Chapter 2

Basic Heuristic-Search Procedures

In this chapter we describe the basic search strategies used in problem solving with special attention to the way in which they incorporate heuristic information. In Chapter 1, we saw that many types of problems are conveniently described as tasks of finding some properties of graphs. Our current paradigm therefore is to find efficient methods of unraveling some properties of graphs using information from sources outside the graph.

Our discussion will be made easier by compiling some basic nomenclature regarding graphs and graph searching, although some of the concepts were already used in Chapter 1.

Basic Graph-Searching Notation

A graph consists of a set of **nodes** (or **vertices**), which in our context represent encodings of subproblems. Every graph we will consider will have a unique node s called the **start node**, representing the initial problem at hand. Certain pairs of nodes are connected by directed **arcs** (or **links**), which represent operators available to the problem solver. If an arc is directed from node n to node n', node n' is said to be a **successor** (or a **child** or an **offspring**) of n and node n is said to be a **parent** (or a **father**) of n'. The number of successors emanating from a given node is called the **branching degree** (or **degree**) of that node and normally is denoted as b. We assume that the branching degrees of all nodes in our graphs are finite, defining a class known as **locally finite graphs**. A pair of nodes may be successors of each other as in the 8-Puzzle, in which case the two arcs can be replaced by an undirected **edge**.

A **tree** is a graph in which each node (except one **root** node) has only one parent. A node in a tree that has no successors is called a **leaf**, a **tip** node, or a terminal node. A **uniform tree** of depth N is a tree in which every node at depth smaller than N has the same branching degree whereas all nodes at depth N are leaves. Often the arcs are assigned **weights** representing either

costs or **rewards** associated with their inclusion in the final solution. We use the notation $c(n, n')$ to denote the cost of the arc from n to n'.

A sequence of nodes $n_1, n_2, \ldots, n_k$, where each n_i is a successor of n_{i-1}, is called a **path** of length k from node n_1 to node n_k. If a path exists from n_1 to n_k, node n_k is said to be a **descendant** of (or **accessible** from) n_1, and node n_1 is called an **ancestor** of n_k. The cost of a path is normally understood to be the sum of the costs of all the arcs along the path (called the **sum cost**). In special cases other cost measures can be defined, such as the maximum of all the arc costs along the path, or the average cost.

The most elementary step of graph searching that we consider is **node generation**, that is, computing the representation code of a node from that of its parent. The new successor is then said to be **generated** and its parent is said to be **explored**. A coarser computational step of great importance is **node expansion**, which consists of generating *all* successors of a given parent node. The parent is then said to be *expanded*. **Pointers** are normally set up from each generated successor back to its parent node. These pointers form a dynamic network of paths (usually a tree) which facilitates tracing a path from some given node n back to the start node or to some other desired ancestor of n. To distinguish between a pointer-traced path and other paths that may go through a given node, we adopt the following notation: a node n is said to be *along* a **pointer-path** PP at a given time if at that time at least one pointer assigned to n by the search procedure is directed along PP. A node n is said to lie *on* a path P, if the implicit problem graph contains a path P going through n.

A **search procedure,** a **policy,** or a **strategy** is a prescription for determining the order in which nodes are to be generated. We distinguish between a **blind,** or **uninformed,** search and an **informed, guided,** or **directed** search. In the former, the order in which nodes are expanded depends only on information gathered by the search but is unaffected by the character of the unexplored portion of the graph, not even by the goal criterion. The latter uses partial information about the problem domain and about the nature of the goal to help guide the search toward the more promising directions.

Naturally the set of nodes in the graph being searched can at any given time be divided into four disjoint subsets:

1. Nodes that have been expanded,
2. Nodes that have been explored but not yet expanded,
3. Nodes that have been generated but not yet explored,
4. Nodes that are still not generated

Several of the search procedures discussed require a distinction between nodes of the first and third group. Nodes that were expanded (i.e., their successors are available to the search procedure) are called **closed,** whereas nodes that were generated and are awaiting expansion are called **open.** Two separate lists called CLOSED and OPEN are used to keep track of these two sets of nodes.

2.1 HILL-CLIMBING: AN IRREVOCABLE STRATEGY

Hill-climbing, a strategy based on local optimizations, is the simplest and most popular search strategy among human problem solvers. It is called hill-climbing because, like a climber who wishes to reach the mountain peak quickly, it chooses the direction of steepest ascent from its current position. Imagine the task of adjusting the knobs of a television set to obtain a picture of acceptable quality. This task is normally performed by adjusting the knobs one at a time, setting each knob in a position that seems to accomplish the greatest progress toward the desired quality before manipulating the next knob. The procedure is based on continuously monitoring the picture quality against a desired standard and making incremental steps of improvement in a direction that, on the basis of the local information available, seems most promising. The reason this strategy is so popular among human operators is that it requires almost no computational effort; at each setting of the knobs we can begin the adjustment afresh, requiring no memory of past attempts nor of the path that has led us to the current setting.

In terms of our graph-search model, the hill-climbing strategy amounts to repeatedly expanding a node, inspecting its newly generated successors, and choosing and expanding the best among these successors, while retaining no further reference to the father or siblings. Obviously the computational simplicity of this strategy is not without shortcomings. Unless the evaluation function used is extremely informative, we stand a very high chance of violating the first tenet of systematic search, that is, do not leave any stone unturned. Deceptively improving evaluation functions may lure the search into deep, or even infinite, exploration paths that in fact contain no solution. Moreover, as soon as we come to a local maximum (a node more valuable than any of its successors), no further improvement is possible by local perturbations and the process must terminate without reaching a solution. The only way to free ourselves from such a deadlock is to start afresh from what we hope is a completely new node, thus risking violations of the second tenet of systematic search: do not turn any stone more than once. This strategy is called **irrevocable**, because the process does not permit us to shift attention back to previously suspended alternatives, even though they may have offered a greater promise than the alternatives at hand.

Hill-climbing is a useful strategy when we possess a highly informative guiding function to keep us away from local maxima, ridges, and plateaus and to lead us quickly toward the global peak. Hill-climbing is also useful in problems where the state transition operators possess a certain type of independence called **commutativity** (Nilsson, 1980), a condition in which the application of no operator can hinder the future applicability of other operators or alter the state resulting from a sequence of other applicable operators. For example, the operations of expanding nodes in a graph (i.e., generating their successors) are commutative; the expansion of any one node does not hinder the expansion of other nodes nor alter their descendants. When commutativity holds, irrevocable strategies can be applied without the risk of missing the solution. Applying an inappropriate operator may delay, but never prevent, the eventual discovery of the solution. For that reason, problems that are manageable by a set of commutative operators (e.g., the minimum spanning tree problem) are also blessed with simple optimization algorithms called "greedy." However, the task of fitting such a set of operators to a given problem may be difficult, as is discussed in Chapter 4.

2.2 UNINFORMED SYSTEMATIC SEARCH: TENTATIVE CONTROL STRATEGIES

In contrast to the irrevocable control mechanism employed by hill-climbing, we now discuss a **tentative** scheme to bring our search strategy in line with the requirements of systematization. If for some reason a given search avenue is chosen for exploration, the other candidate alternatives are not discarded but are kept in reserve in case the avenue chosen fails to produce an adequate solution. Naturally this scheme requires more memory space, but it can save time because the memory stores partially developed alternatives which, if discarded, may need to be developed afresh. Many search strategies use this tentative scheme of decision.

The class of strategies described in this section are **uninformed** or "**blind**" in the sense that the order in which the search progresses does not depend on the nature of the solution we seek. In terms of our graph model we usually say that a search strategy is uninformed when the location of the goal node (or nodes) does not alter the order of node expansion, except of course for the termination conditions. Being uninformed, these strategies are rather inefficient and usually are impractical in large problems. They are worth describing, though, as possible standards against which the performances of various informed, heuristically guided, strategies can be compared.

2.2.1 Depth-First and Backtracking: LIFO Search Strategies

In **depth-first** search, as well as in the popular variation called **backtracking**, priority is given to nodes at deeper levels of the search graph. The finest computational step in depth-first search is node expansion, that is, each node chosen for exploration gets all its successors generated before another node is explored.

After each node expansion, one of the newly generated children is again selected for expansion and this forward exploration is pursued until, for some reason, progress is blocked. If blocking occurs, the process resumes from the

deepest of all nodes left behind, namely, from the nearest decision point with unexplored alternatives. This policy works well when solutions are plentiful and equally desirable, or when we have reliable early warning signals to indicate an incorrect candidate direction.

In searching trees the concept of depth is well defined, and a depth-first algorithm should have no difficulty deciding which node in OPEN is the deepest: the deepest node is the one most recently generated. Therefore, if OPEN is structured as a **stack**, a depth-first strategy should put new successors on top of OPEN and select for expansion the topmost node. This last-in-first-out policy guarantees that no node at depth d will be expanded as long as nodes of depths greater than d still reside on OPEN.

It is easy to see that such a policy, if run unchecked, might be dangerous when implemented on large graphs, especially those of infinite depth. It could continue to probe deeper and deeper along some fruitless path and, not having a mechanism to recover from disappointments, would be unable to backup and try a fresh search avenue. For that reason, depth-first algorithms are usually equipped with a **depth-bound**—a stopping rule that, when triggered, returns the algorithm's attention to the deepest alternative not exceeding this bound. Thus the program backtracks under one of two conditions:

1. The depth-bound is exceeded.
2. A node is recognized as a dead end.

The latter event occurs when a node fails to pass a test for some property that must hold true for any node on a path to a solution. For example, requiring that any node explored in the 8-Queens problem should leave at least one unattacked cell in every unfilled row is a legitimate dead-end test.

To summarize, a depth-first policy employs the following sequence of steps:

Depth-First Search

1. Put the start node on OPEN.
2. If OPEN is empty, exit with failure; otherwise continue.
3. Remove the topmost node from OPEN and put it on CLOSED. Call this node n.
4. If the depth of n is equal to the depth bound, clean up CLOSED and go to step 2; otherwise continue.
5. Expand n, generating all of its successors. Put these successors (in no particular order) on top of OPEN and provide for each a pointer back to n.
6. If any of these successors is a goal node, exit with the solution obtained by tracing back through its pointers; otherwise continue.
7. If any of these successors is a dead end, remove it from OPEN and clean up CLOSED.
8. Go to step 2.

The operation "clean up CLOSED" in steps 4 and 7 is performed by purging from CLOSED all those ancestors of the nodes passing the tests in steps 4 and 7 that do not have sons in OPEN. This operation is optional and is designed only to save memory space.

Note that in step 5 placing the successors in an arbitrary order on top of OPEN renders the algorithm totally *uninformed*. An alternative method would be to use heuristics to order these successors, as was discussed in Section 1.1.1. This **partially informed** version of depth-first search will be analyzed in Chapter 6 (Section 6.4).

Figure 2.1 illustrates the sequence of steps taken by a depth-first search of the 4-Queens problem (a 4×4 board version of the 8-Queens problem). Each step is represented by the node being expanded (marked $a, b, \ldots, j, k$) and the status of the explicit portion of the search graph after each expansion. The order of nodes on OPEN can be seen by traversing the leaf nodes from left to right, skipping the dotted lines which represent portions of the graph deleted from memory. The order of node expansion is further illustrated in Figure 2.2.

Note that at any given time the CLOSED list forms a *single* path from the start node to the currently expanded node. This feature reflects the storage economy of depth-first policies; the maximum storage required cannot exceed the product of the depth-bound and the branching degree. The path of CLOSED nodes maintained by the program appears to be traversing the tree,

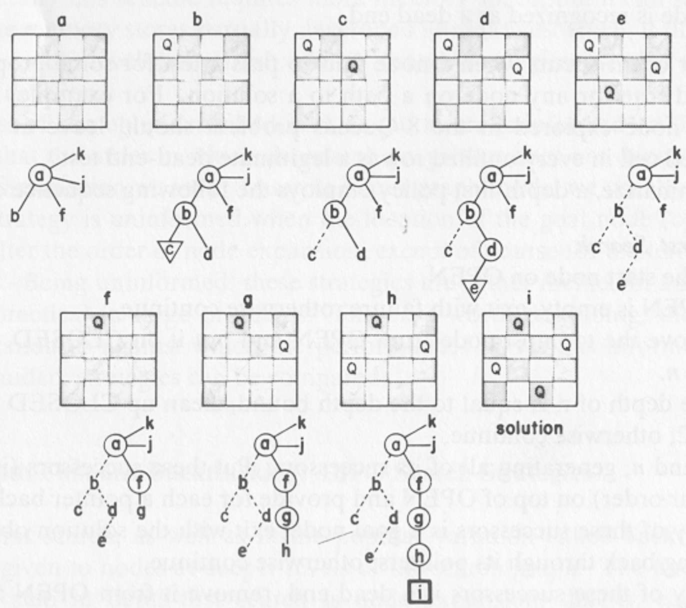

Figure 2.1

Successive steps in a depth-first search of the 4-Queens problem. Circled symbols represent CLOSED nodes, uncircled symbols represent nodes in OPEN, triangles stand for dead ends and boxed nodes are those that satisfy the goal conditions.

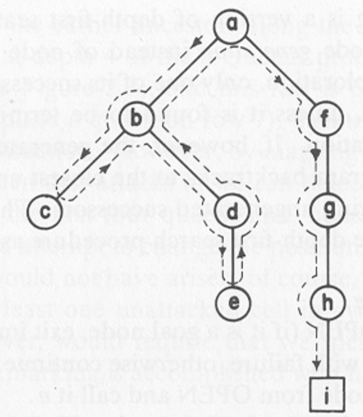

Figure 2.2

Order of node expansion by a depth-first search of the 4-Queens problem.

sweeping across it, from left to right (see Figure 2.2) and is called the **traversal path**.

When implementing depth-first policies on graphs that are not trees, we encounter certain complications. Strictly speaking, the concept of **depth** for a node in a graph is defined recursively as *one plus the depth of its shallowest parent*, with the start node having zero depth. Thus to adhere to a strict depth-first rule the program must first inspect all parents of newly generated nodes, including those parents that were recognized as dead ends and deleted from memory. Such inspection would introduce enormous complications and significantly increase storage requirements. Therefore, in searching graphs we normally compromise the strict depth-first rule (i.e., that no node will be expanded while deeper nodes reside on OPEN) in favor of the last-in-first-out principle, which is much easier to maintain. Thus we still follow the eight-step procedure just outlined, especially step 3, regardless of whether the topmost node is also the deepest node in the strict sense of the word. In step 4, however, where comparison to the depth-bound is required, the depth of a given node is taken as *one plus the depth of the parent designated by its current pointer*. This definition prevents the traversal path from exceeding the depth-bound.

There is still need, however, to check for **node duplications** in order to avoid unnecessary expansions. For example, in the 8-Puzzle, which is an undirected graph, it is important to prevent the search from exploring a repetitive back-and-forth movement of any given tile. This can be done by checking whether any of the newly generated successors already resides on the traversal path and, if a match is found, deleting the matched successor from OPEN. Again, for the sake of maintaining both simplicity and small storage, depth-first strategies normally check for a match only among those nodes residing on the traversal path, allowing duplication of nodes that were generated in the past but since deleted from memory.

Backtracking. Backtracking is a version of depth-first search that applies the last-in-first-out policy to node *generation* instead of node *expansion*. When a node is first selected for exploration, only one of its successors is generated and this newly generated node, unless it is found to be terminal or dead end, is again submitted for exploration. If, however, the generated node meets some stopping criterion, the program backtracks to the closest unexpanded ancestor, that is, an ancestor still having ungenerated successors. This policy can be implemented by modifying the depth-first search procedure as follows:

The Backtracking Procedure

1. Put the start node on OPEN (if it is a goal node, exit immediately).
2. If OPEN is empty, exit with failure, otherwise continue.
3. Examine the topmost node from OPEN and call it n.
4. If the depth of n is equal to the depth-bound or if all the branches emanating from n have already been traversed, remove n from OPEN and go to step 2; otherwise continue.
5. Generate a new successor of n (along a branch not previously traversed), call it n'. Put n' on top of OPEN and provide a pointer back to n.
6. Mark n to indicate that the branch (n, n') has been traversed.
7. If n' is a goal node, exit with the solution obtained by tracing back through its pointers; otherwise continue.
8. If n' is a dead end, remove it from OPEN.
9. Go to step 2.

OPEN is the only list used in backtracking and this time it serves to store the nodes on the traversal path. The marking of nodes in step 6 is usually done by appending to n an encoding of the transformation rules that have been applied to n in the past. For example, in the 4-Queens puzzle of Figure 2.1, when node c is generated, we should mark node b to indicate that the branch (b, c) has already been traversed. Later on, when node e is found to be a dead end, we will return to b and use this mark to avoid regenerating c.

The main advantage of backtracking over depth-first search lies in achieving an even greater storage economy. Instead of retaining in storage all the successors of an expanded node, we only store a single successor at any given time. In addition, backtracking avoids generating nodes that stand to the right of the solution path found, such as nodes k and j in Figure 2.1. These advantages are marred by the algorithm's inability to use heuristic information for evaluating candidate successors as in the informed version of depth-first search. An informed version of backtracking would have to either temporarily generate all successors of an explored node, or alternatively, to score the available operators on the basis of their own features, without explicating their resultant states.

More sophisticated backtracking strategies also use a technique called **backmarking** with which, after meeting a dead-end condition, they may back up *several levels at once* (Gaschnig, 1979a). This is done by submitting the dead-end condition to a critical analysis to see if the "blame" for that condition can be placed on one of the earlier ancestors along the traversal path. For example, if a given node at depth 4 of the 8-Queens problem leaves no free cell on the seventh row (see Figure 2.3), and this fact is only discovered at depth 7 (i.e., when trying to place a queen on row 7), backtracking to level 6 or even 5 will not remedy the situation. However, by marking each cell in row 7 with the age of the oldest queen that attacks it, we can detect that the blame really lies with the position of the first four queens and, hence, that we must backtrack directly to level 4 and attempt to change the position of the queen on the fourth row. This situation would not have arisen, of course, were we to insist that each new queen leave at least one unattacked cell in every unfilled row. Meeting this condition, however, would require that we spend extra work on forward testing, whereas backmarking is accomplished with no extra tests.

Backtracking strategies are also useful in **optimization** and semi-optimization problems. If the graph contains many solution paths and our task is to find one with the lowest cost, we can maintain a record of the cheapest solution encountered so far and continue the search until it becomes certain that no cheaper solution may lie ahead of us. For example, if we know that path costs cannot decrease through extension of the path, then finding that the cost of an early portion of a candidate path already exceeds the record we maintain allows us to *prune* that path prior to completion. This pruning is illustrated in Figure 2.4, which describes a backtracking strategy for finding the minimum-sum column in a matrix with nonnegative entries. The search tree, in this case, resembles a curtain where every nonterminal node except the start node is of degree 1. Po-

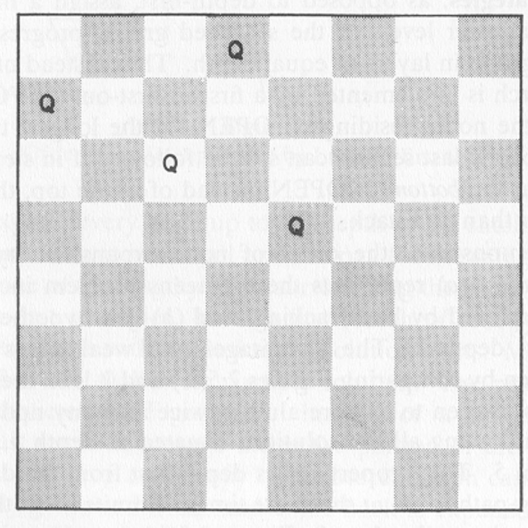

Figure 2.3
A dead-end position in the 8-Queens problem which may not be detected until we try to place a queen on the 7th row.

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6

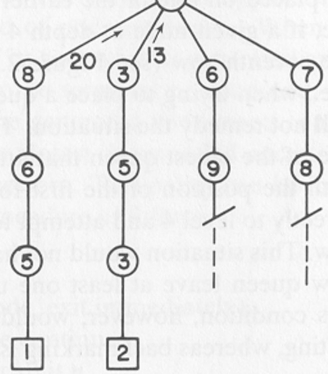

Figure 2.4
Pruning in a backtracking search for the minimum-sum column in a matrix with nonnegative entries.

tential solution paths are pruned as soon as their partial sums exceed the lowest sum encountered on the left. The importance of heuristics for ordering the first group of successors is clearly shown in this example; the amount of pruning increases when we succeed in finding cheap solutions earlier in the search.

2.2.2 Breadth-First: A FIFO Search Strategy

Breadth-first strategies, as opposed to depth-first, assign a higher priority to nodes at the shallower levels of the searched graph, progressively exploring sections of that graph in layers of equal depth. Thus instead of a LIFO policy, breadth-first search is implemented by a first-in-first-out (FIFO) policy, giving first priority to the nodes residing on OPEN for the longest time. The eight-step procedure of the last section can still be followed if in step 5 we place the new successors at the *bottom* of OPEN instead of at the top, thus using OPEN as a queue rather than as a stack.

Figure 2.5 demonstrates the order of node expansions by a breadth-first search on two trees: (*a*) represents the 4-Queens problem including the symmetric half unexplored by backtracking, and (*b*) is a hypothetical tree with a solution node at depth 2. The advantages and weaknesses of breadth-first search can be seen by comparing Figures 2.5(*a*) and 2.1. In the 4-Queens problem, breadth-first is seen to explore almost twice as many nodes as depth-first. This results from having all the solutions situated at depth 5, and no node at depth larger than 5. This property frees depth-first from the danger of following hopeless long paths and, at the same time, commits breadth-first to explore every node of depth smaller than 5. Figure 2.5(*b*), by contrast, represents a situation advantageous to breadth-first search. A solution exists at depth 2 surrounded by paths that may go to a much greater depth. Breadth-first is, again,

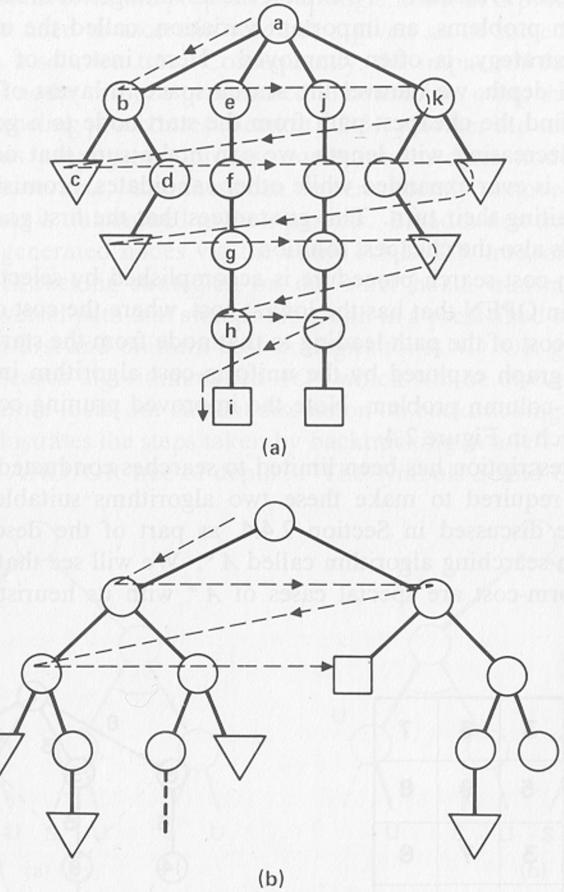

Figure 2.5
Order of node expansion by breadth-first search.

committed to explore every node up to the depth of the shallowest goal, but in this case this commitment is a small price to pay for the safeguard against getting trapped in one of the lengthy paths surrounding the goal.

Unlike depth-first strategies, breadth-first search of locally finite graphs is guaranteed to terminate with a solution if a solution exists. Moreover, it is also guaranteed to find the shallowest possible solution. The price paid for these guarantees is depicted in Figure 2.5. Instead of a single traversal path, breadth-first must retain in storage the entire portion of the graph that it explores. Only by retaining a full copy of the search graph can it shift attention away from newly generated nodes and come back to expand nodes suspended many steps earlier. The need for this abrupt shift of attention, combined with the large storage requirements, is the main reason that breadth-first strategies are rarely adopted by human problem solvers.

The Uniform-Cost Procedure. To exploit the advantages of breadth-first search in optimization problems, an important variation, called the **uniform-cost** or **cheapest-first strategy,** is often employed. Here, instead of progressing in layers of equal depth, we unravel the search space in layers of equal cost. If our task is to find the cheapest path from the start node to a goal and if path costs are nondecreasing with length, we can make sure that no node of cost greater than C is ever expanded while other candidates, promising costs lower than C, are waiting their turn. This guarantees that the first goal node chosen for expansion is also the cheapest solution.

The uniform-cost search procedure is accomplished by selecting for expansion the node in OPEN that has the lowest cost, where the cost of each node is defined as the cost of the path leading to that node from the start node. Figure 2.6 shows the graph explored by the uniform-cost algorithm in searching the minimum-sum-column problem. Note the improved pruning compared to the depth-first search in Figure 2.4.

So far our description has been limited to searches conducted on trees. The modifications required to make these two algorithms suitable for handling graphs will be discussed in Section 2.4.4. as part of the description of the heuristic graph-searching algorithm called A^*. We will see that both breadth-first and uniform-cost are special cases of A^* with its heuristic information suppressed.

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6

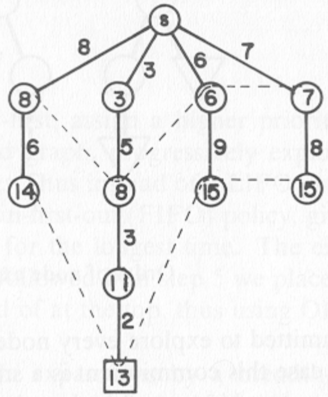

Figure 2.6
Uniform-cost search for the minimum-sum column in the matrix on the left.

2.2.3 Uninformed Search of AND/OR Graphs

Breadth-first and depth-first strategies are easily adaptable to searching AND/OR graphs. The difference lies mainly in the procedures employed for determining the termination conditions. Whereas in state-space searches the termination test involves the property of a single node, in a problem-reduction formulation we need to ascertain whether a given *set* of solved nodes collectively supports a solution tree. Consequently, the impact of each newly generated group of nodes must be propagated to determine, using the "solved" and "unsolvable" labeling procedures (Section 1.2.3), whether the start node can be labeled. In this propagation process some intermediate nodes are labeled "solved" or "unsolvable," and these can be permanently removed from storage after their impact is transmitted to their parents. Breadth-first transmits the impact of newly generated nodes via the entire AND/OR tree, or graph, that it maintains. Backtracking strategies, on the other hand, transmit that impact through the traversal path that sweeps the graph in a prescribed order.

Both breadth-first and uniform-cost strategies for AND/OR graphs are special cases of a search algorithm called AO^* which will be discussed in Section 2.4.4. We therefore focus our current discussion on backtracking strategies.

Figure 2.7 illustrates the steps taken by backtracking in a left-to-right traversal of a binary AND/OR tree of depth 3. The symbols S and U stand for the

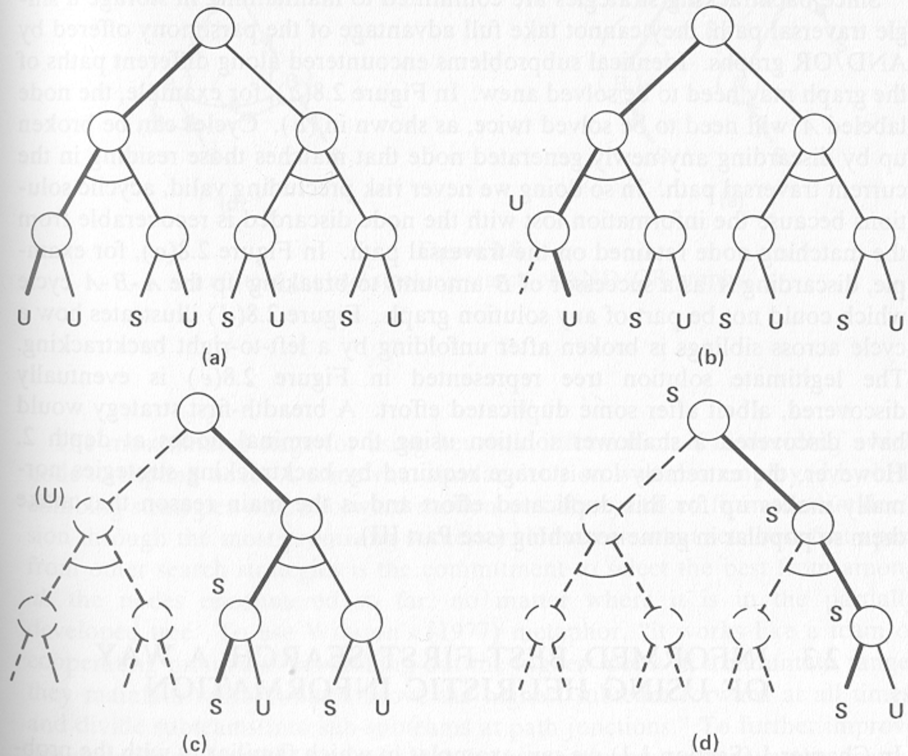

Figure 2.7
Typical steps in the execution of backtracking search of an AND/OR tree. The heavy line represents the traversal path, whereas the broken lines represent portions of the tree that can be purged from memory.

"solve" and "unsolvable" labels that propagate up the traversal path. Note that labeling any intermediate node permits us to remove that node from storage and shift the traversal path one step to the right. This removal, however, although not affecting the correct labeling of the start node, prevents us from reconstructing the actual solution graph once s is labeled as solved. Therefore, if in addition to testing whether a problem can be solved we also wish to exhibit the final solution tree, additional bookkeeping is needed. Prior to its removal, each "solved" node should transmit to its father a code for the solution graph residing below it. These codes are then combined by the various AND nodes and passed upward until the start node, if labeled "solved," receives a code for the entire solution graph beneath it. For example, in Figure 2.7(c) the intermediate node labeled S will transmit to its father the code (1, 1), indicating that a solution can be obtained by going down two levels following the leftmost successors. At stage (d) the AND node assembles the tree code (1, 1) (2, 1), and finally the start node receives the solution tree 2, ((1, 1) (2, 1)).

Since backtracking strategies are committed to maintaining in storage a single traversal path, they cannot take full advantage of the parsimony offered by AND/OR graphs. Identical subproblems encountered along different paths of the graph may need to be solved anew. In Figure 2.8(a), for example, the node labeled A will need to be solved twice, as shown in (b). Cycles can be broken up by discarding any newly generated node that matches those residing in the current traversal path. In so doing we never risk precluding valid, acyclic solutions because the information lost with the node discarded is recoverable from the matching node retained on the traversal path. In Figure 2.8(c), for example, discarding A as a successor of B amounts to breaking up the A-B-A cycle which could not be part of any solution graph. Figure 2.8(d) illustrates how a cycle across siblings is broken after unfolding by a left-to-right backtracking. The legitimate solution tree represented in Figure 2.8(e) is eventually discovered, albeit after some duplicated effort. A breadth-first strategy would have discovered a shallower solution using the terminal nodes at depth 2. However, the extremely low storage required by backtracking strategies normally makes up for this duplicated effort and is the main reason that make them so popular in game-searching (see Part III).

2.3 INFORMED, BEST-FIRST SEARCH: A WAY OF USING HEURISTIC INFORMATION

In Chapter 1 (Section 1.1) we saw examples in which familiarity with the problem domain enabled us to judge certain directions of search more promising than others, thus using knowledge beyond that which is built into the state and operator definitions. In this section, we see how this heuristic knowledge can be used in the formalism of systematic search.

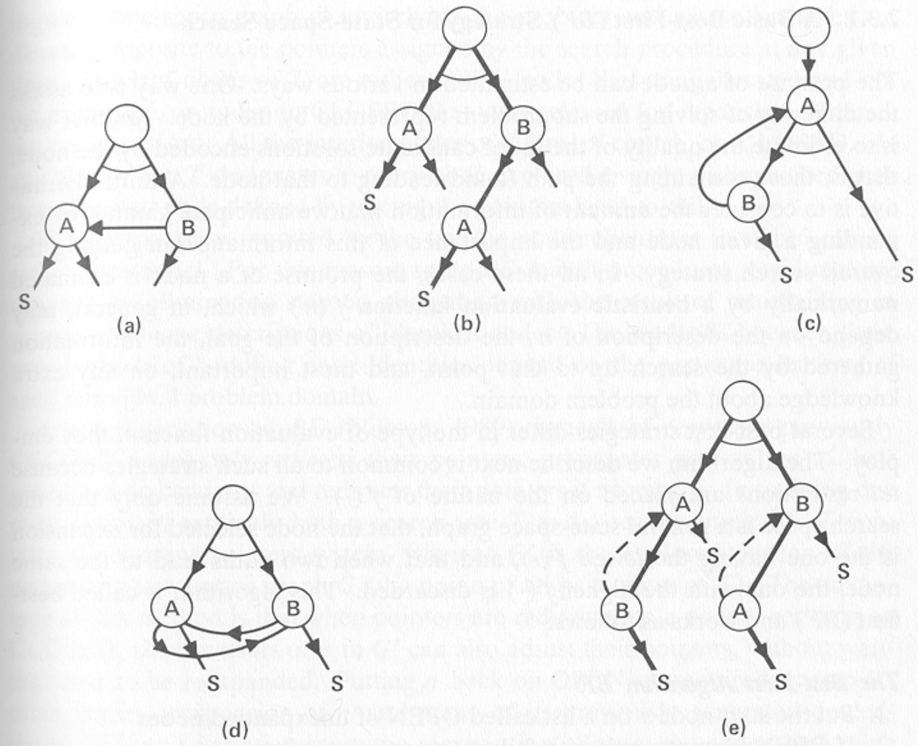

Figure 2.8
Left-to-right backtracking search of AND/OR graphs.

The most natural stage for using heuristic information is in deciding which node to expand next. A somewhat similar decision was also employed by hill-climbing strategies where forward motion was always taken from the last decision through the most promising successor. However, what sets best-first apart from other search strategies is the commitment to select the best from among *all* the nodes encountered so far, no matter where it is in the partially developed tree. To use Winston's (1977) metaphor, "it works like a team of cooperating mountaineers seeking out the highest point in a mountain range: they maintain radio contact, move the highest subteam forward at all times, and divide subteams into sub-subteams at path junctions." To further improve this metaphor we should also add that the mountaineers employ various instruments to estimate the current altitude of the various subteams and the altitude of the paths ahead, and that the readings on these instruments are occasionally corrupted by noise. The noise represents the fact that heuristic information, by its very nature, may occasionally be misleading.

2.3.1 A Basic Best-First (*BF*) Strategy for State-Space Search

The promise of a node can be estimated in various ways. One way is to assess the difficulty of solving the subproblem represented by the node. Another way is to estimate the quality of the set of candidate solutions encoded by the node, that is, those containing the path found leading to that node. A third alternative is to consider the amount of information that we anticipate gaining by expanding a given node and the importance of this information in guiding the overall search strategy. In all these cases, the promise of a node is estimated numerically by a **heuristic evaluation function** $f(n)$ which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain.

Several best-first strategies differ in the type of evaluation function they employ. The algorithm we describe next is common to all such strategies because no restrictions are placed on the nature of $f(\cdot)$. We assume only that the search space is a general state-space graph, that the node selected for expansion is the one having the *lowest* $f(\cdot)$, and that when two paths lead to the same node, the one with the higher $f(\cdot)$ is discarded. This algorithm is called **best-first** (*BF*) and works as follows:

The Best-First Algorithm BF

1. Put the start node s on a list called OPEN of unexpanded nodes.
2. If OPEN is empty, exit with failure; no solution exists.
3. Remove from OPEN a node n at which f is minimum (break ties arbitrarily), and place it on a list called CLOSED to be used for expanded nodes.
4. Expand node n, generating all its successors with pointers back to n.
5. If any of n's successors is a goal node, exit successfully with the solution obtained by tracing the path along the pointers from the goal back to s.
6. For every successor n' of n:
   a. Calculate $f(n')$.
   b. If n' was neither on OPEN nor on CLOSED, add it to OPEN. Assign the newly computed $f(n')$ to node n'.
   c. If n' already resided on OPEN or CLOSED, compare the newly computed $f(n')$ with the value previously assigned to n'. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n' now points back to n instead of to its previous predecessor). If the matching node n' resided on CLOSED, move it back to OPEN.
7. Go to step 2.

In step 6c, the decision to avoid node repetitions amounts to maintaining an explicit tree T, called the **search tree** or **traversal tree,** which is a subtree of the implicit state-space graph G underlying the problem and whose branches are directed opposite to the pointers assigned by the search procedure at any given time. The leaf nodes of T are either OPEN nodes that are generated and not yet expanded, or nodes on CLOSED that were selected for expansion, generating no successors. All the interior nodes of T are, of course, on CLOSED. At any given time, T designates to every node explored a single distinguished path from the start node defined by tracing the pointers head-to-tail.

The computation required by the test for node repetitions in step 6 can be rather substantial. For this reason, some *BF* search procedures avoid making this test by eliminating step 6c altogether, with the result of maintaining in storage duplicate descriptions of identical nodes. The tradeoffs between these two methods of handling node identities depend on the particular features of each individual problem domain.

Another variation of *BF* (Nilsson, 1980), instead of *reopening* previously CLOSED nodes only *redirects* their pointers, propagates new values to their generated descendants and redirects their pointers if necessary. In so doing one must maintain explicitly both a search tree T and a search graph G'. T contains the current pointers system, whereas G' is the explored portion of the underlying state-space graph G (the union of all past traces of T). The advantage of this method is that when pointers are redirected to a node n residing on CLOSED, all successors of n in G' can also adjust their pointers, without waiting for n to be reexpanded. Putting n back on OPEN, as required by step 6c, often invites reexpansion of n and many of its previously generated descendants. Nilsson's variation saves the reexpansion effort at the expense of value propagation and pointer-redirecting effort which may occasionally be superfluous, since some descendants of n in G' may not require regeneration. Most of our discussions in later sections will remain valid for both variations of the *BF* procedure, since the motion of T through G remains the same in both cases.

2.3.2 A General Best-First Strategy for AND/OR Graphs (*GBF*)

The basic best-first principle, namely the commitment to explore the best among *all* available candidates, can also be applied in searching AND/OR graphs. Here, however, we need to define more carefully what we mean by "best" and what we mean by a "candidate." Like in the state-space formulation we consider candidates to be subsets of potential solutions and, since in problem-reduction formulation the objects of pursuit are solution graphs, we ought to treat *subsets of solution graphs* as candidates awaiting exploration. The AND/OR graph formalism encodes each such subset by a special data structure called a **solution base**.

Assume that at a certain phase of the search the explicated portion of the underlying AND/OR graph G is represented by the subgraph G' of G. Since we can only explore G by repeated node expansions starting from s, G' must be connected, must contain s, and will have a frontier containing all those nodes generated and not yet expanded. Every subgraph of G' that has the potential of being extended into a solution graph of G represents a subset of potential solutions and ought to be regarded as a candidate for exploration. We can enumerate this set of candidate subgraphs by simply assuming that all the frontier nodes of G' that are not already labeled "unsolvable" are in fact "solved," and then find the set of solution graphs that these frontier nodes can support. Thus a **solution base** is any subgraph G'' of G' for which the following conditions hold:

1. G'' contains the start node s.
2. If an expanded AND node n is in G'', then all n's successors are also in G''.
3. If an expanded OR node n is in G'', then exactly one successor of n is also in G''.
4. None of the nodes in G'' is currently labeled "unsolvable."

We now return to consider which among the frontier nodes of G' should be rated most deserving of the next expansion. While executing *BF* on OR graphs we maintained a one-to-one correspondence between the candidates for expansion and the candidate solution bases. At any given time the set of solution bases considered by *BF* were all the (root-to-frontier) paths contained in the traversal tree T, and each one of these paths was represented by a unique node on OPEN. This one-to-one correspondence no longer holds in searching AND/OR graphs; each solution base may contain many nodes that are candidates for expansion, and a given node may participate in several candidate solution bases. Hence the interpretation of what is meant by "best" and the translation of the best-first principle to a working procedure are more complex.

The procedure described in this section applies the best-first principle in two steps. First, it identifies the most promising solution-base graph using a **graph evaluation function** f_1. Then, it expands nodes within that graph using a **node evaluation function** f_2. These two functions, serving in two different roles, provide two different types of estimates: f_1 estimates some properties of the set of solution graphs that may emanate from a given candidate base, whereas f_2 estimates the amount of information that a given node expansion may provide regarding the alleged superiority of its hosting graph. Most works in search theory focus on the computation of f_1, whereas f_2 is usually chosen in an ad hoc manner. The principles underlying the nature of f_1 are discussed later in this section.

Once we select the two guiding functions f_1 and f_2, a best-first search algorithm *GBF* for an AND/OR graph can be stated as follows:

The GBF Algorithm

1. Put the start node s on OPEN.
2. From the explicit search graph G' constructed so far, (initially, just s), compute the most promising solution-base graph G_0, using f_1 and the heuristics $\boldsymbol{h}$ provided in step 4.
3. Using f_2, select a node n that is both on OPEN and in G_0; remove n from OPEN and place it on CLOSED.
4. Expand node n, generating all its immediate successors, and add them to OPEN and to the search graph G' with pointers back to n (merge duplicate nodes in OPEN and in G'). For each successor n', provide heuristic information $\boldsymbol{h}$ (e.g., a list of parameters) that characterizes the set of solution graphs rooted at n' (see the text that follows).
5. If any successor n' is a terminal node, then
   a. Label node n' "solved" if a goal or "unsolvable" if not.
   b. If the label of n' induces a label on any of its ancestors, label these ancestors "solved" or "unsolvable" using the labeling procedure that follows.
   c. If the start node is solved, exit with G_0 as a solution graph.
   d. If the start node is labeled "unsolvable," exit with failure.
   e. Remove from G' any nodes whose label can no longer influence the label of s.
6. Go to step 2.

The **labeling procedure** mentioned in step 5 is defined recursively and can be implemented using either depth-first (see Figure 2.7) or breadth-first policies.

The "Solve" Labeling Procedure

1. A terminal node is labeled "solved" if it is a goal node (representing a primitive subproblem); otherwise it is labeled "unsolvable" (representing a subproblem that cannot be reduced any further).
2. A nonterminal AND node is labeled "unsolvable" as soon as one of its successors is labeled "unsolvable"; it is labeled "solved" if all its successors are "solved."
3. A nonterminal OR node is labeled "solved" as soon as one of its successors is labeled "solved"; it is labeled "unsolvable" if all its successors are "unsolvable."

In step 4 of *GBF* each node n' on the frontier of G' is assumed to be characterized by information regarding the set of *solution graphs rooted at* n'. This set (possibly empty) consists of subgraphs $g(n')$ of G where each $g(n')$ satisfies:

1. n' is the root node of $g(n')$.
2. n' can be labeled "solved" by applying the labeling procedure to $g(n')$.

Of course, since n' is not yet expanded, the subgraphs $g(n')$ are not available explicitly and can only be assessed using heuristic information from outside of G. The role of this information is to assess both the difficulty of finding a solution for the subproblem represented by n' and the quality of the solution when found.

Note that the preceding procedure leaves several computations unspecified. Aside from the graph and node selection functions f_1 and f_2, we also left unspecified the heuristic information provided at step 4, without which f_1 and f_2 cannot be computed. In the next section we will see that restrictions placed on these computations define a natural taxonomy of best-first algorithms. But first it may be instructive to demonstrate the major steps of the *GBF* algorithm by illustrating the sequence of solution bases selected in a simple problem.

An Example of Applying GBF. Let us assume that we wish to find a solution graph in the (implicit) AND/OR graph G of Figure 2.9(a). Expanding s in Figure 2.9(b), we unravel three solution bases (i.e., three disjoint subsets of potential solution graphs represented by the three arcs emanating from s). To apply the best-first principle we need to use some criterion (f_1) for assessing the promise offered by these solution bases. Let us assume that we decide to pursue our search within whichever subset appears to contain the solution graph with the *smallest number of edges*. Although the problem itself is not posed as an optimization task (any solution graph will do), we can reasonably argue that the pursuit of the smallest graph automatically guides us toward an early termination.

Now, to rank our three candidates by this criterion, we need to obtain some information regarding the class of solution graphs stemming from each of the three given arcs. If the entire G graph were accessible, we could have ranked our candidates perfectly; all we would need to know is the size of the smallest

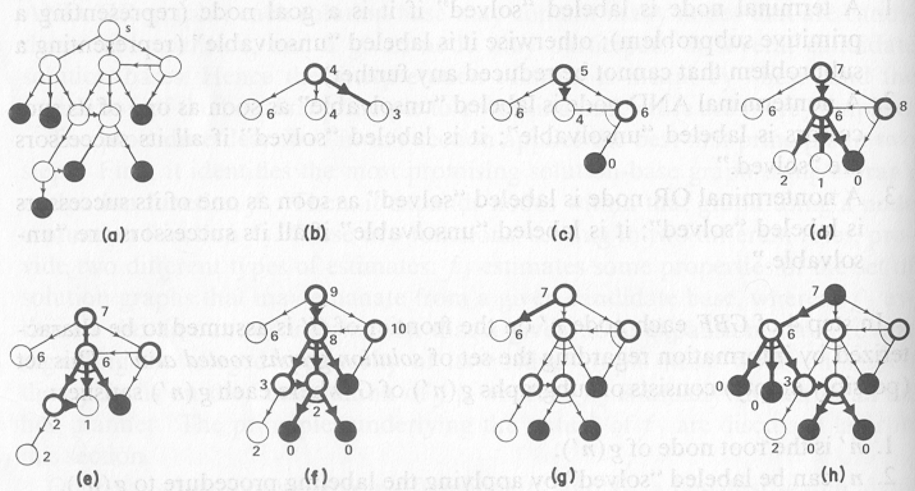

Figure 2.9

Successive steps in the execution of general-best-first (GBF) search on the implicit AND/OR graph of part (a). Solid circles represent solved nodes, heavy hollow circles nodes in CLOSED, and thin circles nodes in OPEN. The heavy lines stand, at each stage, for the current most promising solution base.

solution graph beneath each one of the three nodes currently on OPEN. Not having access to G itself, we must try to estimate the needed information from the descriptions of the nodes themselves, and from our general knowledge of the problem domain.

Assuming that such estimates were computed and gave the values 6, 4, and 3 to the three nodes on OPEN, we now make an additional assumption that these estimates be taken *at face value* for the purpose of computing the merit of each candidate solution base. Recalling that perfect estimates, if available, should be increased by one to give the required size assessment of our candidates, we apply the identical procedure to the imperfect estimates and obtain the values 7, 5, and 4. The best candidate appears to be the rightmost, with an overall estimate of 4, and this estimate can now be stored at s for later use.

This one simple step applied to the three nodes in Figure 2.9 involves two fundamental principles in heuristic programming. The choice of an auxiliary minimization criterion as a means to hasten the search invokes what we call the **small-is-quick** principle. That choice determines the kind of heuristic estimates ***h*** that should be provided in step 4 of *GBF*; ***h*** *should contain estimates of a set of parameters sufficient for the computation of the minimization criterion chosen.* Substituting these imperfect estimates for the actual values of the parameters constitutes the second principle and would be referred to as the **face-value** principle. Together these two principles determine the graph selection function f_1 in step 2 of *GBF*.

Following step 2, the rightmost candidate solution base is declared most promising, and its (only) OPEN node is expanded (step 3). This allows reevaluation of the candidate using the estimates assigned to its newly generated nodes. In our example (Figure 12.9(c)), the rightmost candidate was reevaluated to 7, rendering it inferior to the middle candidate, valued at 5. Therefore that latter candidate is declared most promising and draws future attention.

The graphs in Figure 2.9 illustrate successive developments of the search graph G' (in thin lines) and the shifting of its most promising solution base G_0 (in heavy lines). Figure 2.9(d) illustrates a situation where the node selection function f_2 is invoked for the first time. The reason for choosing the leftmost tip node of G_0 for expansion stems from the following argument: If the current G_0 is in fact a base of a legitimate solution, all its tip nodes must eventually be expanded, and the order of expansion is immaterial. If it is not, however, the most sensible node to expand would be the one that reveals the error the earliest.

To identify the node most likely to refute G_0's superiority, we need some information regarding the accuracy of each of the estimates. In the absence of such information all we can do is attempt to infer these accuracies from the magnitude of the estimates themselves, assuming, for example, that the errors are proportional to the magnitude of the estimates. In our example we may accordingly argue that the node most likely to prove G_0 inferior is the one with the largest size estimate, since a small percentage error in this estimate may cause large fluctuations in the value of G_0. More elaborate selection criteria can be applied if additional information is available regarding the candidate tip nodes. For example, it would be appropriate to combine both the likelihood that a node turns unsolvable with the estimate of the effort required to demonstrate it.

Figure 2.9(e) raises another dilemma. One successor (marked 1) of the node expanded at this stage is already on OPEN, with size estimate 1. Should this estimate be also attributed to its second father, or ignored because it has already been counted by its first father? The first alternative compromises our decision to seek the solution graph with the smallest number of edges; we may find ourselves chasing graphs with a different measure of size where each edge is counted several times depending on the number of parents it has. The second alternative, counting each edge only once, severely complicates the computation of f_1, making it impossible to calculate the merit of candidate solution bases from the estimates assigned to their frontier nodes even if these estimates were exact.

Take, for instance, the nodes marked 1 and 2 in Figure 2.9(d) and assume that these estimates are exact. The value of the parent node would be 6 if the optimal solution graphs beneath these two nodes were disjoint. On the other hand, the value of the parent is only 5 if the two solution graphs share an edge as in Figure 2.9(e). This complication implies that the contribution of a given frontier node to the size of a solution base is no longer a constant but may vary depending on the nature of the solution base for which this node is being considered. For example, the node marked 2 in Figure 2.9(e) may contribute only one edge to the size of G_0 but may contribute two full edges to a solution base emerging from its other parent, such as that in Figure 2.9(h). Thus each candidate solution base must be recalculated separately with each node expansion, even if the candidates share a large portion of the graph in common.

In optimization problems we have no choice but to pursue the second alternative, counting each edge once. In satisficing problems, however, because the choice of optimization pursuit was only an auxiliary device for achieving the solution more quickly, we may compromise the optimization purity for the sake of a more efficient computation of f_1.

Provisions for Optimality*: *GBF*. In Figure 2.9(f) a solution graph is found, the start node is labeled "solved," and the algorithm terminates with the current structure of G_0 whose value is 9. This is not the smallest size solution though; Figure 12.9(h) contains one with only seven edges. To guarantee that the *GBF* algorithm terminates with an optimal solution graph as defined by the function f_1, all cost estimates must be optimistic (i.e., underestimating costs and overestimating merit) and the termination condition must be modified. Instead of accepting the first solution that induces a "solved" label on the start node (step 5c), we should check for termination only after G_0 is selected (step 2) to see if G_0, in isolation, constitutes a solution graph. Thus, the modified algorithm, which we call *GBF**, would simply discard step 5c, and modify step 2 to read:

2′. Compute the most promising solution-base graph G_0; if all leaf nodes of G_0 are "solved," exit with G_0 as a solution graph.

Applying *GBF** to the situation in Figure 12.9(f) we see that the last expansion has increased the size of G_0 to 9, thus rendering it no longer most promising. Instead, the leftmost branch of Figure 12.9(g) promises a size of 7 and now assumes the G_0 status. In the next step, Figure 2.9(h), a new G_0 is found that retains its most-promising status and simultaneously proves to be a legitimate solution graph; at this point the algorithm terminates.

The fact that **GBF* always finds an optimal solution graph** (if the f_1 estimates are optimistic) can be seen by viewing the solutions as points in some space, the solution bases as subsets of points in that space and the operation of node expansion as the act of splitting those subsets. Since the estimates assigned to the subsets are optimistic (i.e., they exaggerate the merit of the best point in each subset) and the algorithm terminates only when it attempts to split the best subset and finds that it is a point (where the estimate is accurate), the point found could not be inferior to some other point. If it were, the subset containing that other point would have been chosen for splitting on account of its overly optimistic estimate.

This argument is the basis for the branch-and-bound method in operations research and several results in heuristic search theory. It will be proven formally in Chapter 3 in the context of path-seeking problems. The formal proof, however, essentially mirrors the steps of the set-splitting argument but lacks the latter's generality and appeal.

Relations between GBF and BF. If we let *GBF* search a state-space graph G (where all nodes are OR nodes), its operation would resemble that of *BF*. The two differ, however, in one essential feature: whereas *GBF* traverses the implicit graph G using an explicit *subgraph* G', *BF* traverses G using a *subtree* T. This difference results from step 2 of *BF* where, if a node is found to have two parents, an *irrevocable decision* is made to *discard* the one with the higher f value. That leaves each tip node in T to represent one unique solution base (the path leading to that node) and permits us to characterize the merit of each node with one parameter. Such irrevocable decisions are often justified in state-space search where the relative merits of the two contending solution bases usually will not be altered by new information regarding their common extension and where, more significantly, if one base is solvable, so is the other. These decisions are rarely justified in searching AND/OR graphs though, because one parent can be part of a solution graph, whereas the other parent may not be in one. So, by discarding one parent, we may be removing an edge from the only solution graph possible.

Note, however, that under certain circumstances parent-discarding may not be allowable even in path-seeking problems. For example, if we must find a path containing two edges with equal cost, we run the risk that the discarded subpath will turn out to contain the cost-matching edge needed for a solution. For that reason, *BF* can be guaranteed not to miss a solution only when the criterion qualifying a path as a solution invokes the properties of the *last* node on the path, but not when it depends on global properties of the path as a whole. This condition is assumed throughout our discussions, thus ruling out problems such as: "find the shortest path containing two equal edges."

In the preceding section we saw that if an optimal solution is required, *GBF* should be modified by retarding its termination test, thus leading to algorithm *GBF**. A similar modification can be performed on *BF* if we only delay its termination until a goal node is actually selected for expansion. The resulting procedure would naturally be called ***BF****, but its optimality guarantees are more involved than those of *GBF** because of the irrevocable parent-selection step of *BF*. For example, if we seek a path to a goal that has the smallest difference between the most costly and the least costly edges, we cannot discard one parent from future consideration simply because it currently represents a path with a higher difference than a competitor parent. The preference can easily be reversed through extension of the subpath common to the two parents (see exercise 2.4). In the next section and in Chapter 3 we establish conditions that guarantee that *BF** terminates with an optimal solution.

2.4 SPECIALIZED BEST-FIRST ALGORITHMS: Z*, A*, AO, AND AO*

2.4.1 Why Restrict the Evaluation Functions?

In their current general form, the algorithms describing *BF* and *GBF* are merely skeletons of strategies and are far from exhibiting the details necessary for implementation. Since the preference functions, f for *BF* and both f_1 and f_2 for *GBF*, remain arbitrary, the algorithms do not specify how these functions are computed, where the information needed for deciding the "best" choice comes from, or how it propagates through the graph. These issues constitute a major component of the search effort and will, therefore, play an important role in the taxonomy of best-first algorithms.

In large graphs, the computation of f_1 would be a hopeless task if each candidate subgraph had to be evaluated separately or if the entire set of candidates had to be reevaluated afresh with each node expansion. In the example of Figure 2.9, however, the computation of f_1 was facilitated by two main features:

1. *Shared computations* —intermediate computations in the evaluation of some solution candidates could be saved and reused in the evaluation of other candidates.

2. *Selective updating* —only ancestors of newly expanded nodes required updating, whereas all other nodes retained their values unaltered.

These two useful features are results of the **recursive** nature of the cost function chosen as a target of pursuit, that is, because the cost of solving node n could be determined from the costs of solving its successors. Indeed, we saw in Figure 2.9(e) how attempting to minimize the number of graph edges (a nonrecursive cost measure) led to computational difficulties, and how these difficulties disappeared as soon as we settled for minimizing a compromised measure, the number of edges in the tree-unfolding of the graph, which can be computed recursively. Much greater complications would have arisen had we chosen a more elaborate minimization measure, such as the median weight of the edges in the graph. (The median of a set of n numbers is the $(n/2)$th smallest number in the set.) This measure is notoriously difficult because it defies recursive formulation; the median of a set cannot be computed from the medians of its subsets but requires the values of every individual element in the set.

Fortunately common cost or merit criteria do not possess these difficulties. They exhibit a certain regularity in the way they combine, which greatly simplifies the implementation of best-first strategies. This regularity is formalized in the next section using the domain of AND/OR graphs.

2.4.2 Recursive Weight Functions

For a given solution graph G, we designate its weight by W_G, where W_G is that property of G chosen as an optimization measure, representing either merit (Q) or cost (C). If we remove from G all nodes but those that are descendants of some given node n, the remaining portion of the graph is a solution graph for n, and its weight is denoted by $W_G(n)$. In general, the weight of any solution graph may be a complex function of all the quantities in the graph: node weights (e.g., representing processing delays), arc weights (e.g., representing transition costs), and terminal node weights (e.g., representing final payoffs).

DEFINITION: *A weight function $W_G(n)$ is* **recursive** *if for every node n in the graph*

$$W_G(n) = F[\mathbf{E}(n); W_G(n_1), W_G(n_2), \ldots, W_G(n_b)] \tag{2.1}$$

where: $n_1, n_2, \ldots, n_b$ *are the immediate successors of n.*
$\mathbf{E}(n)$ *stands for a set of local properties characterizing the node n.*
F is an arbitrary combination function, monotonic in its $W_G(\cdot)$ arguments.

If such a combining function F exists, it is possible to evaluate the merit of any given solution graph from the bottom up, starting at the payoffs associated with the terminal nodes and working upward until the merit of the entire solution graph is computed at the root node. The process is similar to the "solve" labeling procedure described in Section 1.2.3 and will henceforth be called the **cost labeling, merit labeling,** or **weight labeling** procedure, as the case requires. F is sometimes called the **rollback function.**

Let us demonstrate the weight-labeling procedure on a few typical examples:

1. **The Counterfeit Coin problem with the maximum number of tests as a weight measure.** The solution objects are test-specification graphs where the terminal nodes represent an identified coin and the nonterminal nodes are of two types: *action nodes* (OR nodes with a single successor specifying the type of test to be conducted) and *outcome nodes* (AND nodes whose AND links point at the three possible outcomes of each test). The overall cost of a given solution graph G is taken to be the number of tests along the longest path in the graph. Thus the cost combining function can be written:

$$C_G(n) = \begin{cases} 0 & \text{if } n \textit{ is terminal} \\ 1 + C_G(n_1) & \text{if } n \textit{ is an OR node} \\ \max\limits_{i=1,2,3} [C_G(n_i)] & \text{if } n \textit{ is an AND node} \end{cases} \tag{2.2}$$

2. **The Counterfeit Coin problem with the expected testing cost as a weight measure.** Let $c(n, n')$ be the cost of the test represented by the arc from n to n', and let p_1, p_2, and p_3 be the probabilities associated with the three outcomes of any given test. The expected cost of the strategy represented by a solution graph G rooted at n is given by:

$$C_G(n) = \begin{cases} 0 & \text{if } n \textit{ is terminal} \\ c(n, n_1) + C_G(n_1) & \text{if } n \textit{ is an OR node} \\ \sum\limits_{i=1,2,3} p_i\, C_G(n_i) & \text{if } n \textit{ is an AND node} \end{cases} \tag{2.3}$$

3. **Game-playing strategies.** The quality Q of a given strategy S is determined by the payoff $v(n)$ given to player 1 from the terminal node n which is eventually reached. Assuming, however, that player 2 acts in such a way as to minimize this payoff, we have:

$$Q_S(n) = \begin{cases} v(n) & \text{if } n \textit{ is terminal} \\ Q_S(n_1) & \text{if } n \textit{ admits moves by player } 1 \\ \min\limits_i Q_S(n_i) & \text{if } n \textit{ admits moves by player } 2 \end{cases} \tag{2.4}$$

Thus the quality of any fixed strategy is given by the lowest payoff over all the terminal nodes in the graph representing this strategy.

These three examples represent the use of the two most common cost-combination rules: maximum-cost and (weighted) sum-cost. In these examples the local properties $\mathbf{E}(n)$ that influence the combination rule are the costs and probabilities associated with the arcs and some characteristics of the node itself, for example, whether it is a terminal node, an OR node, or an AND node.

2.4.3 Identifying G_0, The Most Promising Solution-Base Graph

We will now demonstrate how the recursive nature of the weight measure W simplies the task of rating the candidate solution bases. We assume that the weight signifies the quality or merit of a given solution with the understanding that *cost measures are simply the negative of quality*.

If the entire search space had been explored, an *optimal* solution graph could be constructed and its quality $Q^*(s)$ could be computed by taking the maximum of $Q_G(s)$ over all solution graphs rooted at s. This maximization can be performed recursively by the following **merit-labeling procedure**:

1. If n is a terminal node, then $Q^*(n) = v(n)$, where $v(n)$ is the terminal payoff associated with n.
2. If n is a nongoal node without successors thus representing an unsolvable subproblem, then $Q^*(n)$ is $-\infty$.
3. If n is an AND node with successors $n_1, n_2, \ldots, n_b$, then $Q^*(n) = F[\mathbf{E}(n); Q^*(n_1), Q^*(n_2), \ldots, Q^*(n_b)]$.
4. If n is an OR node with successors $n_1, n_2, \ldots, n_b$, then $Q^*(n) = \max\limits_i F[\mathbf{E}(n), Q^*(n_i)]$.

According to this definition, $Q^*(n)$ is finite if and only if the problem represented by node n is solvable. For each solvable n, $Q^*(n)$ gives the quality of an optimal solution graph rooted at n. If s is the start node, then $Q^*(s)$ is the quality of an optimal solution to the initial problem.

For example, in example 2 of the Counterfeit Coin problem in Section 2.4.2, the optimal cost $C^*(n)$ associated with subproblem n is computed by the cost-labeling procedure:

$$C^*(n) = \begin{cases} 0 & \text{if } n \textit{ is terminal} \\ \min\limits_i [c(n, n_i) + C^*(n_i)] & \text{if } n \textit{ is an OR node} \\ \sum\limits_i p_i\, C^*(n_i) & \text{if } n \textit{ is an AND node} \end{cases} \tag{2.5}$$

In game-playing (example 3 in Section 2.4.2), the merit-labeling procedure yields the celebrated minimax rule:

$$Q^*(n) = \begin{cases} v(n) & \text{if } n \textit{ is terminal} \\ \max_i [Q^*(n_i)] & \text{if } n \textit{ admits moves by player } 1 \\ \min_i [Q^*(n_i)] & \text{if } n \textit{ admits moves by player } 2 \end{cases} \tag{2.6}$$

The **optimal solution graph** G^* is likewise defined in terms of Q^*.

DEFINITION OF G^*:

1. *The start node s is in G^*.*
2. *If an AND node is in G^*, all its (AND) successors are in G^*.*
3. *If an OR node n is in G^*, one successor n′ of n is in G^* such that* $F[\mathrm{E}(n); Q^*(n')] = \max_i F[\mathrm{E}(n); Q^*(n_i)]$.

If the search space is only partially explored, there is no way, of course, to compute the optimal solution or even to identify which among the solution bases unraveled leads to an optimal solution. However, if each node n at the search frontier is assigned an *estimate* $h(n)$ of $Q^*(n)$, we may invoke the face-value principle and compute what appears to be the **most promising potential solution**. This solution is computed in two steps:

1. Assign to each tip node n of a solution base an evaluation function $h(n)$ which estimates the quality of the best solution rooted at n.
2. Use the combining function F and the merit-labeling procedure *as if* the estimates $h(n)$ were true terminal payoffs (the face-value principle).

These two steps offer constructive definitions of the function f_1 and the most promising solution base G_0, both of which are used in step 2 of *GBF*. First, we define a **backed-up evaluation function** $e(n)$ which gives an estimate of $Q^*(n)$ for all nodes of the explored graph:

$$e(n) = \begin{cases} h(n) & \text{if } n \textit{ is in OPEN} \\ F[\mathrm{E}(n); e(n_1) \cdots e(n_b)] & \text{if } n \textit{ is a CLOSED AND node} \\ \max_i F[\mathrm{E}(n); e(n_i)] & \text{if } n \textit{ is a CLOSED OR node} \end{cases} \tag{2.7}$$

This evaluation function gives rise to the following top-down definition of G_0.

DEFINITION OF G_0:

1. *The start node s is in G_0.*
2. *If a CLOSED AND node is in G_0, all its (AND) successors are in G_0.*
3. *If a CLOSED OR node n is in G_0, one (OR) successor n′ of n is in G_0, such that* $e(n) = F[\mathrm{E}(n); e(n')]$.

In practice, the identification of G_0 can be done bottom-up, together with the computation of $e(n)$, by simply marking the OR successor that has the highest value of e among its siblings.

In this procedure, G_0 is identified without making explicit use of the graph selection function f_1, however, if one wishes to assess the promise $f_1(G')$ of any candidate solution-base graph G', all that is required is to compute e bottom-up along the arcs in G' (each OR node will have only one successor since G' is a solution base). The value finally assigned to the start node $e_{G'}(s)$, is equal to the required promise estimate $f_1(G')$:

$$f_1(G') = e_{G'}(s) \tag{2.8}$$

It is interesting to see how *BF* can use this procedure to compute the node selection function $f(n)$ while traversing an OR graph. Since *BF* explores the graph using a traversal tree T, each node n on OPEN takes part in only one solution base, given by the path $P(n)$ from s to n, currently in T. To decide (in step 3 of *BF*) which node warrants expansion, the evaluation function $e_{P(n)}(s)$ must be assigned to each node on OPEN, where $e_{P(n)}(s)$ is computed bottom-up (using F as in Eq. 2.7) along the unique path connecting s and n. Thus the function $f(n)$, computed in step 6 of *BF*, is merely the negative of $e_{P(n)}(s)$:

$$f(n) = -e_{P(n)}(s) \tag{2.9}$$

(Tradition dictates that f should represent costs and therefore be minimized, whereas e represents merits and is to be maximized.)

Under normal circumstances (e.g., when the solution weight is determined by the sum cost of its edges), we need not traverse the entire path from n to s to compute $e_{P(n)}(s)$. A few auxiliary parameters stored at the parent of n would permit $e_{P(n)}(s)$ to be calculated locally and be transmitted, from father to son, with each node expansion. These parameters correspond to the notion of state description in dynamic programming (Dreyfus and Law, 1977). For example, in the case of the additive cost measure, we need to transmit only one parameter, $g(n)$, which is the overall cost along the path $P(n)$, and this gives rise to the celebrated A^* algorithm (Nilsson, 1971) with heuristic function:

$$f(n) = -e_{P(n)}(s) = g(n) + h(n) \tag{2.10}$$

for each node on OPEN.

2.4.4 Specialized Best-First Strategies

If the selection functions f_1 (for *GBF*) and f (for *BF*) are computed recursively by the rollback procedure of Eqs. (2.7) and (2.8), these algorithms assume special names; *BF* becomes what we will call algorithm **Z** and *GBF* becomes

AO. If we further modify these algorithms by delaying the termination test as was done in GBF^*, Z becomes $\mathbf{Z^*}$ and AO becomes $\mathbf{AO^*}$.

The motivation for delaying the termination test until G_0 is selected lies, of course, in the hope of returning an optimal solution when one is required. This hope will be realized for AO^* if the heuristic estimates h of the tip nodes are optimistic (i.e., overestimating merits and underestimating costs), because under such conditions the estimates $f_1(G')$ are also optimistic for all solution bases, and, invoking the same set-splitting argument used for GBF^*, AO^* cannot return a suboptimal solution.

The optimality condition for Z^* is slightly more involved due to the parent-selection step of BF. In addition to requiring that the estimates $h(n)$ be optimistic, we still need to ascertain that by discarding the parent with the inferior f value we do not in fact throw away the optimal solution path. This would be guaranteed when the rank order of two contending parents remains independent of the path rooted at their common child. Therefore, we need also to require that F, the rollback function that defines the cost of any given path, satisfies:

$$F(\mathbf{E}_1, h) \geqslant F(\mathbf{E}_2, h) \Longrightarrow F(\mathbf{E}_1, h') \geqslant F(\mathbf{E}_2, h') \qquad (2.11)$$

for all $\mathbf{E}_1$, $\mathbf{E}_2$, h and h'. If the path-cost definition satisfies the **order-preserving** property of Eq. (2.11) and the estimates $h(n)$ are optimistic, Z^* terminates with an optimal (lowest cost) path. Fortunately the two most commonly used weight measures—the additive cost (where $F = c(n, n') + h(n')$) and the max cost (where $F = \max[c(n, n'), h(n')]$)—do satisfy this property. A more general discussion of the order-preserving restriction is given in Section 3.3, pertaining to the optimality of BF^*.

So far we have encountered eight variations of best-first strategies, four suitable for AND/OR graphs (GBF, GBF^*, AO, and AO^*) and four limited to searching OR graphs (BF, BF^*, Z, and Z^*). Before introducing our last strategy, the celebrated specialization of Z^* called A^*, it is instructive to reiterate the relationships between the algorithms defined so far. They are illustrated in the hierarchical diagram of Figure 2.10, where each arrow specifies the restriction placed on the parent algorithm in order to produce the more specialized successor.

Figure 2.10 also shows the position of A^* in this hierarchy; it is a specialization of Z^* where the target of pursuit is the path of minimum sum-cost. A^* may be used both for optimization problems and satisficing problems because the shortest path is the most natural choice for the small-is-quick principle. The reader may wonder why Figure 2.10 does not contain an additive-cost successor of AO^*, which, in fact, was described in the example of Figure 2.9. The reason is simply that the additive-cost condition does not introduce a significant enough change in the algorithm to warrant a new name. A^*, too, is only slightly different from Z^*; however, in order to keep our notation consistent with the large body of literature on A^* and out of respect for its inventors (Hart, Nilsson, and Raphael, 1968) we prefer to keep its traditional name intact.

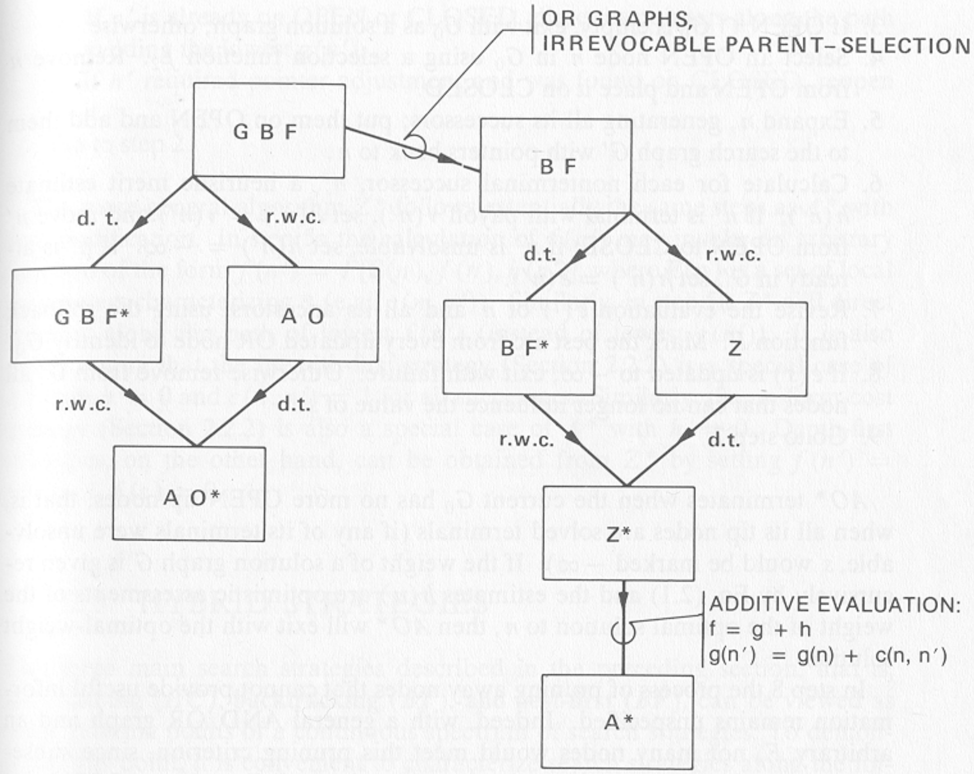

Figure 2.10
Hierarchical diagram showing the relationships between the nine best-first algorithms discussed in Section 2.4 (d.t. stands for delayed-termination; r.w.c. stands for recursive-weight-computation).

Since AO^* and A^* are the most popular strategies in use, we now give an explicit description of these two algorithms, followed by a few words of explanation. A formal treatment of the properties of A^* and BF^* is to be found in Chapter 3.

The AO Algorithm*

1. Create a search graph G' initially consisting of the start node s. Put s on OPEN. Let G_0 initially include just s.
2. Trace down the marked connectors of a subgraph G_0 (G_0 stands for the most promising solution-base graph, and its connectors are marked in step 7) and inspect its tip nodes.

3. If OPEN $\cap$ G_0 is empty, exit with G_0 as a solution graph; otherwise
4. Select an OPEN node n in G_0 using a selection function f_2. Remove n from OPEN and place it on CLOSED.
5. Expand n, generating all its successors; put them on OPEN and add them to the search graph G' with pointers back to n.
6. Calculate for each nonterminal successor, n', a heuristic merit estimate $h(n')$. If n' is terminal with payoff $v(n')$, set $h(n') = v(n')$ and move n' from OPEN to CLOSE; if n' is unsolvable, set $h(n') = -\infty$. If n' is already in G', set $h(n') = e(n')$.
7. Revise the evaluation $e(\cdot)$ of n and all its ancestors, using the rollback function F. Mark the best arc from every updated OR node to identify G_0.
8. If $e(s)$ is updated to $-\infty$, exit with failure. Otherwise remove from G' all nodes that can no longer influence the value of s.
9. Go to step 2.

AO^* terminates when the current G_0 has no more OPEN tip nodes, that is, when all its tip nodes are solved terminals (if any of its terminals were unsolvable, s would be marked $-\infty$). If the weight of a solution graph G is given recursively by Eq. (2.1) and the estimates $h(n)$ are optimistic assessments of the weight of the optimal solution to n, then AO^* will exit with the optimal-weight solution.

In step 8 the process of pruning away nodes that cannot provide useful information remains unspecified. Indeed, with a general AND/OR graph and an arbitrary F, not many nodes would meet this pruning criterion, since subsequent expansions may significantly alter the values of all nodes on OPEN and their ancestors. Only unsolvable nodes are truly safe from future alterations and can be pruned away. If, however, F is known to comply with additional restrictions (e.g., that $e(n)$ cannot increase by expansion) or if the graph searched is a tree, this pruning can result in substantial savings. We therefore postpone description of the pruning process until we have discussed some specific types of F functions.

The A* Algorithm

1. Put the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN and place on CLOSED a node n for which f is minimum.
4. If n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s.
5. Otherwise expand n, generating all its successors, and attach to them pointers back to n. For every successor n' of n:
   a. If n' is not already on OPEN or CLOSED, estimate $h(n')$ (an estimate of the cost of the best path from n' to some goal node), and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$.

   b. If n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n')$.
   c. If n' required pointer adjustment and was found on CLOSED, reopen it.
6. Go to step 2.

The more general algorithm Z^* follows essentially the same steps as A^* with one modification. In step 5a the calculation of $f(n')$ may invoke an arbitrary function of the form $f(n') = F[\mathbf{E}(n), f(n), h(n')]$, where $\mathbf{E}(n)$ is a set of local parameters characterizing n (e.g., $c(n, n')$). Similarly, in step 5b Z^* will direct pointers along the path of lowest $f(n')$ (instead of lowest $g(n')$). It is also worth noting that the breadth-first strategy (Section 2.2.2) is a special case of A^* with $h = 0$ and $c(n, n') = 1$ for all successors. Similarly, the uniform-cost strategy (Section 2.2.2) is also a special case of A^* with $h = 0$. Depth-first strategies, on the other hand, can be obtained from Z^* by setting $f(n') = f(n) - 1, f(s) = 0$.

2.5 HYBRID STRATEGIES

The three main search strategies described in the preceding section, that is, **hill-climbing (*HC*), backtracking (*BT*), and best-first (*BF*)**, can be viewed as three extreme points of a continuous spectrum of search strategies. To demonstrate this point, it is convenient to characterize search strategies along the following two dimensions:

1. **Recovery of pursuit** (R): The degree to which a search strategy allows recovery from disappointing search avenues to reaccess previously suspended alternatives.
2. **Scope of evaluation** (S): The number of alternatives considered in each decision.

Along the R dimension we find HC at one extreme, permitting no recovery whatsoever of suspended alternatives, and BT and BF at the other extreme, where all decisions are tentative and all suspensions are revocable. Along the S dimension we find HC and BT focusing narrowly on the set of newly available alternatives, whereas BF examines before each decision the entire set of available alternatives, those newly generated as well as all those suspended in the past.

These extreme cases are illustrated schematically in Figure 2.11. The shaded area represents a continuous spectrum of search strategies that combine some characteristics from each of the three prototypes in order to achieve a better mix of their computational features. To recapitulate, recall that HC spends the minimal amount of computation at the risk of missing the solution. BF

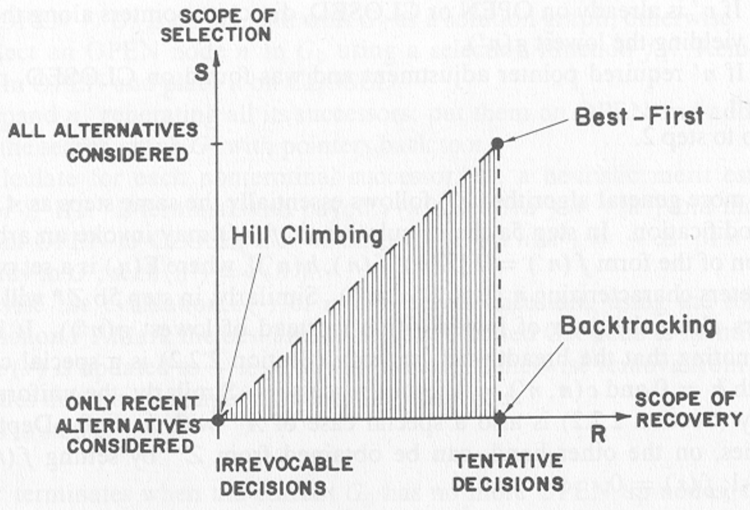

Figure 2.11
Hill-climbing (HC), backtracking (BT), and best-first (BF) strategies as three extreme points in a 2-dimensional space of hybrid strategies.

proceeds wisely and cautiously; it will find the solution (if one exists) after the fewest possible decisions but must pay dearly in node storage for these judicious decisions. *BT* is committed to maintaining in storage only a single path containing the set of alternatives leading to the current decision point. It proceeds forward heedlessly by considering only a narrow scope of young alternatives, and so it usually pays for its memory economy in increased run-time.

2.5.1 *BF-BT* Combinations

If one cannot afford the memory space required by a pure *BF* strategy, various *BF-BT* combinations can be implemented that cut down the storage requirement at the expense of narrowing the evaluation scope. One such combination is depicted in Figure 2.12(*a*), where the *BF* strategy is applied at the top of the search graph (represented by the shaded area with the irregular frontier) and a *BT* strategy at the bottom (represented by the left-to-right arrow). As soon as the memory space allotted to the *BF* strategy is exhausted, the *BT* search takes over from the best node on OPEN until the entire graph beneath that node is traversed. If *BT* fails to find a solution, this node is declared unsolvable and the second best node on OPEN is taken as a new root for a *BT* search, and so on.

An opposite approach is shown in Figure 2.12(*b*). Here *BT* is employed at the top of the graph, whereas *BF* is used at the bottom. We begin *BT* until a depth-bound d_0 is reached. At this point, instead of backing up, we employ a

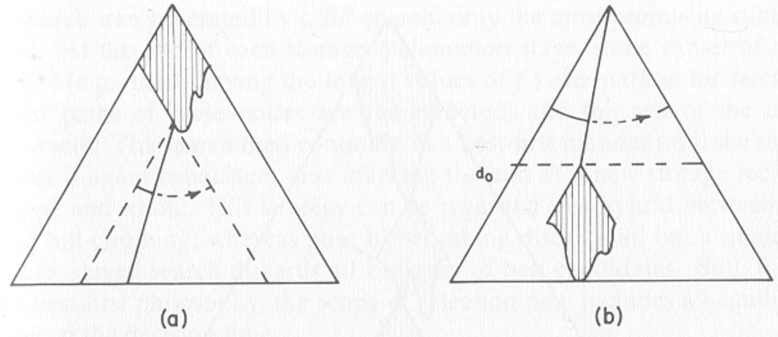

Figure 2.12
Schematic representations of two BF-BT hybrid strategies. Part (*a*) represents a BF search on top (shaded area) followed by BT ending. Part (*b*) represents a BT start (left-to-right arrow) followed by BF ending.

BF search from the node at d_0 until it returns a solution or exits with failure. If it fails to find a solution, we return to backtracking and again use *BF* upon reaching the depth-bound d_0. The parameter d_0 is chosen in such a way that the memory consumed by the *BF* strategy will not exceed the available allotment. It is harder to control than the strategy in Figure 2.12(*a*), where memory utilization itself was the mechanism that triggered the transition from *BF* to *BT*, but employing *BF* at the bottom of the graph has an advantage. *BF* performance is at its best when its guiding heuristic is more informed, and this usually happens at the bottom of the search graph, where the differences between promising nodes and dead-end nodes are more apparent.

A more elaborate scheme is shown in Figure 2.13, where *BF* is performed locally and *BT* globally. We begin searching in a best-first manner until a memory allotment M_0 is exhausted. At this point we regard all the nodes on OPEN as direct successors of the root node and submit them to a *BT* search. *BT* selects the best among these successors and "expands" it using *BF* search, that is, it submits the chosen node to a local *BF* search until a new memory allotment M_0 is consumed and treats the new nodes on OPEN as direct successors of the node "expanded." In summary, this strategy amounts to running an informed depth-first search where each node expansion is accomplished by a memory-limited *BF* search and the nodes on OPEN are defined as children. The overall memory required for this strategy is equal to M_0 times the maximum length of the global traversal path. It is linear with the depth of the search graph, in contrast to the exponential memory growth required by purely *BF* strategies.

Ibaraki (1978b) has proposed another scheme of mixing *BF* and *BT* strategies. Whereas informed depth-first seach considers for expansion only the newly generated nodes, Ibaraki's scheme expands the scope of selection to include, in addition to the set of new successors, the k best alternatives from the set considered at the previous decision. Thus, at the t^{th} decision, the node ex-

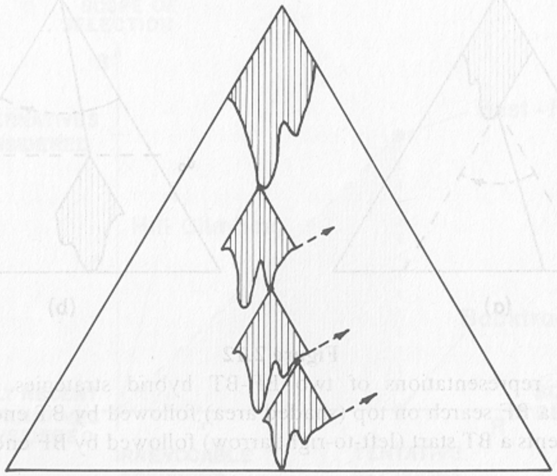

Figure 2.13
Schematic representation of a hybrid BF-BT strategy. Local BF searches (shaded areas) are nested in a global BT traversal.

panded is the best within a subset of OPEN, which we will call FOCAL(t), containing the set of newly generated nodes together with the k best nodes of FOCAL($t-1$). Clearly, the case $k = 0$ corresponds to pure depth-first while $k = \infty$ amounts to pure *BF*. Although this scheme does not possess an elegant geometrical representation such as those shown in Figures 2.12 and 2.13, Ibaraki was able to show that the storage consumed by this strategy is equal to bN^{k+1} where N is the depth of the search tree and b is its branching degree. One can, therefore, adjust k to meet practical storage limitations.

2.5.2 Introducing Irrevocable Decisions

The need to combine *BF* and *BT* strategies was a result of computational considerations aimed at finding a good storage-time mix when storage space is limited. These combinations retained the purely tentative nature of all decisions. As soon as we introduce irrevocable decisions of the hill-climbing type, permanently discarding alternatives before they are proven fruitless, we also introduce a risk of missing a solution and exiting with failure, or returning a bad solution when an optimal one is required. Occasions may arise in which even the storage required for local best-first search is too large and purely depth-first strategies are too time consuming. In such cases, it may be desirable to take a risk and prune the tree so far generated by the search in order to free needed storage space and allow the search to continue deeper.

A pruning technique that has gained some popularity is called **staged search** (Nilsson, 1971) and proceeds as follows. Instead of maintaining in storage the entire search tree generated by a *BF* search, only the most promising subtree is retained. At the end of each storage reclamation stage, some subset of nodes on OPEN (e.g., those having the lowest values of f) are marked for retention. The best paths of these nodes are remembered, and the rest of the tree is thrown away. The search then continues in a best-first manner until the storage allotment is again exhausted, thus marking the end of a new storage reclamation stage, and so on. This strategy can be regarded as a hybrid between best-first and hill-climbing; whereas pure hill-climbing discards all but a single best candidate, staged search discards all but a *set* of best candidates. Still, following the best-first philosophy, the scope of selection now includes all candidates available at the decision time.

An important variant of staged search also introduces an element of *BT* search into the *HC-BF* mixture. It is tailored after the hybrid strategy of Figure 2.13, but instead of submitting the entire frontier of the local *BF* search to a global *BT*, only the best subset of these frontier nodes are retained whereas the rest are discarded permanently. Of course, even if A^* is used in each stage and the whole process does terminate in a solution path, there is now no guarantee that it is an optimal path. In Chapter 5 we analyze a version of this search strategy and show that under certain conditions one can choose the search parameters (e.g., the size of the local search tree and the threshold of pruning) in such a way that the search will run in linear average time and the solution found will almost always fall in an arbitrarily close neighborhood of the optimal solution.

2.6 BIBLIOGRAPHICAL AND HISTORICAL REMARKS

The book by Horowitz and Sahni (1978) contains a thorough discussion of backtracking and other search methods including a more detailed specification of the computational steps involved. Improvements on the basic backtracking algorithm are reported and analyzed by Gaschnig (1979a), Haralick and Elliot (1980), and Nudel (1983). Look-ahead methods of solving satisficing problems, sometimes called "relaxation," "constraint propagation," or "consistency checking," were introduced by Rosenfeld, Hummel, and Zucker (1976), Waltz (1975), Montanari (1974), and Mackworth (1977). These are especially effective in object-recognition tasks where constraints have a local character (e.g., only few lines meet in each vertex of a polyhedran object).

Systematic, uninformed graph-searching procedures are normally discussed under the topic "shortest path problems." Moore's (1959) maze searching algorithm is essentially a breadth-first strategy, while Dijkstra's (1959) two-point shortest path algorithm is what we call uniform-cost (or A^* with $h=0$). An up-to-date survey of shortest path methods is given in Deo and Pang (1982). Dynamic programming (Bellman and Dreyfus, 1962) is a recursive formulation of breadth-first search which, for search spaces with regular structures, may yield analytical expressions for the optimal cost or the optimal policy.