{z^4}{1+z^4}\,dz$
$\int \frac{1}{(-1+z^2 1+z^2)}\,dz$
Integration by parts
$\int -$
$\int z^2\,dz$
$\int \frac{dz}{1+z^2}$
$w = -z$
$z = \tan(w)$
$\int dw$
dw

Figure 6.4 Symbolic integration as an *AND-OR* search problem (Nilsson, 1971).

6.2 Dendral

One of the earliest successes of AI was the program *Dendral* (for Dendritic Algorithm), developed at Stanford University during 1965–1980, by a team led by Joshua Lederberg, Edward Feigenbaum, Bruce Buchanan and Carl Djerassi (Lindsay et al., 1980, 1993). The *Dendral* program was the first AI program to emphasize the power of specialized knowledge over generalized problem-solving methods.

The objective of the program was to assist chemists in the task of determining the structure of a chemical compound. This problem is important because the chemical and physical properties of compounds are determined not just by what their constituent atoms are, but by the arrangement of these atoms as well[1]. The difficulty is that the number of candidate structures (called hypotheses) for a given compound can be very large, and grows combinatorially (running into millions) as we consider molecules with more and more atoms. *Dendral* led to a program called *CONGEN* (CONstrained GENerator) that allows a chemist to constrain the generation of candidates. Figure 6.5 shows some of the hypotheses generated by *CONGEN* for the compound $C_6H_{13}NO_2$.

The candidates generated were used as inputs to a system that produced synthetic spectrograms which were then tested against real spectrograms of the given compounds, before presenting them to the user. The key to its success was to use its knowledge of chemistry in the form of rules about allowed and taboo connections, to consider only plausible candidates. The success of the program is illustrated by the following quote:

> *"By observing structural constraints within molecules which made certain combinations of atoms implausible, generating and testing hypotheses about the identity of the compound, and ruling out candidates that did not fit within the structural constraints, Dendral traced branches of a tree chart that contained all possible configurations of atoms, until it reached the configuration that matched the instrument data most closely. Hence, its name, from "dendron, "the Greek word for tree. … In its practical utilization, Dendral was designed to relieve chemists of a task that was demanding, repetitive, and time-consuming: surveying a large number of molecular structures, to find those that corresponded to instrument data. Once fully operational, the program performed this task with greater speed than an expert spectrometrist, and with comparable accuracy."*[2]

Figure 6.6 illustrates the kind of search space explored by *Dendral*. The reader will recognize it as an *AO* graph.

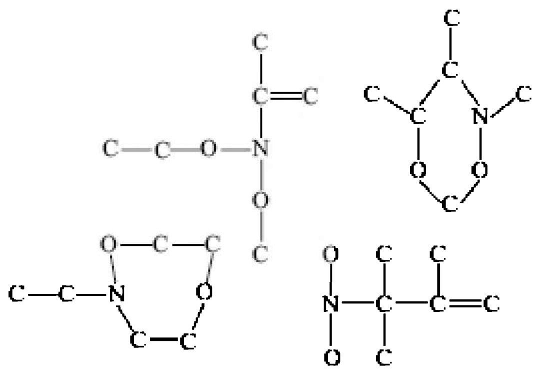


Figure 6.5 Some of the different candidate structures for $C_6H_{13}NO_2$ generated by *CONGEN* (Buchanan, 82). The hydrogen atoms are not shown. They can be filled up, based on valence.

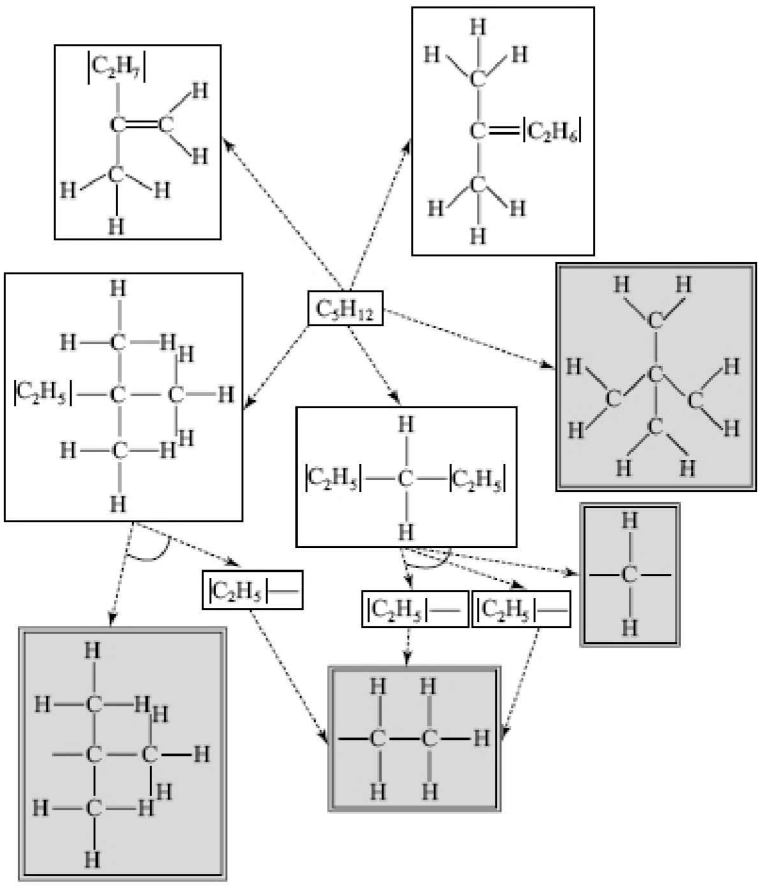


Figure 6.6 The search space for *Dendral* is an AO graph (figure *adapted* from (Mainzer, 2003) Chapter 6). The double line nodes are SOLVED nodes.

6.3 Goal Trees

We look at a heuristic algorithm to search goal trees. We assume that we have a heuristic function that estimates the cost of solving each node.

The solution of an *AO* problem involves reduction to a set of primitive problems which have trivial solutions. We will label the primitive problems in the *AO* graph as *SOLVED*, to indicate that no further reduction is required. We will assume that *SOLVED* nodes have a certain cost associated with them. This cost may be zero in problems like symbolic integration, where the solution is directly available. This cost may also be nonzero for some problems. For example, if we were to pose the problem of constructing a house then one of the primitives of our system could be “install a door”, which would have a cost associated with it. We may also have costs associated with each arc or edge, representing the problem transformation cost. Whatever the costs associated with a problem, we will assume that a heuristic function estimates the total cost of *solving* a given node. Furthermore, we will require the heuristic function to be a lower bound on the actual cost of solving a node, in order to ensure that the optimal cost solution is found. The argument for underestimating the actual cost is similar to the one presented for *Branch&Bound* and *A** algorithms seen in Chapter 4. That is, as long as there is the possibility of finding a cheaper solution, the algorithm will continue searching even after one has been found.

Before writing the algorithm for solving goal trees, let us cook up a small problem and investigate how search could progress, using a heuristic function. Let us say that we start with a goal *G* of estimated cost 45, and expand it to get two ways of solving it, one an *OR* arc and the other an *AND* arc as shown in Figure 6.7. Let the three successor nodes *A*, *B* and *C* have heuristic values as 42, 22 and 24 respectively. Let the three edges leading to them have costs 4, 3 and 2 respectively. Which node should the algorithm refine (expand) next?

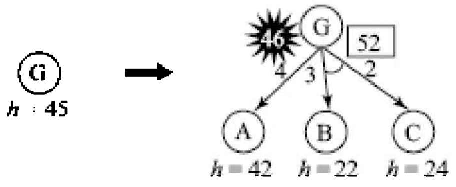


Figure 6.7 Which is the best node to expand next? Even though node *B* has the lowest heuristic value it is a part of a more expensive looking option. This is because it is a leaf on an *AND* (hyper) arc. The estimated cost of that solution is 22 + 3 + 24 + 3 = 52. The more expensive looking node *A* has an estimated cost of 42 + 4 = 46. Note that the choice is made on the basis of backed up values 46 and 52.

Unlike the *A** search for optimal solutions, (paths) a node in the *AO* graph is not a representative of a solution. This is because solutions are not usually paths but subtrees, and a node may have an *AND* sibling (or a cousin) which also contributes to the solution with its own associated cost.

Thus, looking at the heuristic value of a node by itself may not be useful. Even counting the costs of the edges leading to the node (like the *g*-values in *A**) will not help, because the node only accounts for a part of the solution. Instead, we need to look at the total estimated cost of a solution. In the above example, the total estimated cost of the two options is 46 and 52 respectively. After every expansion by the algorithm, the best choice at each node is marked. The search should refine the marked solution. In the above example, this means that it should expand the node *A* next.

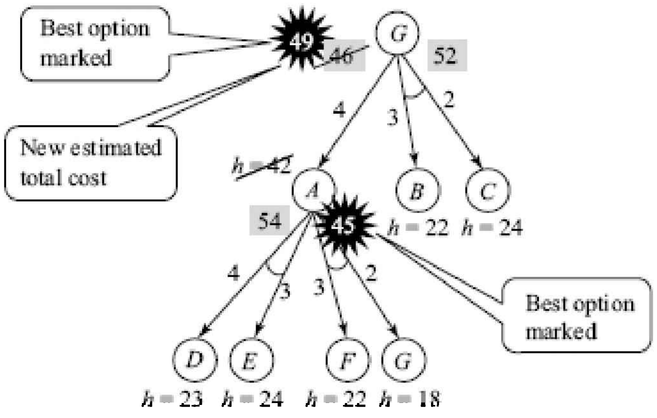


Figure 6.8 After node *A* is expanded, there are two options (*D*, *E*) and (*F*, *G*) with estimated costs 54 and 45 respectively. The revised estimate of *A* now becomes 45, the lower of the two. Propagated back to root, this option has a revised cost of 49. It is still a better option, and AO* will now go down the marked path and expand one of *F* and *G*.

The resulting graph is shown in Figure 6.8 and it represents a typical *AO* graph, not yet solved completely. Observe that there are two choice points in the graph, and at both, the best solution is marked. The topmost task at the root has two options. Let us call them the left option and the right option. In general, of course, there may be any number of options. The left option has been refined and it itself has two options for further decomposition. Again, the best option is marked. The left option, represented by node *A*, had a heuristic estimate of 42 before expansion, which was revised to 45, the better of the two sub-options, represented by nodes *F* and *G*, and marked as the better option. The revised estimate of node *A* has to be propagated up, and the left option at the root now evaluates to 49 instead of 46.

In every cycle of refinement, the algorithm starts at the root. It follows the marked best option at each choice point, until it reaches some yet unsolved node or a set of nodes. It will refine one of them, and then propagate the new values back up again. In the process if the subproblems of a given node are *SOLVED* nodes then it may also propagate *SOLVED* label back. The algorithm will terminate when the *SOLVED* label is backed up right up to the root.

Thus, the algorithm for solving the goal tree, known as the *AO** algorithm (Martelli and Montanari, 1978; Nilsson, 1980), has the following cycle:

- Starting at the root, traverse the graph along marked paths till the algorithm reaches a set of unsolved nodes *U*.
- Pick a node *n* from *U* and refine it.
- Propagate the revised estimate of *n* up via all ancestors.
- If for a node all *AND* successors along the marked path are marked *SOLVED*, mark it *SOLVED* as well.
- If a node has *OR* edges emanating from it, and the cheapest successor is marked *SOLVED* then mark the node *SOLVED*.
- Terminate when the root node is marked *SOLVED*.

The detailed algorithm given in Figure 6.9 below has been adopted from (Rich and Knight, 1991).

6.3.1 An Example Trace of *AO**

Let us look at a complete example depicting the progress of the *AO** algorithm on a synthetic problem. We begin with a version of a problem where the heuristic function is *not* a lower bound on the actual cost of solving each node. In the following example (Figure 6.10), assume that the *SOLVED* nodes all have an associated cost zero. The labels on the nodes are heuristic values. Let the cost of every arc in the graph be one. This means that the cost of solving a node is dependent only on the number of arcs leading from it to *SOLVED* nodes. A cursory glance at the figure reveals that the heuristic function shown in the figure is quite wild. It definitely overestimates the cost of solving nodes specially the ones closer to *SOLVED* nodes.

Figures (6.11 continued in 6.12) show the progress of the *AO** algorithm. At each stage, the algorithm goes down the marked path and expands the node shown in bold. The backed-up values in the nodes are the best known cost estimates for that node, backed up after each expansion.

After the last node in the above sequence is refined, the algorithm terminates with a solution as shown in Figure 6.12.

The cost of the solution found is 8, but the heuristic functions in many places had much higher estimates. An overestimating heuristic function makes the search opinionated. It refuses to consider unseen alternatives because it estimates them to be worse. In the process, it may miss better solutions. For example, the *AO* graph has a better solution costing 7 units which the algorithm misses.

```
Algorithm AO*()
  /* uses a graph G instead of open and closed
   the graph G is initialized to the start node start */
  G ← start
  Compute h(start)
  while start is not labeled SOLVED AND h(start) ≤ Futility
   do          /* Futility is the maximum cost solution acceptable */
     /* Forward phase */
     Trace the marked path leading to a set U of unexpanded nodes
     Select node n from U
       children ← successors of n
       if children is EMPTY
         then h(n) ←  Futility
         else  Check for looping in members of children
               Remove any looping members
               for each s ∈ children
                 do  Add s to G
                   if s is primitive
                     then   Label s SOLVED
                            Compute h(s)        /* could be zero*/
       /*Propagate Back*/
       /* Let M be the set of nodes that are modified */
       M ← n
       while M is not EMPTY
         do Select deepest node d from M, and remove it from M
           Compute best cost of d from its children
           Mark best option at d as MARKED
           if all nodes connected through marked arc are labeled SOLVED
             then Label d as SOLVED
           if d has changed
             then Add all parents of d to M
  if start is marked SOLVED
     then return marked subgraph starting at start
     else return FAILURE
```

Figure 6.9 Algorithm *AO**.

The way to ensure that the optimal solution is found is by making sure the heuristic function underestimates the cost of solving a node. In this problem, this can be done by dividing the heuristic value by ten. Equivalently, we can assume each arc to cost 10 units, which makes the heuristic values given in problems as underestimating the actual costs. The progress of the algorithm is shown in Figures 6.13 and 6.14.

The underestimating function does not grab the first solution in sight, and continues searching till no better options are left. The progress of the algorithm is continued below.

The algorithm terminates with the optimal solution costing 70 shown in the right in Figure 6.15. In fact, at that point, it has also discovered another optimal solution shown in the figure on the left.

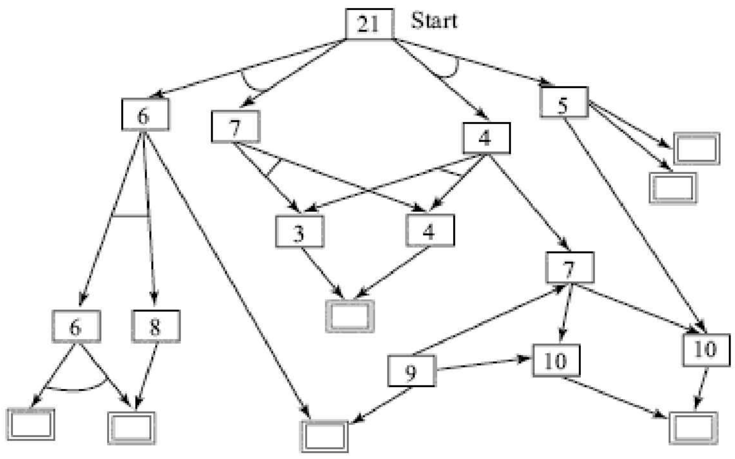


Figure 6.10 A synthetic *AND*/*OR* problem. Assume every arc costs one unit to traverse. Nodes are labelled with heuristic values. Solved nodes represented by double-lined boxes have cost zero.

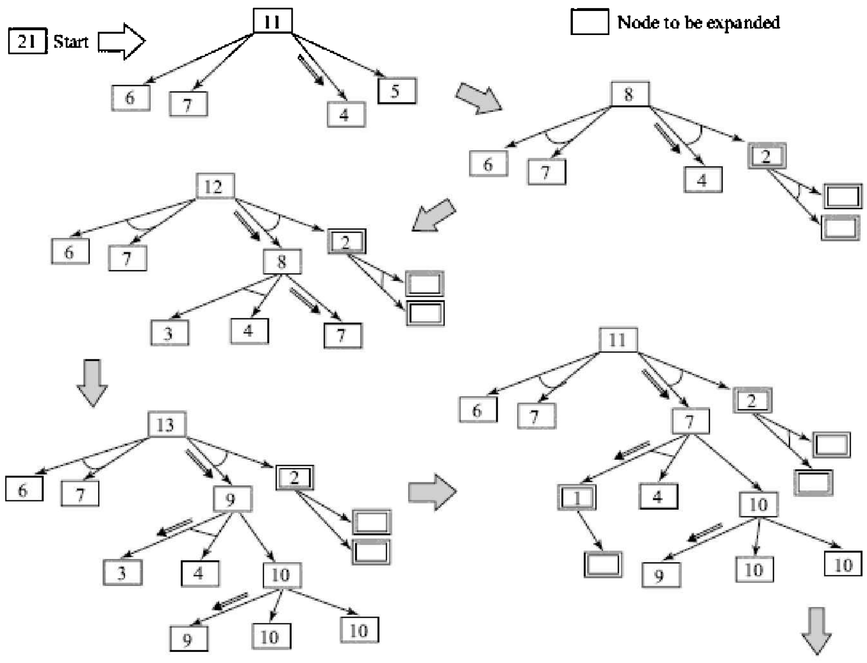


Figure 6.11 The progress of the *AO** algorithm. The best options at each choice point are marked by arrows. Nodes in double-line squares are *SOLVED*.

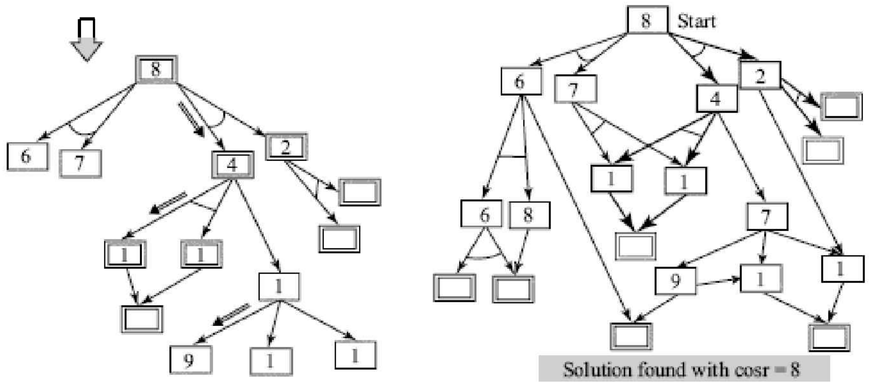


Figure 6.12 The algorithm terminates as the root (start) is labelled *SOLVED* (double square). The solution is shown in bold lines on the right.

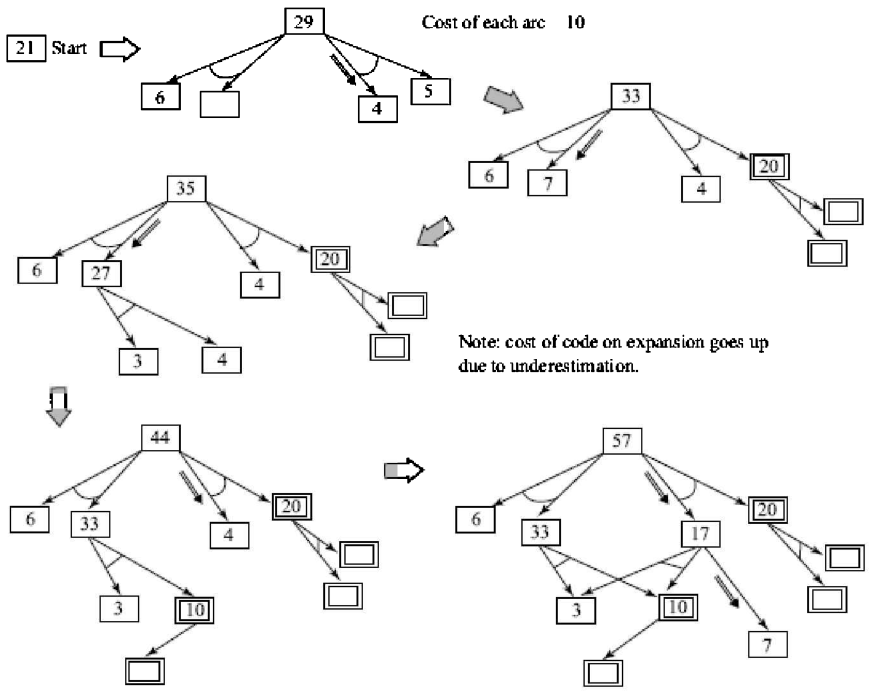


Figure 6.13 Searching with an underestimating function. Observe that an underestimating heuristic function makes the search try out more options at each stage. Searching with an overestimating function had plunged down one path.

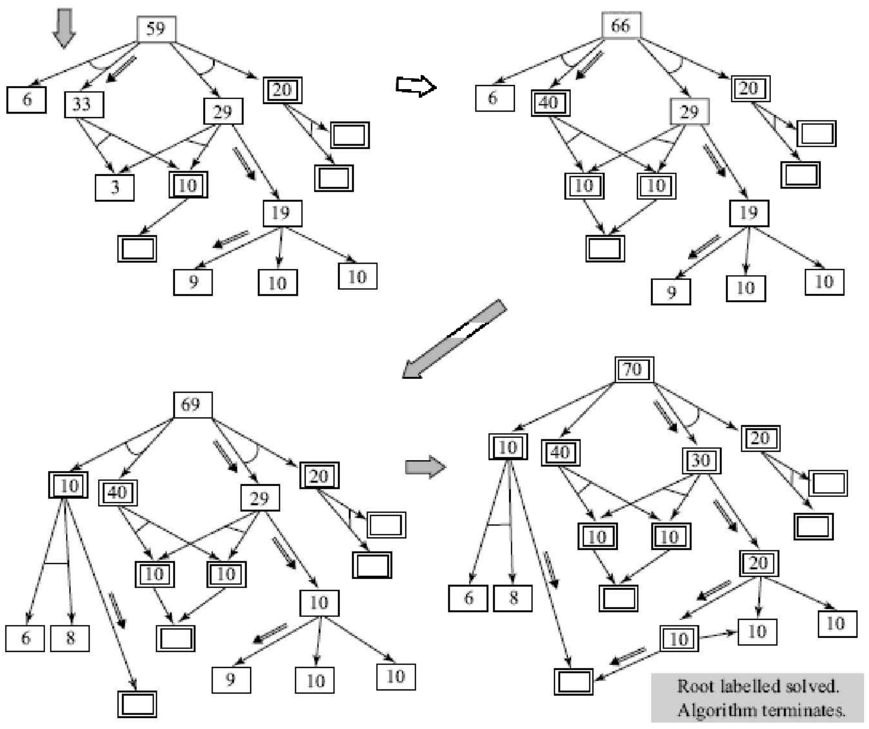


Figure 6.14 *AO** terminates after exploring a larger search space, but finds an optimal solution.

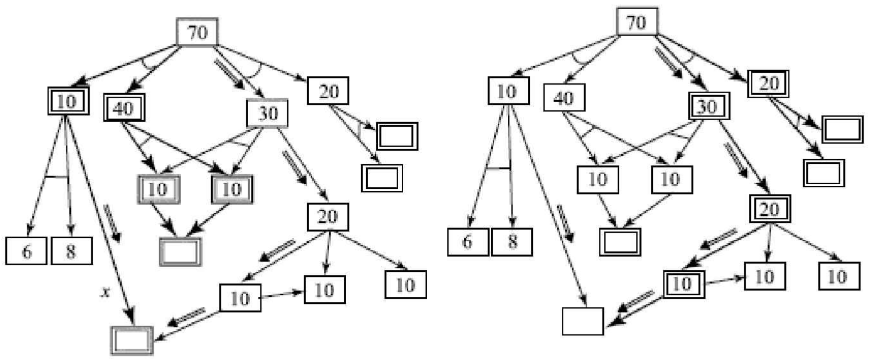


Figure 6.15 *AO** terminates with the optimal solution on the right with cost 70. It has also found the solution on the left at termination. If the arc marked ***x*** were to be 9 instead of ten, it would have reported that solution with cost 69.

6.4 Rule Based Systems

Rule based systems or *production systems* have been used in general to decompose a problem and address it in parts. In its most abstract form, a rule or a production is a statement of the form,