3.1 Generate-and-Test

The generate-and-test strategy is the simplest of all the approaches we discuss. It consists of the following steps:

Algorithm: Generate-and-Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.
2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" may be a very long time.

The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can, of course, also operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found. In this form, it is also known as the British Museum algorithm, a reference to a method for finding an object in the British Museum by wandering randomly.[1] Between these two extremes lies a practical middle ground in which the search process proceeds systematically, but some paths are not considered because they seem unlikely to lead to a solution. This evaluation is performed by a heuristic function, as described in Section 2.2.2.

The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, as described above, to traverse a graph rather than a tree.

For simple problems, exhaustive generate-and-test is often a reasonable technique. For example, consider the puzzle that consists of four six-sided cubes, with each side of each cube painted one of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person (who is a much slower processor for this sort of thing than even a very cheap computer) in several minutes by systematically and exhaustively trying all possibilities. It can be solved even more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. As many of them as possible should be placed to abut the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

[1] Or, as another story goes, if a sufficient number of monkeys were placed in front of a set of typewriters and left alone long enough, then they would eventually produce all of the works of Shakespeare.

Scanned by CamScanner

Unfortunately, for problems much harder than this, even heuristic generate-and-test, all by itself, is not a very effective technique. But when combined with other techniques to restrict the space in which to search even further, the technique can be very effective.

For example, one early example of a successful AI program is DENDRAL [Lindsay *et al.*, 1980], which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data. It uses a strategy called *plan-generate-test*, in which a planning process that uses constraint-satisfaction techniques (see Section 3.5) creates lists of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way, the generate-and-test procedure has proved highly effective.

This combination of planning, using one problem-solving method (in this case, constraint satisfaction) with the use of the plan by another problem-solving method, generate-and-test, is an excellent example of the way techniques can be combined to overcome the limitations that each possesses individually. A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only to produce pieces of solutions that will then be exploited in the generate-and-test process, the lack of detailed accuracy becomes unimportant. And, at the same time, the combinatorial problems that arise in simple generate-and-test are avoided by judicious reference to the plans.

Scanned by CamScanner

3.2 Hill Climbing (feed back from test procedure)

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. But if the test function is augmented with a heuristic function[2] that provides an estimate of how close a given state is to a goal state, the generate procedure can exploit it as shown in the procedure below. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

Recall from Section 2.3.4 that one way to characterize problems is according to their answer to the question, "Is a good solution absolute or relative?" Absolute solutions exist whenever it is possible to recognize a goal state just by examining it. Getting downtown is an example of such a problem. For these problems, hill climbing can terminate whenever a goal state is reached. Only relative solutions exist, however, for maximization (or minimization) problems, such as the traveling salesman problem. In these problems, there is no *a priori* goal state. For problems of this sort, it makes sense to terminate hill climbing when there is no reasonable alternative state to move to.

---

[2] What we are calling the heuristic function is sometimes also called the *objective function*, particularly in the literature of mathematical optimization.

Scanned by CamScanner

3.2.1 Simple Hill Climbing

The simplest way to implement hill climbing is as follows.

Algorithm: Simple Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   (b) Evaluate the new state.
      i. If it is a goal state, then return it and quit.
      ii. If it is not a goal state but it is better than the current state, then make it the current state.
      iii. If it is not better than the current state, then continue in the loop.

The key difference between this algorithm and the one we gave for generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process. It is the use of such knowledge that makes this and the other methods discussed in the rest of this chapter *heuristic* search methods, and it is that same knowledge that gives these methods their power to solve some otherwise intractable problems.

Notice that in this algorithm, we have asked the relatively vague question, "Is one state *better* than another?" For the algorithm to work, a precise definition of *better* must be provided. In some cases, it means a higher value of the heuristic function. In others, it means a lower value. It does not matter which, as long as a particular hill-climbing program is consistent in its interpretation.

To see how hill climbing works, let's return to the puzzle of the four colored blocks. To solve the problem, we first need to define a heuristic function that describes how close a particular configuration is to being a solution. One such function is simply the sum of the number of different colors on each of the four sides. A solution to the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration into another. Actually, one rule will suffice. It says simply pick a block and rotate it 90 degrees in any direction. Having provided these definitions, the next step is to generate a starting configuration. This can either be done at random or with the aid of the heuristic function described in the last section. Now hill climbing can begin. We generate a new state by selecting a block and rotating it. If the resulting state is better, then we keep it. If not, we return to the previous state and try a different perturbation.

3.2.2 Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called *steepest-ascent hill*

Scanned by CamScanner

climbing or *gradient search*. Notice that this contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

Algorithm: Steepest-Ascent Hill Climbing (SUCC)

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
   (a) Let *SUCC* be a state such that any possible successor of the current state will be better than *SUCC*.
   (b) For each operator that applies to the current state do:
      i. Apply the operator and generate a new state.
      ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to *SUCC*. If it is better, then set *SUCC* to this state. If it is not better, leave *SUCC* alone.
   (c) If the *SUCC* is better than current state, then set current state to *SUCC*.

To apply steepest-ascent hill climbing to the colored blocks problem, we must consider all perturbations of the initial state and choose the best. For this problem, this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move (usually longer for steepest-ascent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem.

Both basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge.

A *local maximum* is a state that is better than all its neighbors but is not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called *foothills*.

A *plateau* is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

A *ridge* is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

Scanned by CamScanner

There are some ways of dealing with these problems, although these methods are by no means guaranteed:

- Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.

- Make a big jump in some direction to try to get to a new section of the search space. This is a particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.

- Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a particularly good strategy for dealing with ridges.

Even with these first-aid measures, hill climbing is not always very effective. It is particularly unsuited to problems where the value of the heuristic function drops off suddenly as you move away from a solution. This is often the case whenever any sort of threshold effect is present. Hill climbing is a local method, by which we mean that it decides what to do next by looking only at the "immediate" consequences of its choice rather than by exhaustively exploring all the consequences. It shares with other local methods, such as the nearest neighbor heuristic described in Section 2.2.2, the advantage of being less combinatorially explosive than comparable global methods. But it also shares with other local methods a lack of a guarantee that it will be effective. Although it is true that the hill-climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function. Consider the blocks world problem shown in Figure 3.1. Assume the same operators (i.e., pick up one block and put it on the table; pick up one block and put it on another one) that were used in Section 2.3.1. Suppose we use the following heuristic function:

> **Local:** Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

Using this function, the goal state has a score of 8. The initial state has a score of 4 (since it gets one point added for blocks C, D, E, F, G, and H and one point subtracted for blocks A and B). There is only one move from the initial state, namely to move block A to the table. That produces a state with a score of 6 (since now A's position causes a point to be added rather than subtracted). The hill-climbing procedure will accept that move. From the new state, there are three possible moves, leading to the three states shown in Figure 3.2. These states have the scores: (*a*) 4, (*b*) 4, and (*c*) 4. Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not the global maximum. The problem is that by purely local examination of support structures, the current state appears to be better

Scanned by CamScanner

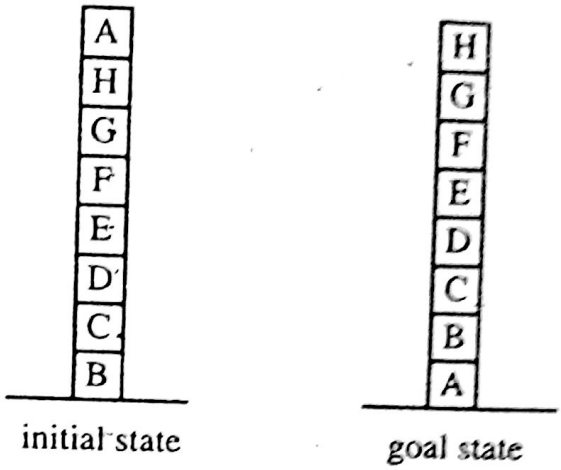


Figure 3.1: A Hill-Climbing Problem

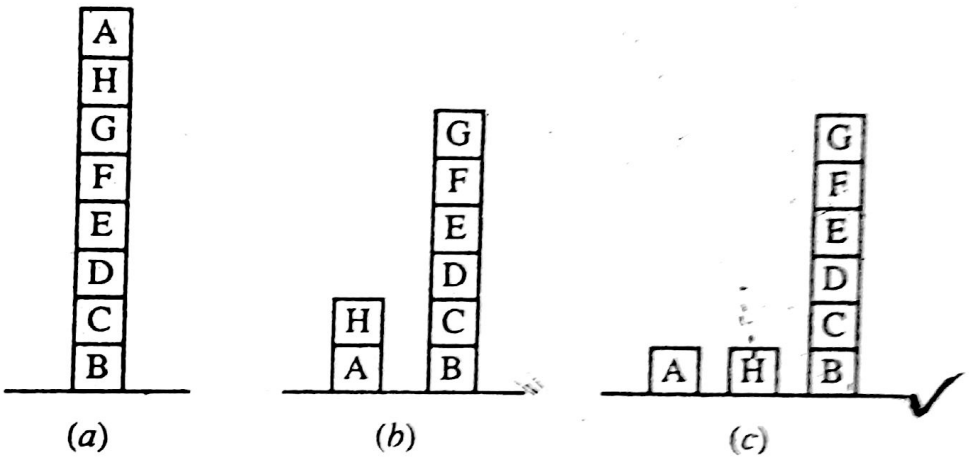


Figure 3.2: Three Possible Moves

than any of its successors because more blocks rest on the correct objects. To solve this problem, it is necessary to disassemble a good local structure (the stack B through H) because it is in the wrong global context.

We could blame hill climbing itself for this failure to look far enough ahead to find a solution. But we could also blame the heuristic function and try to modify it. Suppose we try the following heuristic function in place of the first one:

> **Global:** For each block that has the correct support structure (i.e., the complete structure underneath it is exactly as it should be), add one point for every block in the support structure. For each block that has an incorrect support structure, subtract one point for every block in the existing support structure.

Using this function, the goal state has the score 28 (1 for B, 2 for C, etc.). The initial state has the score −28. Moving A to the table yields a state with a score of −21 since A no

Scanned by CamScanner

longer has seven wrong blocks under it. The three states that can be produced next now have the following scores: (*a*) −28, (*b*) −16, and (*c*) −15. This time, steepest-ascent hill climbing will choose move (*c*), which is the correct one. This new heuristic function captures the two key aspects of this problem: incorrect structures are bad and should be taken apart; and correct structures are good and should be built up. As a result, the same hill climbing procedure that failed with the earlier heuristic function now works perfectly.

Unfortunately, it is not always possible to construct such a perfect heuristic function. For example, consider again the problem of driving downtown. The perfect heuristic function would need to have knowledge about one-way and dead-end streets, which, in the case of a strange city, is not always available. And even if perfect knowledge is, in principle, available, it may not be computationally tractable to use. As an extreme example, imagine a heuristic function that computes a value for a state by invoking its own problem-solving procedure to look ahead from the state it is given to find a solution. It then knows the exact cost of finding that solution and can return that cost as its value. A heuristic function that does this converts the local hill-climbing procedure into a global method by embedding a global method within it. But now the computational advantages of a local method have been lost. Thus it is still true that hill climbing can be very inefficient in a large, rough problem space. But it is often useful when combined with other methods that get it started in the right general neighborhood.

3.2.3 Simulated Annealing

Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau, or a ridge.

In order to be compatible with standard usage in discussions of simulated annealing, we make two notational changes for the duration of this section. We use the term *objective function* in place of the term *heuristic function*.

And we attempt to *minimize* rather than maximize the value of the objective function. Thus we actually describe a process of valley descending rather than hill climbing.

Simulated annealing [Kirkpatrick *et al.*, 1983] as a computational process is patterned after the physical process of *annealing*, in which physical substances such as metals are melted (i.e., raised to high energy levels) and then gradually cooled until some solid state is reached. The goal of this process is to produce a minimal-energy final state. Thus this process is one of valley descending in which the objective function is the energy level. Physical substances usually move from higher energy configurations to lower ones, so the valley descending occurs naturally. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function

$$p = e^{-\Delta E/kT}$$

where ΔE is the positive change in the energy level, T is the temperature, and k is Boltzmann's constant. Thus, in the physical valley descending that occurs during annealing, the probability of a large uphill move is lower than the probability of a small

Scanned by CamScanner

one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Thus such moves are more likely during the beginning of the process when the temperature is high, and they become less likely at the end as the temperature becomes lower. One way to characterize this process is that downhill moves are allowed anytime. Large upward moves may occur early on, but as the process progresses, only relatively small upward moves are allowed until finally the process converges to a local minimum configuration.

The rate at which the system is cooled is called the *annealing schedule*. Physical annealing processes are very sensitive to the annealing schedule. If cooling occurs too rapidly, stable regions of high energy will form. In other words, a local but not global minimum is reached. If, however, a slower schedule is used, a uniform crystalline structure, which corresponds to a global minimum, is more likely to develop. But, if the schedule is too slow, time is wasted. At high temperatures, where essentially random motion is allowed, nothing useful happens. At low temperatures a lot of time may be wasted after the final structure has already been formed. The optimal annealing schedule for each particular annealing problem must usually be discovered empirically.

These properties of physical annealing can be used to define an analogous process of simulated annealing, which can be used (although not always effectively) whenever simple hill climbing can be used. In this analogous process, ΔE is generalized so that it represents not specifically the change in energy but more generally, the change in the value of the objective function, whatever it is. The analogy for kT is slightly less straightforward. In the physical process, temperature is a well-defined notion, measured in standard units. The variable k describes the correspondence between the units of temperature and the units of energy. Since, in the analogous process, the units for both E and T are artificial, it makes sense to incorporate k into T, selecting values for T that produce desirable behavior on the part of the algorithm. Thus we use the revised probability formula

$$p' = e^{-\Delta E/T}$$

But we still need to choose a schedule of values for T (which we still call temperature). We discuss this briefly below after we present the simulated annealing algorithm.

The algorithm for simulated annealing is only slightly different from the simple hill-climbing procedure. The three differences are:

- The annealing schedule must be maintained.
- Moves to worse states may be accepted.
- It is a good idea to maintain, in addition to the current state, the best state found so far. Then, if the final state is worse than that earlier state (because of bad luck in accepting moves to worse states), the earlier state is still available.

Algorithm: Simulated Annealing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize *BEST-SO-FAR* to the current state.

Scanned by CamScanner

3. Initialize T according to the annealing schedule.

4. Loop until a solution is found or until there are no new operators left to be applied in the current state.

   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

   (b) Evaluate the new state. Compute

   $$\Delta E = (\text{value of current}) - (\text{value of new state})$$

   - If the new state is a goal state, then return it and quit.
   - If it is not a goal state but is better than the current state, then make it the current state. Also set *BEST-SO-FAR* to this new state.
   - If it is not better than the current state, then make it the current state with probability p' as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0,1]. If that number is less than p', then the move is accepted. Otherwise, do nothing.

   (c) Revise T as necessary according to the annealing schedule.

5. Return *BEST-SO-FAR*, as the answer.

To implement this revised algorithm, it is necessary to select an annealing schedule, which has three components. The first is the initial value to be used for temperature. The second is the criteria that will be used to decide when the temperature of the system should be reduced. The third is the amount by which the temperature will be reduced each time it is changed. There may also be a fourth component of the schedule, namely, when to quit. Simulated annealing is often used to solve problems in which the number of moves from a given state is very large (such as the number of permutations that can be made to a proposed traveling salesman route). For such problems, it may not make sense to try all possible moves. Instead, it may be useful to exploit some criterion involving the number of moves that have been tried since an improvement was found.

Experiments that have been done with simulated annealing on a variety of problems suggest that the best way to select an annealing schedule is by trying several and observing the effect on both the quality of the solution that is found and the rate at which the process converges. To begin to get a feel for how to come up with a schedule, the first thing to notice is that as T approaches zero, the probability of accepting a move to a worse state goes to zero and simulated annealing becomes identical to simple hill climbing. The second thing to notice is that what really matters in computing the probability of accepting a move is the ratio $\Delta E/T$. Thus it is important that values of T be scaled so that this ratio is meaningful. For example, T could be initialized to a value such that, for an average ΔE, p' would be 0.5.

Chapter 18 returns to simulated annealing in the context of neural networks.

Scanned by CamScanner

3.3 Best-First Search

Until now, we have really only discussed two systematic control strategies, breadth-first search and depth-first search (of several varieties). In this section, we discuss a new method, best-first search, which is a way of combining the advantages of both depth-first and breadth-first search into a single method.

3.3.1 OR Graphs

Depth-first search is good because it allows a solution to be found without all competing branches having to be expanded. Breadth-first search is good because it does not get trapped on dead-end paths. One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.

At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues. Usually what happens is that a bit of depth-first searching occurs as the most promising branch is explored. But eventually, if a solution is not found, that branch will start to look less promising than one of the top-level branches that had been ignored. At that point, the now more promising, previously ignored branch will be explored. But the old branch is not forgotten. Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough that it is again the most promising path.

Figure 3.3 shows the beginning of a best-first search procedure. Initially, there is only one node, so it will be expanded. Doing so generates three new nodes. The heuristic function, which, in this example, is an estimate of the cost of getting to a solution from a given node, is applied to each of these new nodes. Since node D is the most promising, it is expanded next, producing two successor nodes, E and F. But then the heuristic function is applied to them. Now another path, that going through node B, looks more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J. At the next step, J will be expanded, since it is the most promising. This process can continue until a solution is found.

Notice that this procedure is very similar to the procedure for steepest-ascent hill climbing, with two exceptions. In hill climbing, one move is selected and all the others are rejected, never to be reconsidered. This produces the straightline behavior that is characteristic of hill climbing. In best-first search, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising.[3] Further, the best available state is selected in best-first search, even if that state has a value that is lower than the value of the state that was just explored. This

[3] In a variation of best-first search, called *beam search*, only the n most promising states are kept for future consideration. This procedure is more efficient with respect to memory but introduces the possibility of missing a solution altogether by pruning the search tree too early.

Scanned by CamScanner

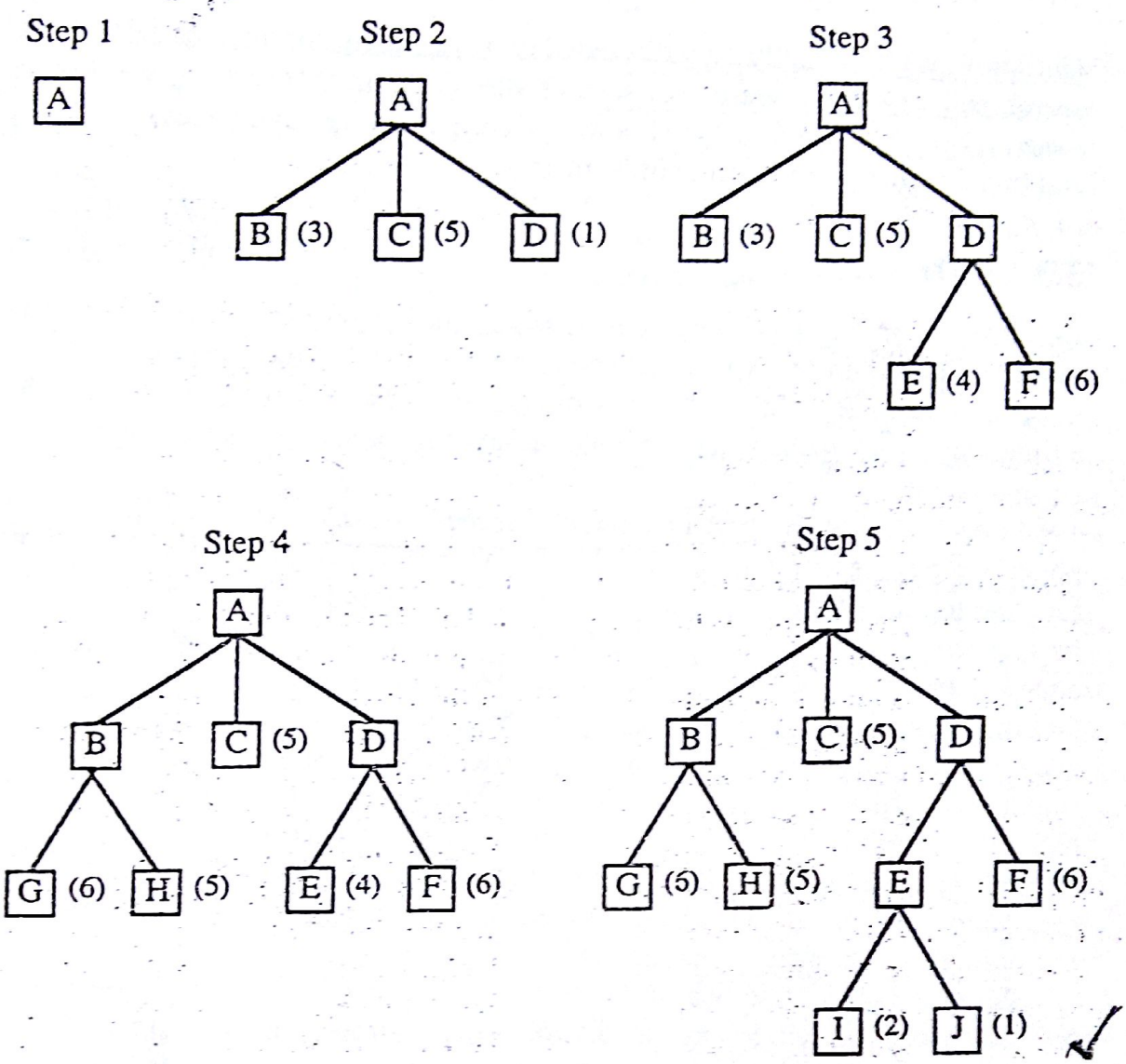


Figure 3.3: A Best-First Search

contrasts with hill climbing, which will stop if there are no successor states with better values than the current state.

Although the example shown above illustrates a best-first search of a tree, it is sometimes important to search a graph instead so that duplicate paths will not be pursued. An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain, in addition to a description of the problem state it represents, an indication of how promising it is, a parent link that points back to the best node from which it came, and a list of the nodes that were generated from it. The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors. We will call a graph of this sort an *OR graph*, since each of its branches represents an alternative problem-solving path.

To implement such a graph-search procedure, we will need to use two lists of nodes:

- *OPEN*—nodes that have been generated and have had the heuristic function

Scanned by CamScanner

applied to them but which have not yet been examined (i.e., had their successors generated). *OPEN* is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function. Standard techniques for manipulating priority queues can be used to manipulate the list.

- *CLOSED*—nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

We will also need a heuristic function that estimates the merits of each node we generate. This will enable the algorithm to search more promising paths first. Call this function f' (to indicate that it is an approximation to a function f that gives the true evaluation of the node). For many applications, it is convenient to define this function as the sum of two components that we call g and h'. The function g is a measure of the cost of getting from the initial state to the current node. Note that g is not an estimate of anything; it is known to be the exact sum of the costs of applying each of the rules that were applied along the best path to the node. The function h' is an estimate of the additional cost of getting from the current node to a goal state. This is the place where knowledge about the problem domain is exploited. The combined function f', then, represents an estimate of the cost of getting from the initial state to a goal state along the path that generated the current node. If more than one path generated the node, then the algorithm will record the best one. Note that because g and h' must be added, it is important that h' be a measure of the cost of getting from the node to a solution (i.e., good nodes get low values; bad nodes get high values) rather than a measure of the goodness of a node (i.e., good nodes get high values). But that is easy to arrange with judicious placement of minus signs. It is also important that g be nonnegative. If this is not true, then paths that traverse cycles in the graph will appear to get better as they get longer.

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor. Then the next step begins.

This process can be summarized as follows.

Algorithm: Best-First Search

1. Start with *OPEN* containing just the initial state.
2. Until a goal is found or there are no nodes left on *OPEN* do:
   (a) Pick the best node on *OPEN*.
   (b) Generate its successors.
   (c) For each successor do:

Scanned by CamScanner

i. If it has not been generated before, evaluate it, add it to *OPEN*, and record its parent.

ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

The basic idea of this algorithm is simple. Unfortunately, it is rarely the case that graph traversal algorithms are simple to write correctly. And it is even rarer that it is simple to guarantee the correctness of such algorithms. In the section that follows, we describe this algorithm in more detail as an example of the design and analysis of a graph-search program.

3.3.2 The A* Algorithm

The best-first search algorithm that was just presented is a simplification of an algorithm called A*, which was first presented by Hart *et al.* [1968; 1972]. This algorithm uses the same f', g, and h' functions, as well as the lists *OPEN* and *CLOSED*, that we have already described.

Algorithm: A*

1. Start with *OPEN* containing only the initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to $h' + 0$, or h'. Set *CLOSED* to the empty list.

2. Until a goal node is found, repeat the following procedure: If there are no nodes on *OPEN*, report failure. Otherwise, pick the node on *OPEN* with the lowest f' value. Call it *BESTNODE*. Remove it from *OPEN*. Place it on *CLOSED*. See if *BESTNODE* is a goal node. If so, exit and report a solution (either *BESTNODE* if all we want is the node or the path that has been created between the initial state and *BESTNODE* if we are interested in the path). Otherwise, generate the successors of *BESTNODE* but do not set *BESTNODE* to point to them yet. (First we need to see if any of them have already been generated.) For each such *SUCCESSOR*, do the following:

   (a) Set *SUCCESSOR* to point back to *BESTNODE*. These backwards links will make it possible to recover the path once a solution is found.

   (b) Compute $g(SUCCESSOR) = g(BESTNODE)$ + the cost of getting from *BESTNODE* to *SUCCESSOR*.

   (c) See if *SUCCESSOR* is the same as any node on *OPEN* (i.e., it has already been generated but not processed). If so, call that node *OLD*. Since this node already exists in the graph, we can throw *SUCCESSOR* away and add *OLD* to the list of *BESTNODE*'s successors. Now we must decide whether *OLD*'s parent link should be reset to point to *BESTNODE*. It should be if the path we have just found to *SUCCESSOR* is cheaper than the current best path to *OLD* (since *SUCCESSOR* and *OLD* are really the same node). So see whether it is cheaper to get to *OLD* via its current parent or to *SUCCESSOR*

Scanned by CamScanner

via *BESTNODE* by comparing their g values. If *OLD* is cheaper (or just as cheap), then we need do nothing. If *SUCCESSOR* is cheaper, then reset *OLD*'s parent link to point to *BESTNODE*, record the new cheaper path in $g(OLD)$, and update $f'(OLD)$.

(d) If *SUCCESSOR* was not on *OPEN*, see if it is on *CLOSED*. If so, call the node on *CLOSED OLD* and add *OLD* to the list of *BESTNODE*'s successors. Check to see if the new path or the old path is better just as in step 2(c), and set the parent link and g and f' values appropriately. If we have just found a better path to *OLD*, we must propagate the improvement to *OLD*'s successors. This is a bit tricky. *OLD* points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still on *OPEN* or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at *OLD*, changing each node's g value (and thus also its f' value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found.[4] This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

(e) If *SUCCESSOR* was not already on either *OPEN* or *CLOSED*, then put it on *OPEN*, and add it to the list of *BESTNODE*'s successors. Compute $f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)$.

Several interesting observations can be made about this algorithm. The first concerns the role of the g function. It lets us choose which node to expand next on the basis not only of how good the node itself looks (as measured by h'), but also on the basis of how good the path to the node was. By incorporating g into f', we will not always choose as our next node to expand the node that appears to be closest to the goal. This is useful if we care about the path we find. If, on the other hand, we only care about getting to a solution somehow, we can define g always to be 0, thus always choosing the node that seems closest to a goal. If we want to find a path involving the fewest number of steps, then we set the cost of going from a node to its successor as a constant, usually 1. If, on the other hand, we want to find the cheapest path and some operators cost more than others, then we set the cost of going from one node to another to reflect those costs. Thus the A* algorithm can be used whether we are interested in finding a minimal-cost overall path or simply any path as quickly as possible.

The second observation involves h', the estimator of h, the distance of a node to the goal. If h' is a perfect estimator of h, then A* will converge immediately to the goal

[4] This second check guarantees that the algorithm will terminate even if there are cycles in the graph. If there is a cycle, then the second time that a given node is visited, the path will be no better than the first time and so propagation will stop.

Scanned by CamScanner

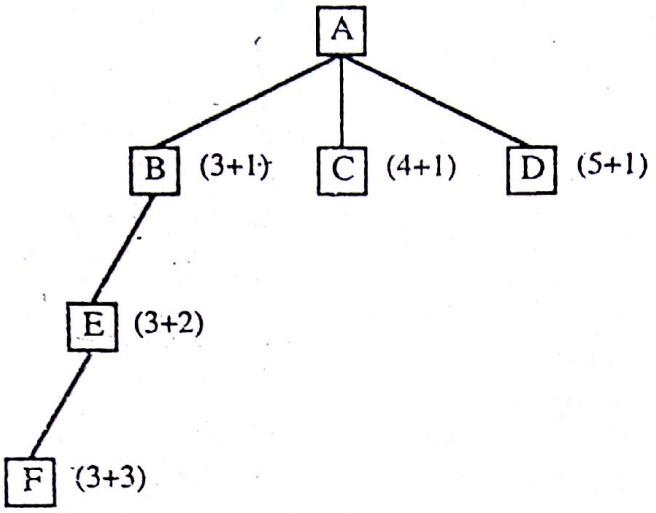


Figure 3.4: h' Underestimates h

with no search. The better h' is, the closer we will get to that direct approach. If, on the other hand, the value of h' is always 0, the search will be controlled by g. If the value of g is also 0, the search strategy will be random. If the value of g is always 1, the search will be breadth first. All nodes on one level will have lower g values, and thus lower f' values than will all nodes on the next level. What if, on the other hand, h' is neither perfect nor 0? Can we say anything interesting about the behavior of the search? The answer is yes if we can guarantee that h' never overestimates h. In that case, the A* algorithm is guaranteed to find an optimal (as determined by g) path to a goal, if one exists. This can easily be seen from a few examples.[5]

Consider the situation shown in Figure 3.4. Assume that the cost of all arcs is 1. Initially, all nodes except A are on *OPEN* (although the figure shows the situation two steps later, after B and E have been expanded). For each node, f' is indicated as the sum of h' and g. In this example, node B has the lowest f', 4, so it is expanded first. Suppose it has only one successor E, which also appears to be three moves away from a goal. Now $f'(E)$ is 5, the same as $f'(C)$. Suppose we resolve this in favor of the path we are currently following. Then we will expand E next. Suppose it too has a single successor F, also judged to be three moves from a goal. We are clearly using up moves and making no progress. But $f'(F) = 6$, which is greater than $f'(C)$. So we will expand C next. Thus we see that by underestimating $h(B)$ we have wasted some effort. But eventually we discover that B was farther away than we thought and we go back and try another path.

Now consider the situation shown in Figure 3.5. Again we expand B on the first step. On the second step we again expand E. At the next step we expand F, and finally we generate G, for a solution path of length 4. But suppose there is a direct path from D to a solution, giving a path of length 2. We will never find it. By overestimating $h'(D)$ we make D look so bad that we may find some other, worse solution without ever expanding D. In general, if h' might overestimate h, we cannot be guaranteed of finding the cheapest path solution unless we expand the entire graph until all paths are

[5] A search algorithm that is guaranteed to find an optimal path to a goal, if one exists, is called *admissible* [Nilsson, 1980].

Scanned by CamScanner

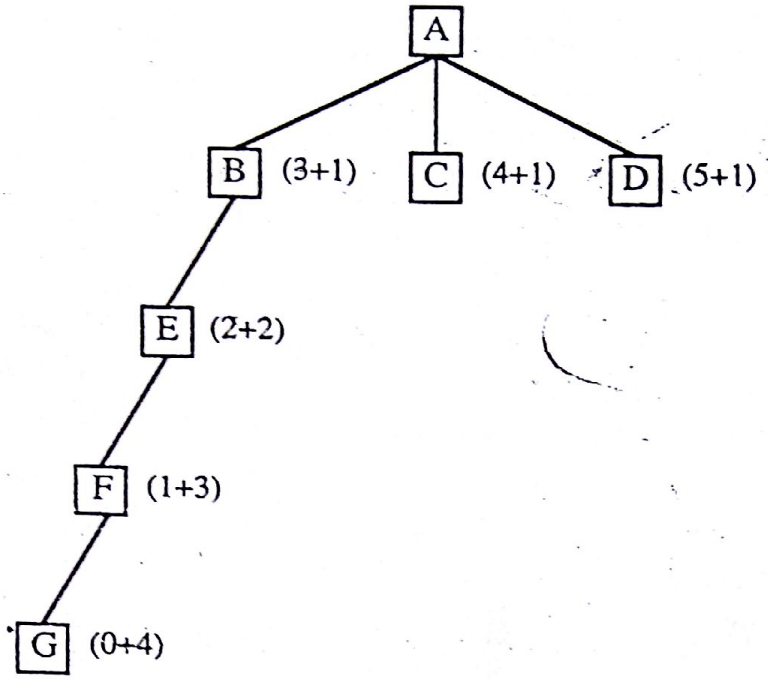


Figure 3.5: h' Overestimates h

longer than the best solution. An interesting question is, "Of what practical significance is the theorem that if h' never overestimates h then A* is admissible?" The answer is, "almost none," because, for most real problems, the only way to guarantee that h' never overestimates h is to set it to zero. But then we are back to breadth-first search, which is admissible but not efficient. But there is a corollary to this theorem that is very useful. We can state it loosely as follows:

> **Graceful Decay of Admissibility:** If h' rarely overestimates h by more than δ, then the A* algorithm will rarely find a solution whose cost is more than δ greater than the cost of the optimal solution.

The formalization and proof of this corollary will be left as an exercise.

The third observation we can make about the A* algorithm has to do with the relationship between trees and graphs. The algorithm was stated in its most general form as it applies to graphs. It can, of course, be simplified to apply to trees by not bothering to check whether a new node is already on *OPEN* or *CLOSED*. This makes it faster to generate nodes but may result in the same search being conducted many times if nodes are often duplicated.

Under certain conditions, the A* algorithm can be shown to be optimal in that it generates the fewest nodes in the process of finding a solution to a problem. Under other conditions it is not optimal. For formal discussions of these conditions, see Gelperin [1977] and Martelli [1977].

3.3.3 Agendas

In our discussion of best-first search in OR graphs, we assumed that we could evaluate multiple paths to the same node independently of each other. For example, in the water

Scanned by CamScanner

jug problem, it makes no difference to the evaluation of the merit of the position (4, 3) that there are at least two separate paths by which it could be reached. This is not true, however, in all situations, e.g., especially when there is no single, simple heuristic function that measures the distance between a given node and a goal.

Consider, for example, the task faced by the mathematics discovery program AM, written by Lenat [1977; 1982]. AM was given a small set of starting facts about number theory and a set of operators it could use to develop new ideas. These operators included such things as "Find examples of a concept you already know." AM's goal was to generate new "interesting" mathematical concepts. It succeeded in discovering such things as prime numbers and Goldbach's conjecture.

Armed solely with its basic operators, AM would have been able to create a great many new concepts, most of which would have been worthless. It needed a way to decide intelligently which rules to apply. For this it was provided with a set of heuristic rules that said such things as "The extreme cases of any concept are likely to be interesting." "Interest" was then used as the measure of merit of individual tasks that the system could perform. The system operated by selecting at each cycle the most interesting task, doing it, and possibly generating new tasks in the process. This corresponds to the selection of the most promising node in the best-first search procedure. But in AM's situation the fact that several paths recommend the same task does matter. Each contributes a reason why the task would lead to an interesting result. The more such reasons there are, the more likely it is that the task really would lead to something good. So we need a way to record proposed tasks along with the reasons they have been proposed. AM used a task agenda. An *agenda* is a list of tasks a system could perform. Associated with each task there are usually two things: a list of reasons why the task is being proposed (often called *justifications*) and a rating representing the overall weight of evidence suggesting that the task would be useful.

An agenda-driven system uses the following procedure.

Algorithm: Agenda-Driven Search

1. Do until a goal state is reached or the agenda is empty:
   (a) Choose the most promising task from the agenda. Notice that this task can be represented in any desired form. It can be thought of as an explicit statement of what to do next or simply as an indication of the next node to be expanded.
   (b) Execute the task by devoting to it the number of resources determined by its importance. The important resources to consider are time and space. Executing the task will probably generate additional tasks (successor nodes). For each of them, do the following:
      i. See if it is already on the agenda. If so, then see if this same reason for doing it is already on its list of justifications. If so, ignore this current evidence. If this justification was not already present, add it to the list. If the task was not on the agenda, insert it.
      ii. Compute the new task's rating, combining the evidence from all its justifications. Not all justifications need have equal weight. It is often useful to associate with each justification a measure of how strong a

Scanned by CamScanner

reason it is. These measures are then combined at this step to produce an overall rating for the task.

One important question that arises in agenda-driven systems is how to find the most promising task on each cycle. One way to do this is simple. Maintain the agenda sorted by rating. When a new task is created, insert it into the agenda in its proper place. When a task has its justifications changed, recompute its rating and move it to the correct place in the list. But this method causes a great deal of time to be spent keeping the agenda in perfect order. Much of this time is wasted since we do not need perfect order. We only need to know the proper first element. The following modified strategy may occasionally cause a task other than the best to be executed, but it is significantly cheaper than the perfect method. When a task is proposed, or a new justification is added to an existing task, compute the new rating and compare it against the top few (e.g., five or ten) elements on the agenda. If it is better, insert the node into its proper position at the top of the list. Otherwise, leave it where it is or simply insert it at the end of the agenda. At the beginning of each cycle, choose the first task on the agenda. In addition, once in a while, go through the agenda and reorder it properly.

An agenda-driven control structure is also useful if some tasks (or nodes) provide negative evidence about the merits of other tasks (or nodes). This can be represented by justifications with negative weightings. If these negative weightings are used, it may be important to check not only for the possibility of moving a task to the head of the agenda but also of moving a top task to the bottom if new, negative justifications appear. But this is easy to do.

As you can see, the agenda mechanism provides a good way of focusing the attention of a complex system in the areas suggested by the greatest number of positive indicators. But the overhead for each task executed may be fairly high. This raises the question of the proper grain size for the division of the entire problem-solving process into individual tasks. Suppose each task is very small. Then we will never do even a very small thing unless it really is the best thing to do. But we will spend a large percentage of our total effort on figuring out what to do next. If, on the other hand, the size of an individual task is very large, then some effort may be spent finishing one task when there are more promising ones that could be done. But a smaller percentage of the total time will be spent on the overhead of figuring out what to do. The exact choice of task size for a particular system depends on the extent to which doing one small thing really means that a set of other small things is likely to be very good to do too. It often requires some experimentation to get right.

There are some problem domains for which an agenda mechanism is inappropriate. The agenda mechanism assumes that if there is good reason to do something now, then there will also be the same good reason to do something later unless something better comes along in the interim. But this is not always the case, particularly for systems that are interacting with people. The following dialogue would not be acceptable to most people:

Person:	I don't want to read any more about China. Give me something else.
Computer:	OK. What else are you interested in?
Person:	How about Italy? I think I'd find Italy fascinating.
Computer:	What things about Italy are you interested in reading about?

Scanned by CamScanner

Person: I think I'd like to start with its history.

Computer: Why don't you want to read any more about China?

It would have been fine to have tried to find out why the person was no longer interested in China right after he or she mentioned it. The computer chose instead to try to find a new area of positive interest, also a very reasonable thing to do. But in conversations, the fact that something is reasonable now does not mean that it will continue to be so after the conversation has proceeded for a while. So it is not a good idea simply to put possible statements on an agenda, wait until a later lull, and then pop out with them. More precisely, agendas are a good way to implement monotonic production systems (in the sense of Section 2.4) and a poor way to implement nonmonotonic ones.

Despite these difficulties, agenda-driven control structures are very useful. They provide an excellent way of integrating information from a variety of sources into one program since each source simply adds tasks and justifications to the agenda. As AI programs become more complex and their knowledge bases grow, this becomes a particularly significant advantage.

3.4 Problem Reduction

So far, we have considered search strategies for OR graphs through which we want to find a single path to a goal. Such structures represent the fact that we will know how to get from a node to a goal state if we can discover how to get from that node to a goal state along any one of the branches leaving it.

3.4.1 AND-OR Graphs

Another kind of structure, the AND-OR graph (or tree), is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved. This is why the structure is called not simply an AND graph but rather an AND-OR graph. An example of an AND-OR graph (which also happens to be an AND-OR tree) is given in Figure 3.6. AND arcs are indicated with a line connecting all the components.

In order to find solutions in an AND-OR graph, we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately. This algorithm should find a path from the starting node of the graph to a set of nodes representing solution states. Notice that it may be necessary to get to more than one solution state since each arm of an AND arc must lead to its own solution node.

To see why our best-first search algorithm is not adequate for searching AND-OR graphs, consider Figure 3.7(*a*). The top node, A, has been expanded, producing two arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of f' at that node. We assume, for simplicity, that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with

Scanned by CamScanner

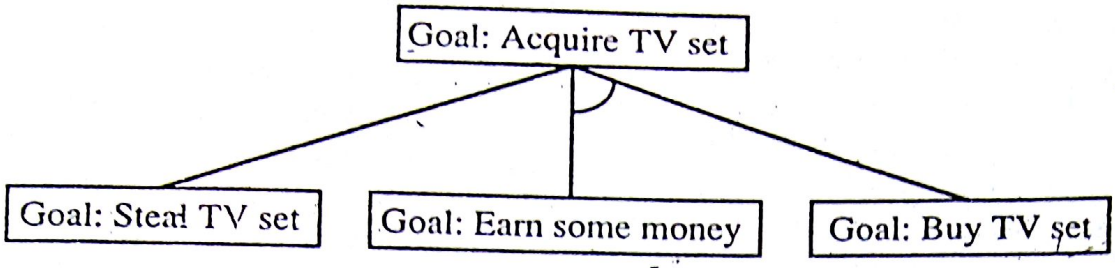
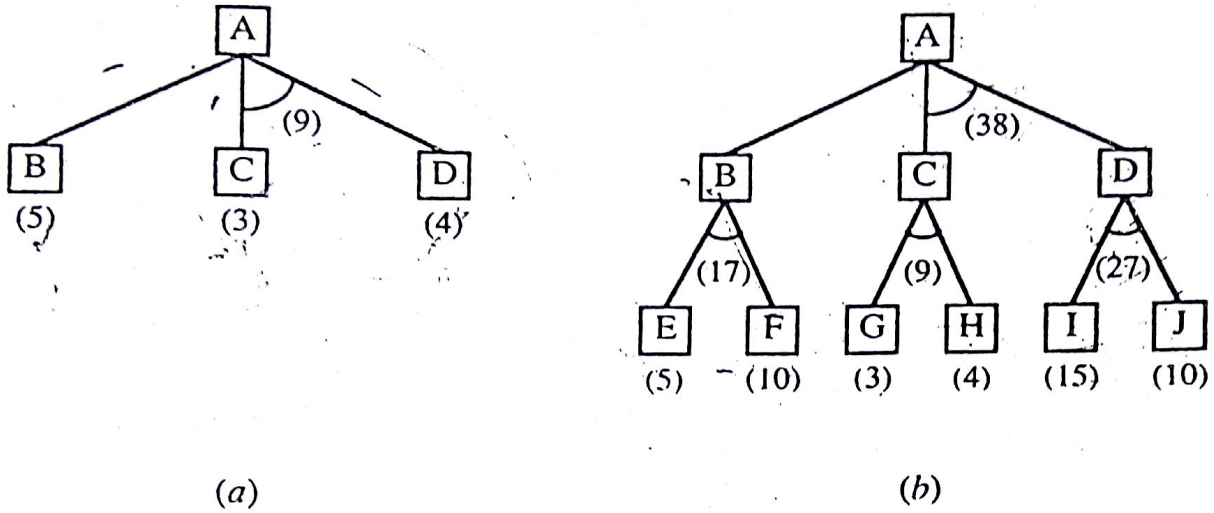


Figure 3.6: A Simple AND-OR Graph



(a) (b)

Figure 3.7: AND-OR Graphs

multiple successors has a cost of 1 for each of its components. If we look just at the nodes and choose for expansion the one with the lowest f' value, we must select C. But using the information now available, it would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C+D+2) compared to the cost of 6 that we get by going through B. The problem is that the choice of which node to expand next must depend not only on the f' value of that node but also on whether that node is part of the current best path from the initial node. The tree shown in Figure 3.7(*b*) makes this even clearer. The most promising single node is G with an f' value of 3. It is even part of the most promising arc G-H, with a total cost of 9. But that arc is not part of the current best path since to use it we must also use the arc I-J, with a cost of 27. The path from A, through B, to E and F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

In order to describe an algorithm for searching an AND-OR graph, we need to exploit a value that we call *FUTILITY*. If the estimated cost of a solution becomes greater than the value of *FUTILITY*, then we abandon the search. *FUTILITY* should be chosen to correspond to a threshold such that any solution with a cost above it is too expensive to be practical, even if it could ever be found. Now we can state the algorithm.

Scanned by CamScanner

Algorithm: Problem Reduction

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled *SOLVED* or until its cost goes above *FUTILITY*:
   (a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
   (b) Pick one of these unexpanded nodes and expand it. If there are no successors, assign *FUTILITY* as the value of this node. Otherwise, add its successors to the graph and for each of them compute f' (use only h' and ignore g, for reasons we discuss below). If f' of any node is 0, mark that node as *SOLVED*.
   (c) Change the f' estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as *SOLVED*. At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propagation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their f' values be the best estimates available.

This process is illustrated in Figure 3.8. At step 1, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C, and D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9. (Marked arcs are indicated in the figures by arrows.) In step 2, node D is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the f' value of D to 10. Going back one more level, we see that this makes the AND arc B-C better than the arc to D, so it is labeled as the current best path. At step 3, we traverse that arc from A and discover the unexpanded nodes B and C. If we are going to find a solution along this path, we will have to expand both B and C eventually, so let's choose to explore B first. This generates two new arcs, the ones to G and to H. Propagating their f' values backward, we update f' of B to 6 (since that is the best we think we can do, which we can achieve by going through G). This requires updating the cost of the AND arc B-C to 12 (6+4+2). After doing that, the arc to D is again the better path from A, so we record that as the current best path and either node E or node F will be chosen for expansion at step 4. This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

In addition to the difference discussed above, there is a second important way in which an algorithm for searching an AND-OR graph must differ from one for searching an OR graph. This difference, too, arises from the fact that individual paths from node to node cannot be considered independently of the paths through other nodes connected

Scanned by CamScanner

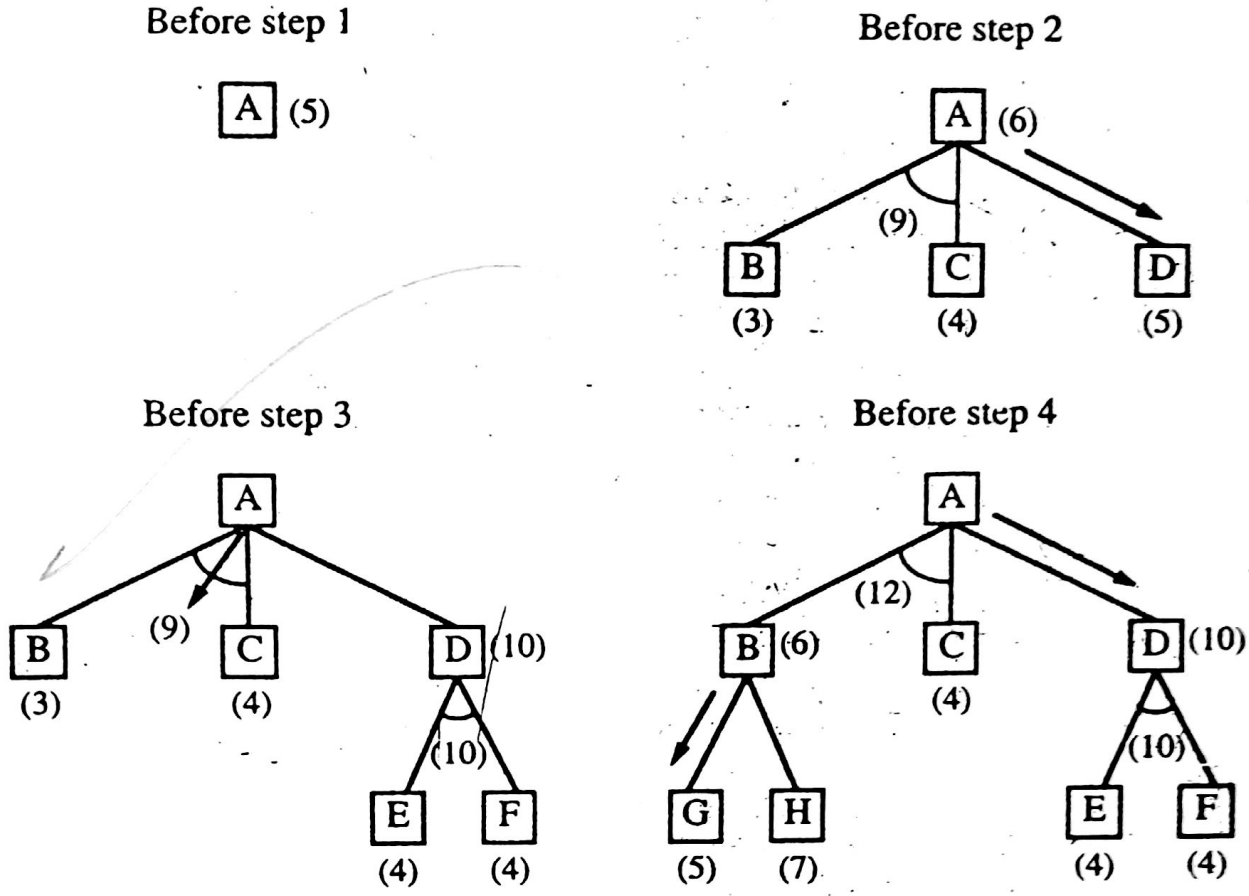


Figure 3.8: The Operation of Problem Reduction

to the original ones by AND arcs. In the best-first search algorithm, the desired path from one node to another was always the one with the lowest cost. But this is not always the case when searching an AND-OR graph.

Consider the example shown in Figure 3.9(*a*). The nodes were generated in alphabetical order. Now suppose that node J is expanded at the next step and that one of its successors is node E, producing the graph shown in Figure 3.9(*b*). This new path to E is longer than the previous path to E going through C. But since the path through C will only lead to a solution if there is also a solution to D, which we know there is not, the path through J is better.

There is one important limitation of the algorithm we have just described. It fails to take into account any interaction between subgoals. A simple example of this failure is shown in Figure 3.10. Assuming that both node C and node E ultimately lead to a solution, our algorithm will report a complete solution that includes both of them. The AND-OR graph states that for A to be solved, both C and D must be solved. But then the algorithm considers the solution of D as a completely separate process from the solution of C. Looking just at the alternatives from D, E is the best path. But it turns out that C is necessary anyway, so it would be better also to use it to satisfy D. But since our algorithm does not consider such interactions, it will find a nonoptimal path. In Chapter 13, problem-solving methods that can consider interactions among subgoals are presented.

Scanned by CamScanner

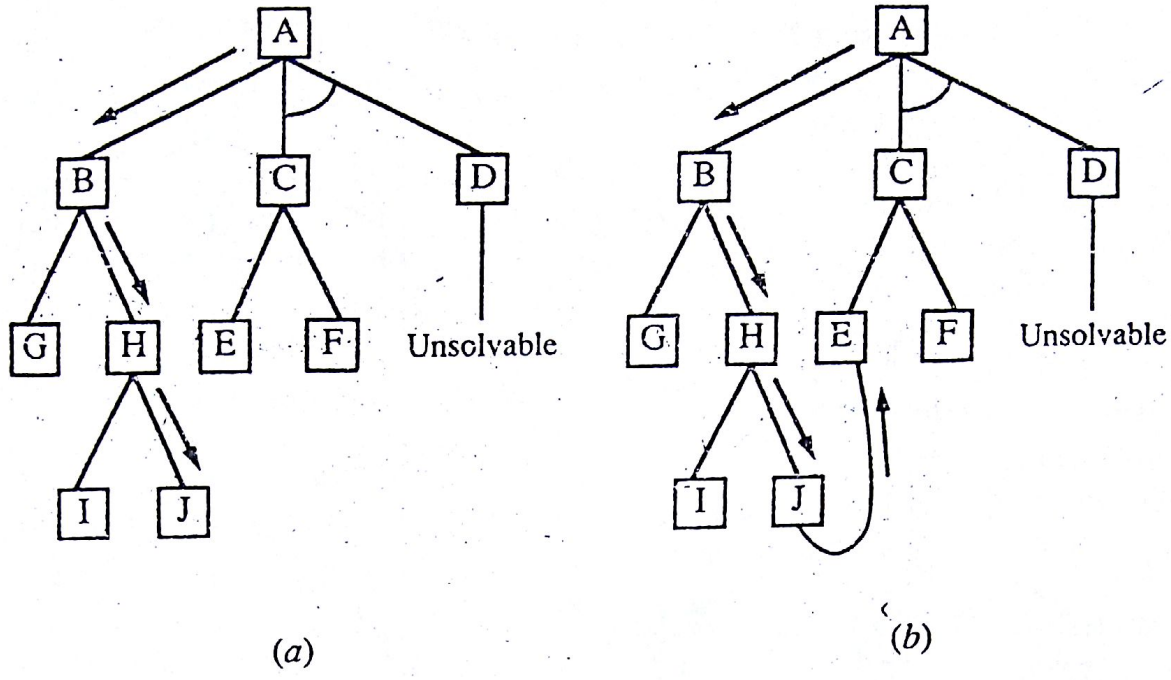


Figure 3.9: A Longer Path May Be Better

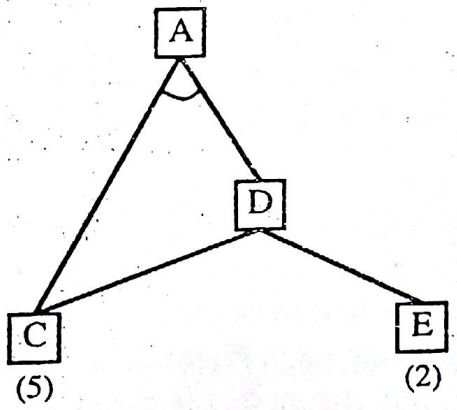


Figure 3.10: Interacting Subgoals

3.4.2 The AO* Algorithm

The problem reduction algorithm we just described is a simplification of an algorithm described in Martelli and Montanari [1973], Martelli and Montanari [1978], and Nilsson [1980]. Nilsson calls it the AO* algorithm, the name we assume.

Rather than the two lists, *OPEN* and *CLOSED*, that were used in the A* algorithm, the AO* algorithm will use a single structure *GRAPH*, representing the part of the search graph that has been explicitly generated so far. Each node in the graph will point both down to its immediate successors and up to its immediate predecessors. Each node in the graph will also have associated with it an h' value, an estimate of the cost of a path from itself to a set of solution nodes. We will not store g (the cost of getting from the start node to the current node) as we did in the A* algorithm. It is not possible to compute a single such value since there may be many paths to the same state. And such a value is not necessary because of the top-down traversing of the best-known path, which guarantees that only nodes that are on the best path will ever be considered for expansion. So h' will serve as the estimate of goodness of a node.

Scanned by CamScanner

Algorithm: AO* (h' value)

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT*.) Compute $h'(INIT)$.

2. Until *INIT* is labeled *SOLVED* or until *INIT*'s h' value becomes greater than *FUTILITY*, repeat the following procedure:

   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE*.

   (b) Generate the successors of *NODE*. If there are none, then assign *FUTILITY* as the h' value of *NODE*. This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR*) that is not also an ancestor of *NODE* do the following:

      i. Add *SUCCESSOR* to *GRAPH*.

      ii. If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an h' value of 0.

      iii. If *SUCCESSOR* is not a terminal node, compute its h' value.

   (c) Propagate the newly discovered information up the graph by doing the following: Let *S* be a set of nodes that have been labeled *SOLVED* or whose h' values have been changed and so need to have values propagated back to their parents. Initialize *S* to *NODE*. Until *S* is empty, repeat the following procedure:

      i. If possible, select from *S* a node none of whose descendants in *GRAPH* occurs in *S*. If there is no such node, select any node from *S*. Call this node *CURRENT*, and remove it from *S*.

      ii. Compute the cost of each of the arcs emerging from *CURRENT*. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT*'s new h' value the minimum of the costs just computed for the arcs emerging from it.

      iii. Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.

      iv. Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED*.

      v. If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to *S*.

It is worth noticing a couple of points about the operation of this algorithm. In step 2(*c*)*v*, the ancestors of a node whose cost was altered are added to the set of nodes whose costs must also be revised. As stated, the algorithm will insert all the node's ancestors into the set, which may result in the propagation of the cost change back up through a large number of paths that are already known not to be very good. For example, in Figure 3.11, it is clear that the path through C will always be better than the path through B, so work expended on the path through B is wasted. But if the cost of E is

Scanned by CamScanner

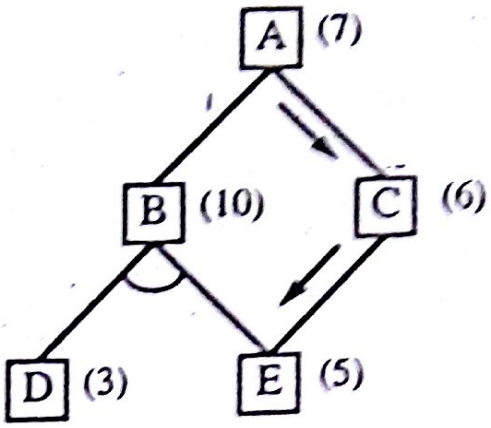


Figure 3.11: An Unnecessary Backward Propagation

revised and that change is not propagated up through B as well as through C, B may appear to be better. For example, if, as a result of expanding node E, we update its cost to 10, then the cost of C will be updated to 11. If this is all that is done, then when A is examined, the path through B will have a cost of only 11 compared to 12 for the path through C, and it will be labeled erroneously as the most promising path. In this example, the mistake might be detected at the next step, during which D will be expanded. If its cost changes and is propagated back to B, B's cost will be recomputed and the new cost of E will be used. Then the new cost of B will propagate back to A. At that point, the path through C will again be better. All that happened was that some time was wasted in expanding D. But if the node whose cost has changed is farther down in the search graph, the error may never be detected. An example of this is shown in Figure 3.12(*a*). If the cost of G is revised as shown in Figure 3.12(*b*) and if it is not immediately propagated back to E, then the change will never be recorded and a nonoptimal solution through B may be discovered.

A second point concerns the termination of the backward cost propagation of step 2(*c*). Because *GRAPH* may contain cycles, there is no guarantee that this process will terminate simply because it reaches the "top" of the graph. It turns out that the process can be guaranteed to terminate for a different reason, though. One of the exercises at the end of this chapter explores why.

Scanned by CamScanner

3.5 Constraint Satisfaction

Many problems in AI can be viewed as problems of *constraint satisfaction* in which the goal is to discover some problem state that satisfies a given set of constraints. Examples of this sort of problem include cryptarithmetic puzzles (as described in Section 2.6) and many real-world perceptual labeling problems. Design tasks can also be viewed as constraint-satisfaction problems in which a design must be created within fixed limits on time, cost, and materials.

By viewing a problem as one of constraint satisfaction, it is often possible to reduce substantially the amount of search that is required as compared with a method that attempts to form partial solutions directly by choosing specific values for components of the eventual solution. For example, a straightforward search procedure to solve a cryptarithmetic problem might operate in a state space of partial solutions in which letters

Scanned by CamScanner

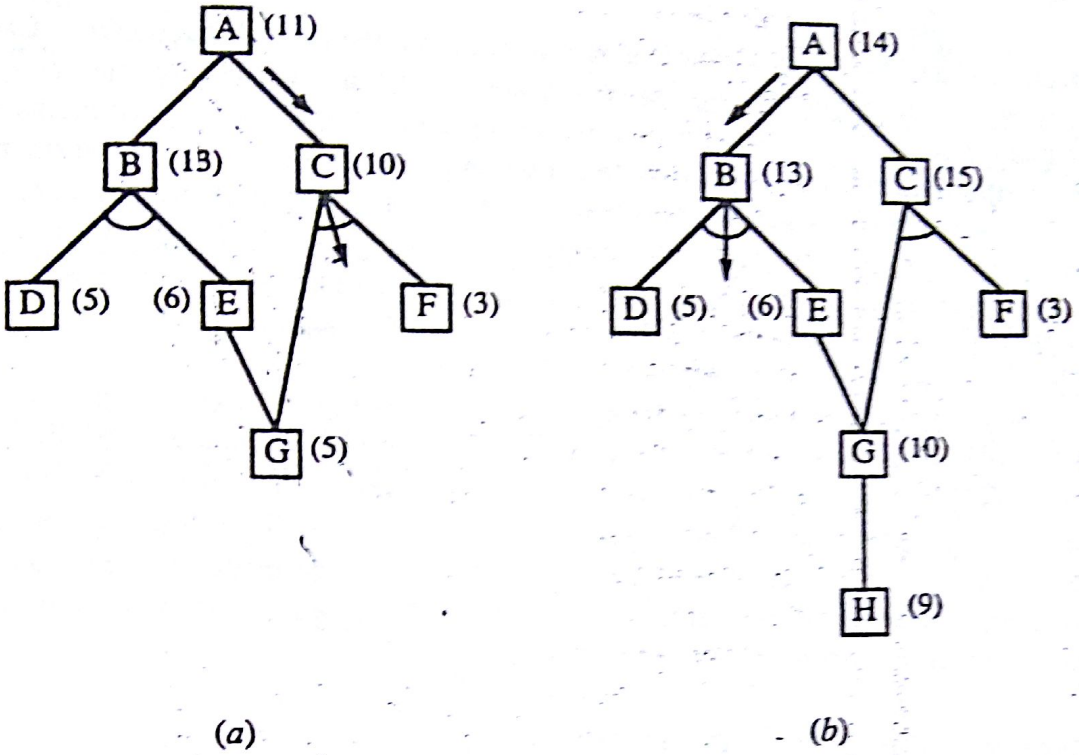


Figure 3.12: A Necessary Backward Propagation

are assigned particular numbers as their values. A depth-first control scheme could then follow a path of assignments until either a solution or an inconsistency is discovered. In contrast to this, a constraint satisfaction approach to solving this problem avoids making guesses on particular assignments of numbers to letters until it has to. Instead, the initial set of constraints, which says that each number may correspond to only one letter and that the sums of the digits must be as they are given in the problem, is first augmented to include restrictions that can be inferred from the rules of arithmetic. Then, although guessing may still be required, the number of allowable guesses is reduced and so the degree of search is curtailed.

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained "enough," where "enough" must be defined for each problem. For example, for cryptarithmetic, enough means that each letter has been assigned a unique numeric value.

Constraint satisfaction is a two-step process. First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.

The first step, propagation, arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint. So, for example, assume we start with one constraint, $N = E + 1$. Then, if we added the constraint $N = 3$, we could propagate that to get a stronger constraint on E, namely that $E = 2$. Constraint propagation also arises from the presence of inference rules

Scanned by CamScanner

that allow additional constraints to be inferred from the ones that are given. Constraint propagation terminates for one of two reasons. First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints. If the contradiction involves only those constraints that were given as part of the problem specification (as opposed to ones that were guessed during problem solving), then no solution exists. The second possible reason for termination is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If this happens and a solution has not yet been adequately specified, then search is necessary to get the process moving again.

At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In the case of the cryptarithmetic problem, for example, this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still more guesses are required, they can be made. If a contradiction is detected, then backtracking can be used to try a different guess and proceed with it. We can state this procedure more precisely as follows:

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set *OPEN* to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until *OPEN* is empty:
   (a) Select an object *OB* from *OPEN*. Strengthen as much as possible the set of constraints that apply to *OB*.
   (b) If this set is different from the set that was assigned the last time *OB* was examined or if this is the first time *OB* has been examined, then add to *OPEN* all objects that share any constraints with *OB*.
   (c) Remove *OB* from *OPEN*.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:
   (a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   (b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

This algorithm has been stated as generally as possible. To apply it in a particular problem domain requires the use of two kinds of rules: rules that define the way constraints may validly be propagated and rules that suggest guesses when guesses are

Scanned by CamScanner

Problem:

```
  SEND
+MORE
-----
MONEY
```

Initial State:

No two letters have the same value.
The sums of the digits must be as shown in the problem.

Figure 3.13: A Cryptarithmetic Problem

necessary. It is worth noting, though, that in some problem domains guessing may not be required. For example, the Waltz algorithm for propagating line labels in a picture, which is described in Chapter 14, is a version of this constraint satisfaction algorithm with the guessing step eliminated. In general, the more powerful the rules for propagating constraints, the less need there is for guessing.

To see how this algorithm works, consider the cryptarithmetic problem shown in Figure 3.13. The goal state is a problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied.

The solution process proceeds in cycles. At each cycle, two significant things are done (corresponding to steps 1 and 4 of this algorithm):

1. Constraints are propagated by using rules that correspond to the properties of arithmetic.
2. A value is guessed for some letter whose value is not yet determined.

In the first step, it does not usually matter a great deal what order the propagation is done in, since all available propagations will be performed before the step ends. In the second step, though, the order in which guesses are tried may have a substantial impact on the degree of search that is necessary. A few useful heuristics can help to select the best guess to try first. For example, if there is a letter that has only two possible values and another with six possible values, there is a better chance of guessing right on the first than on the second. Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will usually lead quickly either to a contradiction (if it is wrong) or to the generation of many additional constraints (if it is right). A guess on a less constrained letter, on the other hand, provides less information.

The result of the first few cycles of processing this example is shown in Figure 3.14. Since constraints never disappear at lower levels, only the ones being added are shown

Scanned by CamScanner

for each level. It will not be much harder for the problem solver to access the constraints as a set of lists than as one long list, and this approach is efficient both in terms of storage space and the ease of backtracking. Another reasonable approach for this problem would be to store all the constraints in one central database and also to record at each node the changes that must be undone during backtracking. C1, C2, C3, and C4 indicate the carry bits out of the columns, numbering from the right.

Initially, rules for propagating constraints generate the following additional constraints:

- M = 1, since two single-digit numbers plus a carry cannot total more than 19.
- S = 8 or 9, since S + M + C3 > 9 (to generate the carry) and M = 1, S + 1 + C3 > 9, so S + C3 > 8 and C3 is at most 1.
- O = 0, since S + M(1) + C3 (<= 1) must be at least 10 to generate a carry and it can be at most 11. But M is already 1, so O must be 0.
- N = E or E + 1, depending on the value of C2. But N cannot have the same value as E. So N = E + 1 and C2 is 1.
- In order for C2 to be 1, the sum of N + R + C1 must be greater than 9, so N + R must be greater than 8.
- N + R cannot be greater than 18, even with a carry in, so E cannot be 9.

At this point, let us assume that no more constraints can be generated. Then, to make progress from here, we must guess. Suppose E is assigned the value 2. (We chose to guess a value for E because it occurs three times and thus interacts highly with the other letters.) Now the next cycle begins.

The constraint propagator now observes that:

- N = 3, since N = E + 1.
- R = 8 or 9, since R + N (3) + C1 (1 or 0) = 2 or 12. But since N is already 3, the sum of these nonnegative numbers cannot be less than 3. Thus R + 3 + (0 or 1) = 12 and R = 8 or 9.
- 2 + D = Y or 2 + D = 10 + Y, from the sum in the rightmost column.

Again, assuming no further constraints can be generated, a guess is required. Suppose C1 is chosen to guess a value for. If we try the value 1, then we eventually reach dead ends, as shown in the figure. When this happens, the process will backtrack and try C1 = 0.

A couple of observations are worth making on this process. Notice that all that is required of the constraint propagation rules is that they not infer spurious constraints. They do not have to infer all legal ones. For example, we could have reasoned through to the result that C1 equals 0. We could have done so by observing that for C1 to be 1, the following must hold: 2 + D = 10 + Y. For this to be the case, D would have to be 8 or 9. But both S and R must be either 8 or 9 and three letters cannot share two values. So C1 cannot be 1. If we had realized this initially, some search could have been avoided. But since the constraint propagation rules we used were not that sophisticated,

Scanned by CamScanner

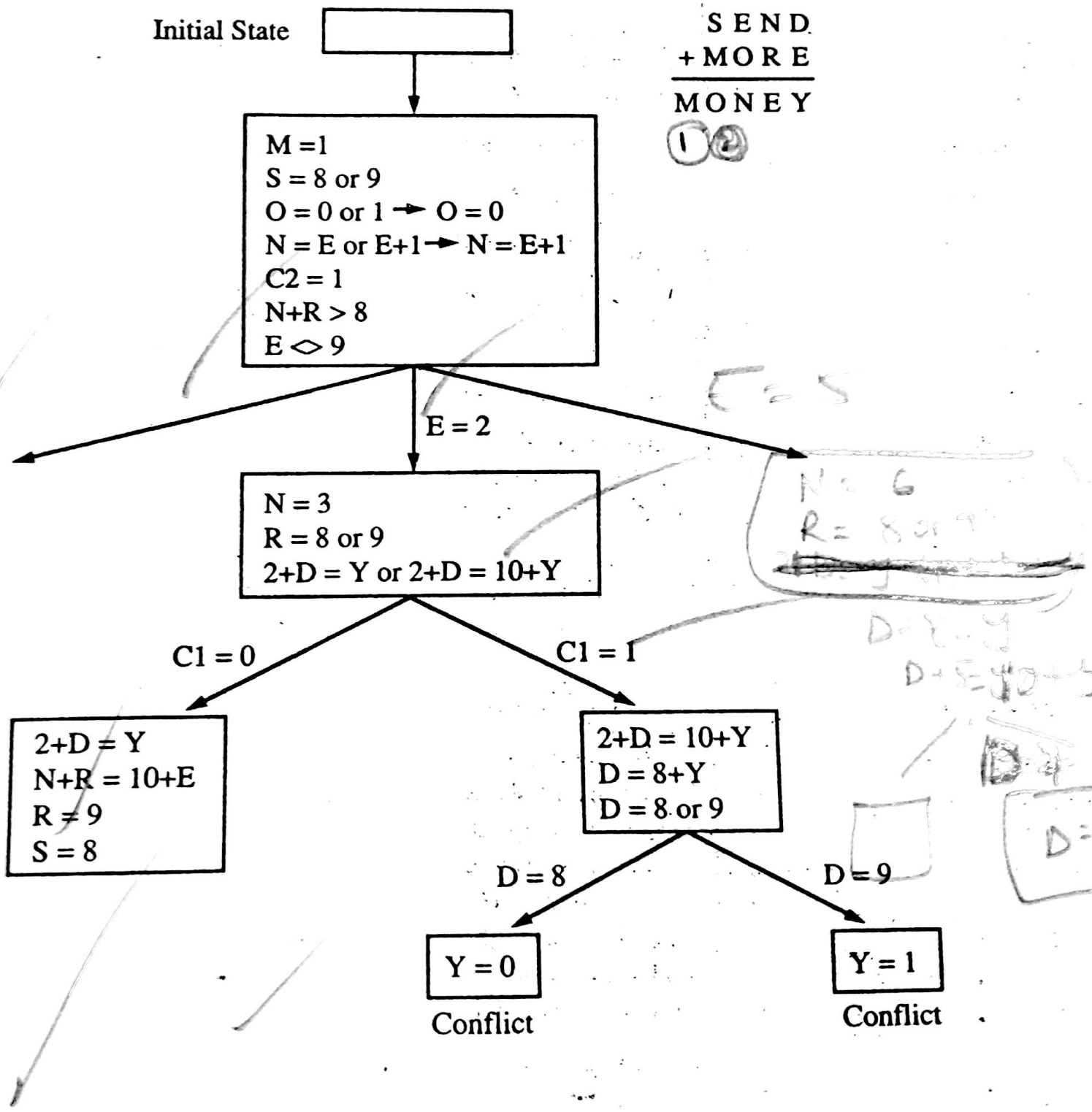


Figure 3.14: Solving a Cryptarithmetic Problem

Scanned by CamScanner

it took some search. Whether the search route takes more or less actual time than does the constraint propagation route depends on how long it takes to perform the reasoning required for constraint propagation.

A second thing to notice is that there are often two kinds of constraints. The first kind are simple; they just list possible values for a single object. The second kind are more complex; they describe relationships between or among objects. Both kinds of constraints play the same role in the constraint satisfaction process, and in the cryptarithmetic example they were treated identically. For some problems, however, it may be useful to represent the two kinds of constraints differently. The simple, value-listing constraints are always dynamic, and so must always be represented explicitly in each problem state. The more complicated, relationship-expressing constraints are dynamic in the cryptarithmetic domain since they are different for each cryptarithmetic problem. But in many other domains they are static. For example, in the Waltz line labeling algorithm, the only binary constraints arise from the nature of the physical world, in which surfaces can meet in only a fixed number of possible ways. These ways are the same for all pictures that that algorithm may see. Whenever the binary constraints are static, it may be computationally efficient not to represent them explicitly in the state description but rather to encode them in the algorithm directly. When this is done, the only things that get propagated are possible values. But the essential algorithm is the same in both cases.

So far, we have described a fairly simple algorithm for constraint satisfaction in which chronological backtracking is used when guessing leads to an inconsistent set of constraints. An alternative is to use a more sophisticated scheme in which the specific cause of the inconsistency is identified and only constraints that depend on that culprit are undone. Others, even though they may have been generated after the culprit, are left alone if they are independent of the problem and its cause. This approach is called dependency-directed backtracking (DDB). It is described in detail in Section 7.5.1.

3.6 Means-Ends Analysis

So far, we have presented a collection of search strategies that can reason either forward or backward, but for a given problem, one direction or the other must be chosen. Often, however, a mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the major parts of a problem first and then go back and solve the small problems that arise in "gluing" the big pieces together. A technique known as *means-ends analysis* allows us to do that.

The means-ends analysis process centers around the detection of differences between the current state and the goal state. Once such a difference is isolated, an operator that can reduce the difference must be found. But perhaps that operator cannot be applied to the current state. So we set up a subproblem of getting to a state in which it can be applied. The kind of backward chaining in which operators are selected and then subgoals are set up to establish the preconditions of the operators is called *operator subgoaling*. But maybe the operator does not produce exactly the goal state we want. Then we have a second subproblem of getting from the state it does produce to the goal. But if the difference was chosen correctly and if the operator is really effective at reducing the difference, then the two subproblems should be easier to solve than the

Scanned by CamScanner

Operator	*Preconditions*	*Results*
PUSH(obj, loc)	at(robot, obj) ∧ large(obj) ∧ clear(obj) ∧ armempty	at(obj, loc) ∧ at(robot, loc)
CARRY(obj, loc)	at(robot, obj) ∧ small(obj)	at(obj, loc) ∧ at(robot, loc)
WALK(loc)	none	at(robot, loc)
PICKUP(obj)	at(robot, obj)	holding(obj)
PUTDOWN(obj)	holding(obj)	¬ holding(obj)
PLACE(obj1, obj2)	at(robot, obj2) ∧ holding(obj1)	on(obj1, obj2)

Figure 3.15: The Robot's Operators

original problem. The means-ends analysis process can then be applied recursively. In order to focus the system's attention on the big problems first, the differences can be assigned priority levels. Differences of higher priority can then be considered before lower priority ones.

The first AI program to exploit means-ends analysis was the General Problem Solver (GPS) [Newell and Simon, 1963; Ernst and Newell, 1969]. Its design was motivated by the observation that people often use this technique when they solve problems. But GPS provides a good example of the fuzziness of the boundary between building programs that simulate what people do and building programs that simply solve a problem any way they can.

Just like the other problem-solving techniques we have discussed, means-ends analysis relies on a set of rules that can transform one problem state into another. These rules are usually not represented with complete state descriptions on each side. Instead, they are represented as a left side that describes the conditions that must be met for the rule to be applicable (these conditions are called the rule's *preconditions*) and a right side that describes those aspects of the problem state that will be changed by the application of the rule. A separate data structure called a *difference table* indexes the rules by the differences that they can be used to reduce.

Consider a simple household robot domain. The available operators are shown in Figure 3.15, along with their preconditions and results. Figure 3.16 shows the difference table that describes when each of the operators is appropriate. Notice that sometimes there may be more than one operator that can reduce a given difference and that a given operator may be able to reduce more than one difference.

Suppose that the robot in this domain were given the problem of moving a desk with two things on it from one room to another. The objects on top must also be moved. The

Scanned by CamScanner

	Push	Carry	Walk	Pickup	Putdown	Place
Move object	*	*				
Move robot			*			
Clear object				*		
Get object on object						*
Get arm empty					*	*
Be holding object				*		

Figure 3.16: A Difference Table

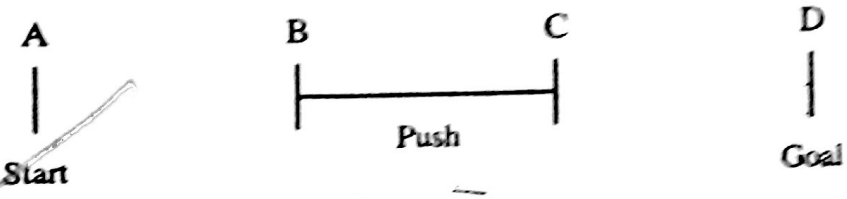


Figure 3.17: The Progress of the Means-Ends Analysis Method

main difference between the start state and the goal state would be the location of the desk. To reduce this difference, either PUSH or CARRY could be chosen. If CARRY is chosen first, its preconditions must be met. This results in two more differences that must be reduced: the location of the robot and the size of the desk. The location of the robot can be handled by applying WALK, but there are no operators than can change the size of an object (since we did not include SAW-APART). So this path leads to a dead-end. Following the other branch, we attempt to apply PUSH. Figure 3.17 shows the problem solver's progress at this point. It has found a way of doing something useful. But it is not yet in a position to do that thing. And the thing does not get it quite to the goal state. So now the differences between A and B and between C and D must be reduced.

PUSH has four preconditions, two of which produce differences between the start and the goal states: the robot must be at the desk, and the desk must be clear. Since the desk is already large, and the robot's arm is empty, those two preconditions can be ignored. The robot can be brought to the correct location by using WALK. And the surface of the desk can be cleared by two uses of PICKUP. But after one PICKUP, an attempt to do the second results in another difference—the arm must be empty. PUTDOWN can be used to reduce that difference.

Once PUSH is performed, the problem state is close to the goal state, but not quite. The objects must be placed back on the desk. PLACE will put them there. But it cannot be applied immediately. Another difference must be eliminated, since the robot must be holding the objects. The progress of the problem solver at this point is shown in Figure 3.18.

The final difference between C and E can be reduced by using WALK to get the robot back to the objects, followed by PICKUP and CARRY.

The process we have just illustrated (which we call MEA for short) can be summarized as follows:

Scanned by CamScanner

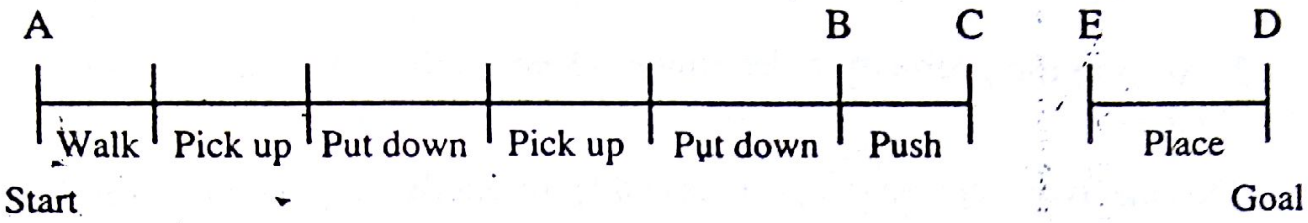


Figure 3.18: More Progress of the Means-Ends Method

Algorithm: Means-Ends Analysis (*CURRENT*, *GOAL*)

1. Compare *CURRENT* to *GOAL*. If there are no differences between them then return.
2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:
   (a) Select an as yet untried operator *O* that is applicable to the current difference. If there are no such operators, then signal failure.
   (b) Attempt to apply *O* to *CURRENT*. Generate descriptions of two states: *O-START*, a state in which *O*'s preconditions are satisfied and *O-RESULT*, the state that would result if *O* were applied in *O-START*.
   (c) If
   (*FIRST-PART* ← MEA(*CURRENT*, *O-START*))
   and
   (*LAST-PART* ← MEA(*O-RESULT*, *GOAL*))
   are successful, then signal success and return the result of concatenating *FIRST-PART*, *O*, and *LAST-PART*.

Many of the details of this process have been omitted in this discussion. In particular, the order in which differences are considered can be critical. It is important that significant differences be reduced before less critical ones. If this is not done, a great deal of effort may be wasted on situations that take care of themselves once the main parts of the problem are solved.

The simple process we have described is usually not adequate for solving complex problems. The number of permutations of differences may get too large. Working on one difference may interfere with the plan for reducing another. And in complex worlds, the required difference tables would be immense. In Chapter 13 we look at some ways in which the basic means-ends analysis approach can be extended to tackle some of these problems.

Scanned by CamScanner

3.8 Exercises

1. When would best-first search be worse than simple breadth-first search?

2. Suppose we have a problem that we intend to solve using a heuristic best-first search procedure. We need to decide whether to implement it as a tree search or as a graph search. Suppose that we know that, on the average, each distinct node will be generated N times during the search process. We also know that if we use a graph, it will take, on the average, the same amount of time to check a node to see if it has already been generated as it takes to process M nodes if no checking is done. How can we decide whether to use a tree or a graph? In addition to the parameters N and M, what other assumptions must be made?

3. Consider trying to solve the 8-puzzle using hill climbing. Can you find a heuristic function that makes this work? Make sure it works on the following example:

Start

1	2	3
8	5	6
4	7	

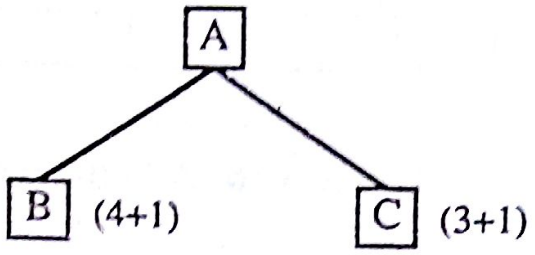
Goal

1	2	3
4	5	6
7	8	

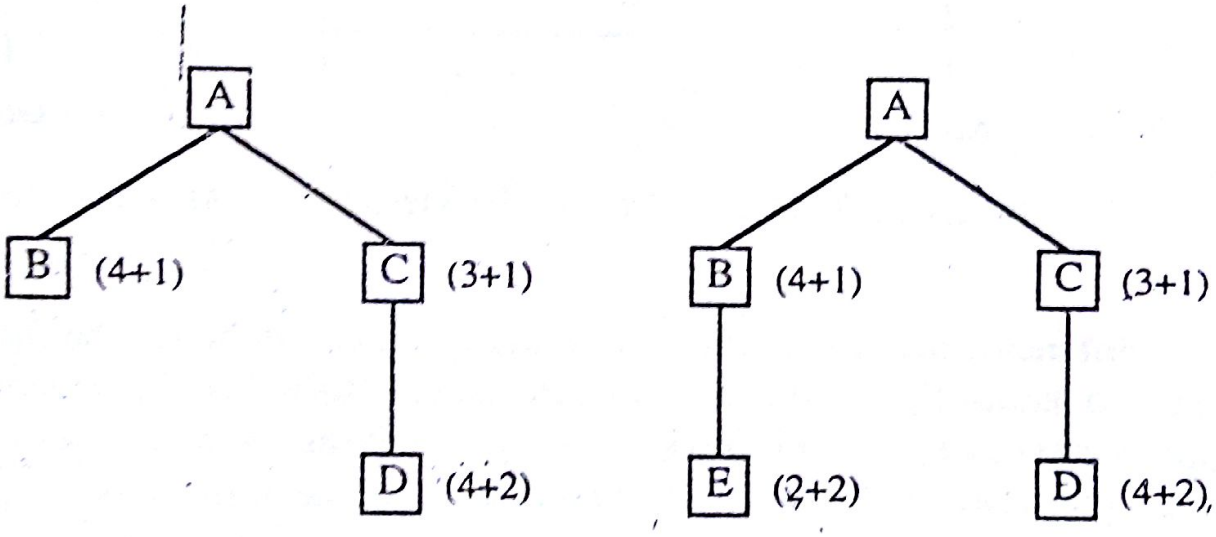
Scanned by CamScanner

4. Describe the behavior of a revised version of the steepest ascent hill climbing algorithm in which step 2(*c*) is replaced by "set current state to best successor."

5. Suppose that the first step of the operation of the best-first search algorithm results in the following situation ($a + b$ means that the value of h' at a node is a and the value of g is b):



The second and third steps then result in the following sequence of situations:

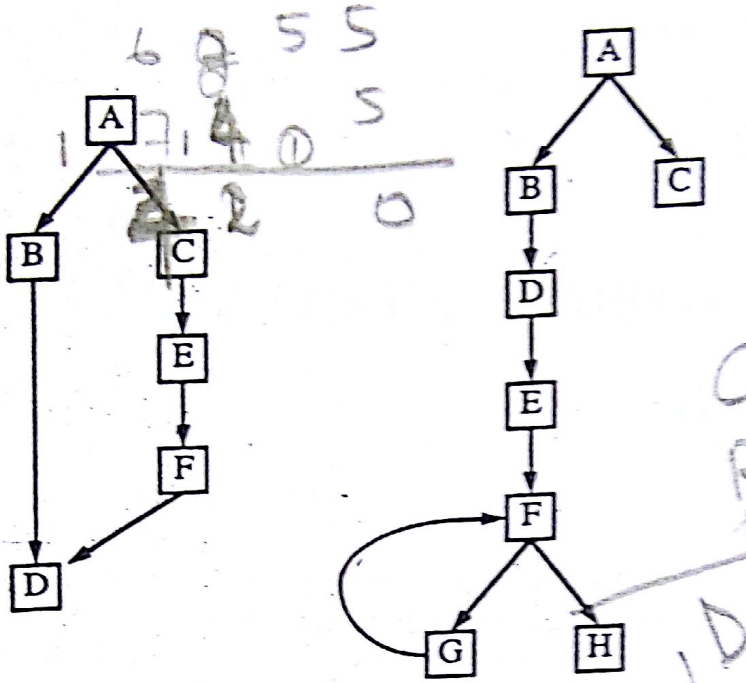


(a) What node will be expanded at the next step?

(b) Can we guarantee that the best solution will be found?

6. Why must the A* algorithm work properly on graphs containing cycles? Cycles could be prevented if when a new path is generated to an existing node, that path were simply thrown away if it is no better than the existing recorded one. If g is nonnegative, a cyclic path can never be better than the same path with the cycle omitted. For example, consider the first graph shown below, in which the nodes were generated in alphabetical order. The fact that node D is a successor of node F could simply not be recorded since the path through node F is longer than the one through node B. This same reasoning would also prevent us from recording node E as a successor of node F, if such was the case. But what would happen in the situation shown in the second graph below if the path from node G to node F were not recorded and, at the next step, it were discovered that node G

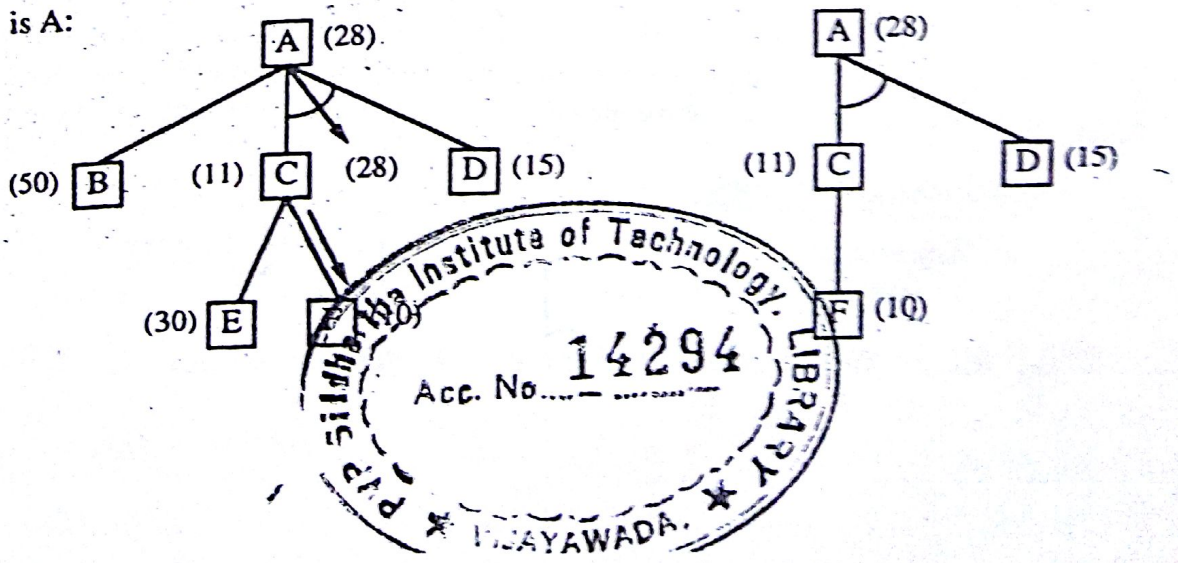
is a successor of node C?



7. Formalize the Graceful Decay of Admissibility Corollary and prove that it is true of the A* algorithm.

8. In step 2(*a*) of the AO* algorithm, a random state at the end of the current best path is chosen for expansion. But there are heuristics that can be used to influence this choice. For example, it may make sense to choose the state whose current cost estimate is the lowest. The argument for this is that for such nodes, only a few steps are required before either a solution is found or a revised cost estimate is produced. With nodes whose current cost estimate is large, on the other hand, many steps may be required before any new information is obtained. How would the algorithm have to be changed to implement this state-selection heuristic?

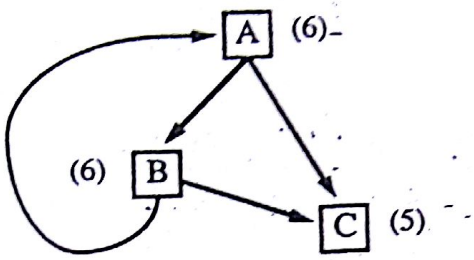
9. The backward cost propagation step 2(*c*) of the AO* algorithm must be guaranteed to terminate even on graphs containing cycles. How can we guarantee that it does? To help answer this question, consider what happens for the following two graphs, assuming in each case that node F is expanded next and that its only successor is A:



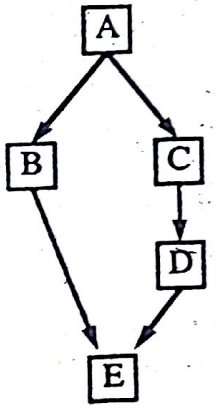
Acc. No. 14294
LIBRARY
VIJAYAWADA

Scanned by CamScanner

Also consider what happens in the following graph if the cost of node C is changed to 3:



10. The AO* algorithm, in step 2(*c*)*i*, requires that a node with no descendants in *S* be selected from *S*, if possible. How should the manipulation of *S* be implemented so that such a node can be chosen efficiently? Make sure that your technique works correctly on the following graph, if the cost of node E is changed:



11. Consider again the AO* algorithm. Under what circumstances will it happen that there are nodes in *S* but there are no nodes in *S* that have no descendants also in *S*?

12. Trace the constraint satisfaction procedure solving the following cryptarithmetic problem:

```
  CROSS
+ROADS
------
DANGER
```

13. The constraint satisfaction procedure we have described performs depth-first search whenever some kind of search is necessary. But depth-first is not the only way to conduct such a search (although it is perhaps the simplest).

    (a) Rewrite the constraint satisfaction procedure to use breadth-first search.

    (b) Rewrite the constraint satisfaction procedure to use best-first search.

14. Show how means-ends analysis could be used to solve the problem of getting from one place to another. Assume that the available operators are walk, drive, take the bus, take a cab, and fly.

Scanned by CamScanner