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.

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

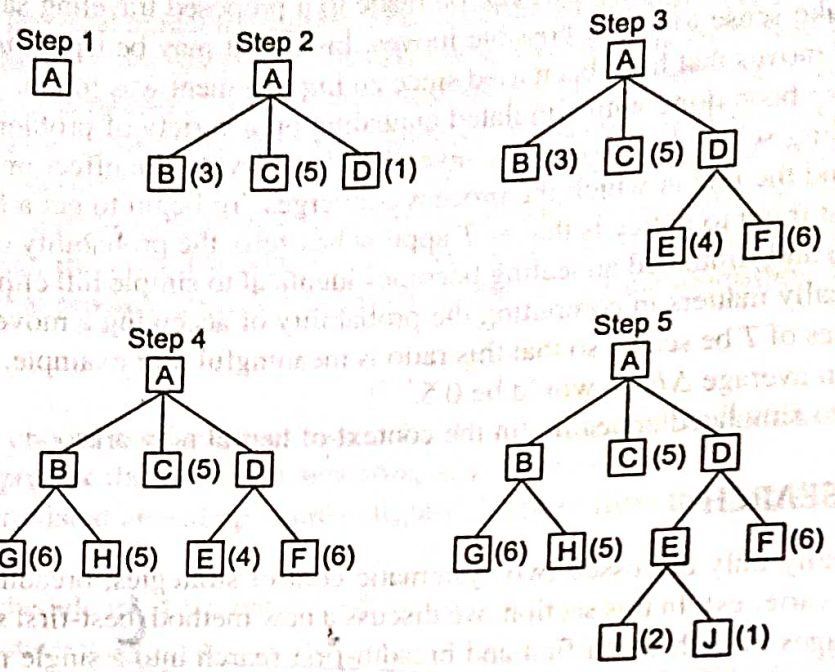

Fig. 3.3 *A Best-First Search*

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 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 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

[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.

function/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:
      (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* 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

[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.

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 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 Fig. 3.4. Assume that the cost of all arcs is 1. Initially, all nodes except A are on *OPEN* (although the Fig. 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.

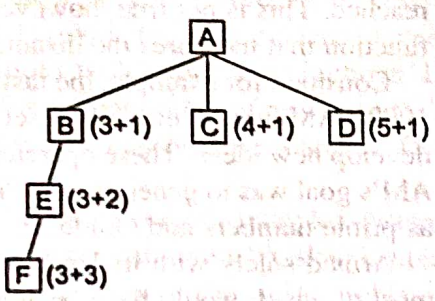

Fig. 3.4 *h' Underestimates h*

Now consider the situation shown in Fig. 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 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:

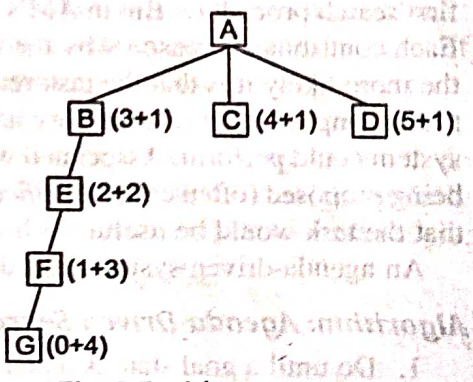

Fig. 3.5 *h' Overestimates h*

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

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

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