Chapter 1

Heuristics and Problem Representations

1.1 TYPICAL USES OF HEURISTICS IN PROBLEM SOLVING

Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices.

A heuristic may be a **rule of thumb** that is used to guide one's actions. For example, a popular method for choosing ripe cantaloupe involves pressing the spot on the candidate cantaloupe where it was attached to the plant, and then smelling the spot. If the spot smells like the inside of a cantaloupe, it is most probably ripe. This rule of thumb does not guarantee choosing only ripe cantaloupe, nor does it guarantee recognizing each ripe cantaloupe judged, but it is effective most of the time.

As an example of a less clear-cut use of heuristics, consider a chess grand master who is faced with a choice of several possible moves. The grand master may decide that a particular move is most effective because that move results in a board position that "appears" stronger than the positions resulting from the other moves. The criterion of "appearing" stronger is much simpler for the grand master to apply than, say, rigorously determining which move or moves forces a checkmate. The fact that grand masters do not always win indicates that their heuristics do not guarantee selecting the most effective move. Finally, when asked to describe their heuristics, grand masters can only give partial and rudimentary descriptions of what they themselves seem to apply so effortlessly.

Even the decision to begin reading this book reflects a tacit use of heuristics—heuristics that have led the reader to expect greater benefit from engaging in this activity rather than doing something else at this point in time.

It is the nature of good heuristics both that they provide a simple means of indicating which among several courses of action is to be preferred, and that they are not necessarily guaranteed to identify the most effective course of action, but do so sufficiently often.

Most complex problems require the evaluation of an immense number of possibilities to determine an exact solution. The time required to find an exact solution is often more than a lifetime. Heuristics play an effective role in such problems by indicating a way to reduce the number of evaluations and to obtain solutions within reasonable time constraints.

To illustrate the role of heuristics in problem solving, we make use of five puzzlelike problems which have served as expository tools in both psychology and artificial intelligence (AI). The expository power of puzzles and games stems from their combined *richness* and *simplicity.* If we were to use examples taken from complex real-life problems, it would take us more than a few pages just to lay the background and specify the situation in sufficient detail so that the reader could appreciate the solution method discussed. Moreover, even after making such an effort we cannot be sure that different readers, having accumulated different experiences, would not obtain diversely different perceptions of the problem situation described. Games and puzzles, on the other hand, can be stated with sufficient precision so as to lay a common ground for discussion using but a few sentences and, yet, their behavior is sufficiently rich and unpredictable to simulate the complexities encountered in real-life situations. This, in fact, is what makes games and puzzles so addictive.

1.1.1 The 8-Queens Problem

To solve this problem, one must place eight queens on a chess board such that no queen can attack another. This is equivalent to placing the queens so that no row, column, or diagonal contains more than one queen.

We can attempt to solve this problem in many ways, from consulting a mystic to looking up the solution in a puzzle book. However, a more constructive method of attack would be to forgo hopes of obtaining the final solution in one step and, instead, to attempt reaching a solution in an **incremental,** step-by-step manner, in much the same way that a treasure hunter "closes in" on a target following a series of local decisions, each based on new information gathered along the way. We can start, for example, with an arbitrary arrangement of eight queens on the board and transform the initial arrangement iteratively, going from one board configuration to another, until the eight queens are adequately dispersed. As with the treasure-hunt, however, we must also make sure that the sequence of transformations is not random but **systematic** so that we do not generate the same configuration over and over and so that we do not miss the opportunity of generating the desired configuration. The former is a requirement of efficiency, whereas the latter is one of utility.

One way to systematize the search is to attempt to **construct** (rather than transform) board configurations in a step-by-step manner. Starting with the empty board we may attempt to place the queens on the board one at a time, ruling out violations of the problem constraints, until a satisfactory configuration with eight queens is finally achieved. Moreover, because we can easily deduce from the problem specification that there must be exactly one queen in every row, we can assign the first queen to the first row, the second queen to the second row, and so on, thus reducing the number of alternative positions considered for each new queen to less than 8. The feature that allows us to systematize the search in this manner is the fact that we cannot recover from constraint violations by future operations. Once a partial configuration violates any of the problem constraints (i.e., it contains two or more attacking queens), it cannot be rectified by adding more queens to the board, and so many useless configurations can be eliminated at an early stage of the search process.

Assume now that at some stage of the search, the first three queens were positioned in the cells marked by Q's in Figure 1.1 and that we must decide where to position the fourth queen, in cell A, B, or C. The role of heuristics in this context would be to provide rule-of-thumb criteria for deciding, at least tentatively, which of the three positions appears to have the highest chance of leading to a satisfactory solution.

In devising such heuristics, we may reason that to be able to place all the eight queens we need to leave as many options as possible for future additions. That means we need to pay attention to the number of cells in the unfilled rows that remain unattacked by the previously placed queens. A candidate placement is to be preferred if it leaves the highest **number of unattacked cells** in the

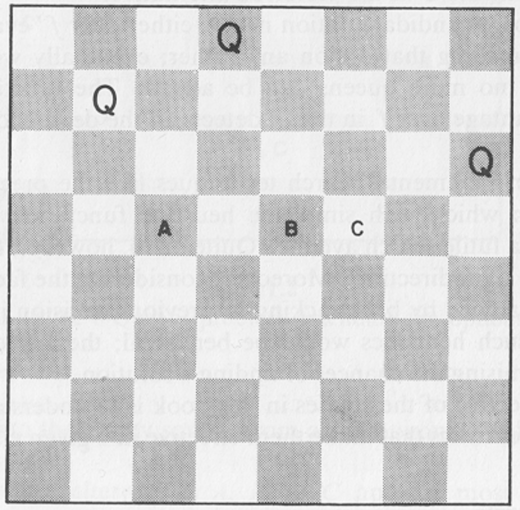

Figure 1.1
A typical decision step in constructing a solution to the 8-Queens problem.

remaining portion of the board. Consequently the number of unattacked cells left by any given alternative constitutes a measure, $f(\cdot)$, of its merit.

If we compute the function $f(\cdot)$ for the alternatives A, B, and C in the preceding example, we obtain

$$f(A) = 8$$
$$f(B) = 9$$
$$f(C) = 10$$

So alternative C should be our preferred choice. Adding credence to the effectiveness of our heuristic is the fact that alternatives B and C both lead to actual eight-queens solutions as shown in Figure 1.2, whereas alternative A leads to no such solution.

A more sophisticated heuristic can be devised by the following consideration. The unfilled rows do not have equal status since those with only few unattacked cells are likely to be "blocked" quicker than rows with many unattacked cells. Consequently, if we wish to minimize the chances of a future blockage, we should focus our attention on the row with the **least number of unattacked cells** and use this number, f', as a figure of merit of each option considered. Performing this computation for the options presented in Figure 1.1, we obtain

$$f'(A) = 1$$
$$f'(B) = 1$$
$$f'(C) = 2$$

which, again, identifies C as the preferred alternative.

Note that if some candidate option makes either f or f' evaluate to 0, there is no sense in pursuing that option any futher; eventually we will come to a **dead end** where no more queens can be added. The function f', however, offers some advantage over f in that it detects all the dead ends predicted by f and many more.

Obviously, for incremental search techniques like the preceding one, there may be cases in which such simplistic heuristic functions would assign the highest value to a futile search avenue. Quite often, however, they would guide the search in the right direction. Moreover, considering the fact that we can recover from bad advice by backtracking to previous decision junctions, the net effect of using such heuristics would be beneficial; they speed up the search without compromising the chances of finding a solution.

The main objective of the studies in this book is to understand and quantify the features of heuristics that make them effective in a given class of problems.

1.1.2 The 8-Puzzle

The objective of the 8-Puzzle is to rearrange a given initial configuration of eight numbered tiles arranged on a 3×3 board into a given final configuration called the **goal state**. The rearrangement is allowed to proceed only by sliding

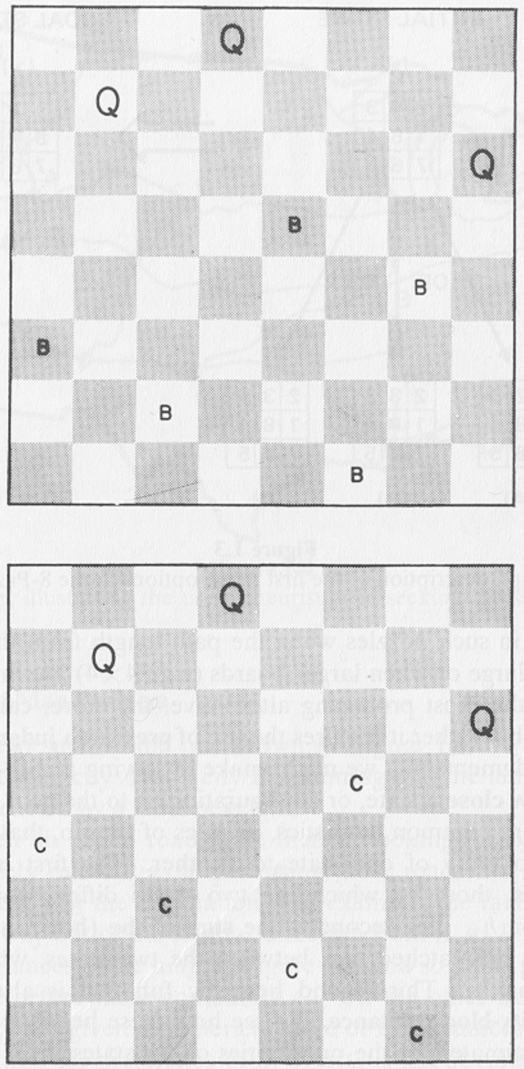

Figure 1.2
Two solutions to the 8-Queens problem extending the options shown in Figure 1.1.

one of the tiles onto the empty square from an orthogonally adjacent position, as shown in Figure 1.3

Which of the three alternatives A, B, or C appears most promising? Of course, the answer can be obtained by exhaustively searching subsequent moves in the puzzle to find out which of the three states leads to the shortest path to the goal. **Exhaustive search**, however, is utterly impractical when the number of states that must be examined is immense. This "combinatorial ex-

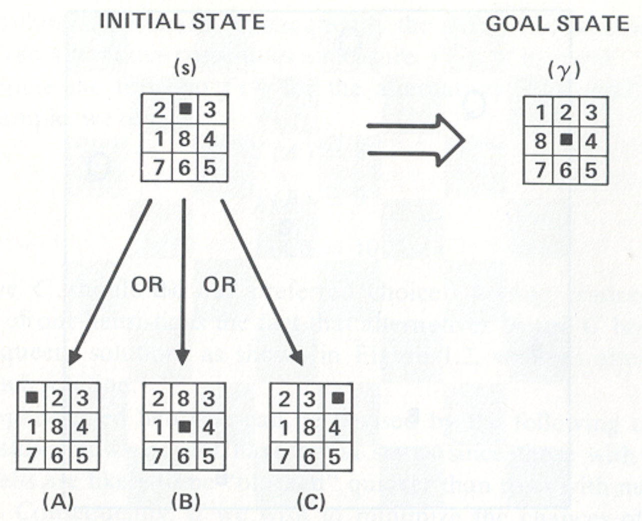

Figure 1.3
A graph description of the first move options in the 8-Puzzle.

plosion" occurs in such puzzles when the path length from the initial state to the goal state is large or when larger boards (e.g., 4×4) are involved. The decision to select the most promising alternative, therefore, cannot rely on exhaustive search, but rather it requires the use of **presearch judgment.**

One of the judgments that we might make in solving a path-finding problem is **estimating how close** a state, or configuration, is to the goal. In the 8-Puzzle there are two very common heuristics, or rules of thumb, that are used for estimating the proximity of one state to another. The first is the **number of mismatched tiles,** those by which the two states differ, which we will call heuristic function h_1. The second is the **sum of the** (horizontal and vertical) **distances** of the mismatched tiles between the two states, which we will call heuristic function h_2. This second heuristic function is also known as the **Manhattan** or **city-block distance.** To see how these heuristic functions work, below are the estimates for the proximities of the states A, B, and C from the goal state in Figure 1.3:

$$h_1(A) = 2 \quad h_1(B) = 3 \quad h_1(C) = 4$$

$$h_2(A) = 2 \quad h_2(B) = 4 \quad h_2(C) = 4$$

Evidently, both heuristics proclaim that state A is the one closest to the goal and that it should, therefore, be explored before B or C.

These heuristic functions are intuitively appealing and readily computable, and may be used to prune the space of possibilities in such a way that only configurations lying close to the solution path will actually be explored. Search strategies that exhibit such pruning through the use of heuristic functions will be discussed in Chapter 2.

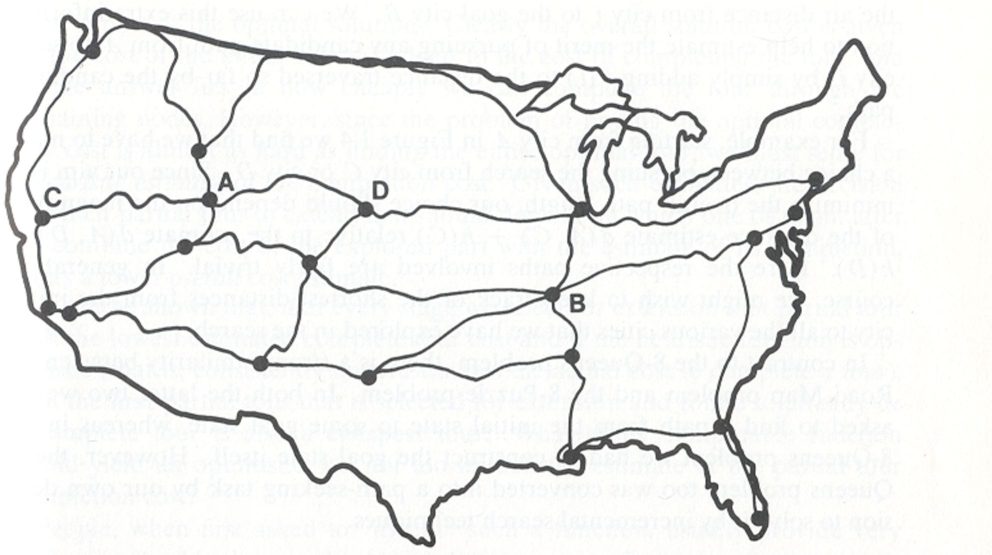

Figure 1.4
A road map, illustrating the use of heuristics in seeking the shortest path from A to B.

1.1.3 The Road Map Problem

Given a road map such as the one shown in Figure 1.4, we want to find the shortest path between city A and city B assuming that the length of each road on the map is specified by a label next to it.

If we are given an actual road map, then in looking for possible paths we would immediately rule out as unpromising those roads that lead away from the general direction of the destination. For example, the fact that city C, unlike city D, lies way out of the natural direction from A to B appears obvious to anyone who glances at the map in Figure 1.4, and so paths going through D will be explored before those containing C. On the other hand, if instead of the map, we are only given a numerical table of the distances $d(i, j)$ between the connected cities, the preference of D over C is not obvious at all; all cities situated at the same distance from A would appear equally meritorious. What extra information does the map provide that is not made explicit in the distance table?

One possible answer is that the human observer exploits vision machinery to estimate the **Euclidean distances** in the map and, since the air distance from D to B is shorter than that between C and B, city D appears as a more promising candidate from which to launch the search. In the absence of a map, we could attempt to simulate these visual computations by supplementing the distance table with extra data of equivalent informational value. For example, because we can easily compute air distances between cities from their coordinates, we can supplement the road-distance table with a heuristic function $h(i)$, which is the air distance from city i to the goal city B. We can use this extra information to help estimate the merit of pursuing any candidate path from A to some city i, by simply adding $h(i)$ to the distance traversed so far by the candidate path.

For example, starting from city A in Figure 1.4 we find that we have to make a choice between pursuing the search from city C or city D. Since our aim is to minimize the overall path length, our choice should depend on the magnitude of the distance estimate $d(A, C) + h(C)$ relative to the estimate $d(A, D) + h(D)$. Here the respective paths involved are fairly trivial. In general, of course, we might wish to keep track of the shortest distances from the initial city to all the various cities that we have explored in the search.

In contrast to the 8-Queens problem, there is a strong similarity between the Road Map problem and the 8-Puzzle problem. In both the latter two we are asked to find a path from the initial state to some goal state, whereas in the 8-Queens problem we had to construct the goal state itself. However, the 8-Queens problem too was converted into a path-seeking task by our own decision to solve it by incremental search techniques.

1.1.4 The Traveling Salesman Problem (TSP)

Here we must find the cheapest tour, that is, the cheapest path that visits every node once and only once, and returns to the initial node, in a graph of N nodes with each edge assigned a nonnegative cost. The problem is usually stated for a complete graph, that is, one in which every node is linked to each of the other nodes.

The TSP is known to belong to a class of problems called **NP-hard** (Garey and Johnson, 1979) for which all known algorithms require an exponential time in the worst case. However, the use of good **bounding functions** often enables us to find the optimal tour in much less time. What is a bounding function?

Consider the graph in Figure 1.5 where the two paths marked ABC and AED represent two partial tours currently being considered by the search procedure. Which of the two, if properly completed to form a tour, is more likely

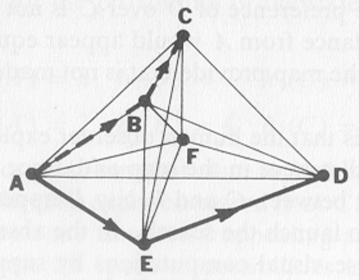

Figure 1.5
Two partial paths as candidates for solving the Traveling Salesman problem.

to be a part of the optimal solution? Clearly the overall solution cost is given by the cost of the initial subtour added to the cost of completing the tour, and so the answer lies in how cheaply we can complete the tour through the remaining nodes. However, since the problem of finding the optimal completion cost is almost as hard as finding the entire optimal tour, we must settle for a *heuristic estimate* of the completion cost. Given such estimates, the decision of which partial tour to extend first should depend on which one of them, after we combine the cost of the explored part with the estimate of its completion, offers a lower *overall* cost estimate.

It can be shown that, if at every stage we select for extension that partial tour with the lowest estimated complete tour cost and if the heuristic function is optimistic (that is, consistently underestimates the actual cost to complete a tour), then the **first** partial tour that is selected for extension and found to already be a complete tour is also a **cheapest** tour. What easily computable function would yield an optimistic, yet not too unrealistic, estimate of the partial tour completion cost?

People, when first asked to "invent" such a function, usually provide very easily computable, but much too simplistic answers. For example, as an estimate of the cost to complete a partial tour, take the cost of the edge from the current end of the partial tour to the initial node. As a marginally more complex example, take the cheapest two-edged path from the current end back to the initial node. These functions, although optimistic, grossly underestimate the completion cost.

Upon deeper thought, more realistic estimates are formed, and the two that have received the greatest attention in the literature are:

1. The cheapest **second degree graph** going through the remaining nodes;
2. The **minimum spanning tree** (MST) through all remaining nodes (Held and Karp, 1971)

The first is obtained by solving the so-called **optimal assignment problem** using on the order of N^3 computational steps, whereas the second requires on the order of N^2 steps.

That these two functions provide optimistic estimates of the completion cost is apparent when we consider that completing the tour requires the choice of a path going through all unvisited cities. A path is both a special case of a graph with degree 2 and also a special case of a spanning tree. Hence, the set of all completion paths is included in the sets of objects over which the optimization takes place, and so the solution found must have a lower cost than that obtained by optimizing over the set of paths only. Figure 1.6(a) shows the shape of a graph that may be found by solving the assignment problem instead of completing subtour AED. The completion part is not a single path but a collection of loops. Figure 1.6(b) shows a minimum-spanning-tree completion of subtour AED. In this case the object found is a tree containing a node with a degree higher than 2.

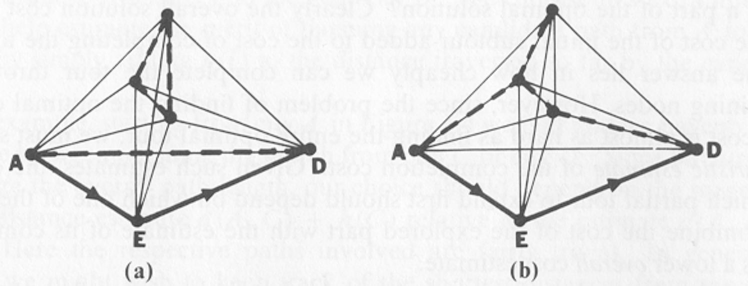

Figure 1.6
Two methods of optimistically estimating the completion cost of candidate $A \to E \to D$ in the Traveling Salesman problem of Figure 1.5. Part (*a*) The shortest degree-2 graph containing $A \to E \to D$. Part (*b*) The minimum spanning tree containing *A*, *D*, and all remaining (unlabeled) cities.

The shapes of the solutions found by these approximations may not resemble the desired tour, but the costs of these solutions may constitute good optimistic estimates of the optimal tour. Therefore they can be used as heuristics for assessing the relative merit of candidate subtours, thus improving the search efficiency. One of the questions addressed later in Chapter 6 is how the accuracy of such estimates affects the complexity of the search.

1.1.5 The Counterfeit Coin Problem

We are given 12 coins, one of which is known to be heavier or lighter than the rest. Using a two-pan scale, we must find the counterfeit coin and determine whether it is light or heavy in no more than three tests.

To solve this problem we must find a weighing strategy that guarantees that, regardless of the tests' outcomes, the counterfeit coin can be identified and characterized in at most three weighings. By **strategy** is meant a prescription of what to weigh first, what to weigh second for each possible outcome of the first weighing, what to weigh third for each possible outcome of each second weighing, and finally an indication of which coin is counterfeit and whether it is heavy or light for each possible outcome of each of the third weighings. In composing such a strategy we can try to construct it a piece at a time, much as we tackled the 8-Queens problem, by choosing a first weighing, then choosing a second weighing for each outcome, et cetera.

This problem can be solved by an exhaustive enumeration of all decisions and all possible test outcomes. The role of heuristics, however, is to focus attention on the most promising weighing strategies and to attempt to find a solution *without* exploring all possible strategies.

Consider the choice between the following two alternatives:

1. Start by weighing a pair of coins (one in each pan).
2. Start by weighing eight coins (four in each pan).

The second alternative appears superior even to a novice problem solver. The reason is that the first alternative has a high probability of ending up with a balanced scale, leaving us with the hard problem of finding a counterfeit coin among ten suspects using only two weighings. The second alternative, on the other hand, leads to more manageable problems: four suspects in case the scale balances and eight suspects each known to be "not-heavy" or "not-light" in case the scale tips.

As in the 8-Puzzle, the decision to defer the exploration of the first alternative in favor of the second can be obtained from a heuristic assessment of the effort required to complete the strategy from the first alternative relative to that of completing it from the second. For example, the base-3 logarithm of the number of remaining possibilities may provide an optimistic estimate for this effort, since every test may result in one of three possible outcomes: scale balances, scale tips to the right, and scale tips to the left.

Note, however, that every action in this problem requires an assessment of not one but three situations, one for each outcome of the balance. The merit of any action, therefore, should be computed by some combination of the merits assigned to the three situations created by that action. The **combination rule** depends on the problem statement. In our example, since the task is to be accomplished in *not more* than three weighings, the appropriate rule would be to assess any action by the effort required to solve the *hardest* of the three problem situations that may result from that action. On the other hand, if we were given an upper limit on the *average* number of weighings, a more appropriate rule would be to take the *expected value* of the effort estimates associated with those three problem situations.

Another difference between the 8-Puzzle and the Counterfeit Coin problem is that in the former we were only concerned with the issue of which action to explore next. In the Counterfeit Coin problem, we also have to address the following question: given that we decide to explore alternative 2 before alternative 1, which **subproblem** deserves our attention next; the easy one, containing four suspected coins, or the hard one, with the eight coins? Moreover, once we shed additional light on the chosen subproblem, should we then attend to its sibling or reevaluate the parent action to test whether it is still more promising than action 1 which we left unexplored? Deciding control issues of this kind is another role heuristics play in computer-based problem solving.

SUMMARY

Each of the five preceding example problems presents us with repetitive choices among a multitude of exploration paths that might be taken to reach a solution. In each case, we have exhibited a heuristic function that serves to indicate which among all those paths seem the most likely to lead to a solution. The heuristics are simple to calculate relative to the complexity of finding a solution and, although they do not necessarily always guide the search in the correct direction, they quite often do. Moreover, whenever they do not, we can resort to some recovery scheme to resume the search from a different point. The key is that these heuristics save time by avoiding *many* futile explorations.

But how do you find a good heuristic, and when you have found one, how should you use it most effectively and how would you evaluate its merit?

The main objective of this book is to try to answer such questions and to quantify the characteristics of heuristics that produce the most beneficial results.

1.2 SEARCH SPACES AND PROBLEM REPRESENTATIONS

Each of the five examples discussed in the preceding section represents a class of problems with unique characteristics that may require a different representation or a different problem-solving strategy. Some of these characteristics are classified and discussed in the following sections.

1.2.1 Optimizing, Satisficing, and Semi-Optimizing Tasks

The Road Map and the Traveling Salesman problems differ from the other three examples in Section 1.1 in that the former require **optimization;** the path or tour sought had to be at least as cheap as any other feasible path or tour. In all five cases the problem statement delineated a space of candidate objects in which a solution is to be found. The type of objects sought varied from problem to problem; for the 8-Queens problem we were seeking a board configuration with certain properties, for the 8-Puzzle a sequence of moves, a path for the Road Map problem, a tour for the TSP, and a strategy of actions for the Counterfeit Coin problem. In the two optimization problems, however, the objective has been not merely to exhibit a formal object satisfying an established set of criteria but also to ascertain that it possesses qualities unmatched by any other object in the candidate space. By contrast, what was desired in the other three problems was to discover any qualified object with as little search effort as possible, a task we call **satisficing** (originally coined by H. Simon, 1956.)

Most problems can be posed both as constraint-satisfaction and as optimization tasks. For example, the 8-Queens problem can be stated as follows: find the cheapest placement of eight queens on a chess board where each capturable queen introduces a cost of one unit. Posed in this manner, the problem generally becomes harder since we do not know *a priori* whether a zero-cost solution exists. Moreover, even if we prove that such a solution does not exist, the work is not over yet; we still have to search higher cost configurations for the optimal solution. However, heuristic information pertaining to the *quality*

of the solution may also be helpful in reducing the effort of finding one, even in cases in which the quality of the solution is of minor importance. In theorem proving, for example, we would normally be satisfied with the first available proof, however inelegant. Nevertheless, by focusing the search effort toward finding the *shortest* proof, we also install safeguards against aimless explorations in a large space of possibilities, and this normally leads to a smaller search effort in finding a proof. This beneficial principle, which we call **small-is-quick**, plays a major role in heuristic methods for satisficing search. In a later chapter we also demonstrate that algorithms driven by an auxiliary optimization objective (smallness) may exhibit the most efficient (quickest) search even when we know in advance that only one solution exists, or that just any solution will do.

Often the difference in complexity between an optimization task and its satisficing counterpart is very substantial. The TSP is an example of such a case; finding some tour through a set of cities is trivial, but finding an optimal tour is NP-hard. In such cases, one is often forced to relax the optimality requirement and settle for finding a "good" solution using only a reasonable amount of search effort. Whenever the acceptance criterion tolerates a neighborhood about the optimal solution, a **semi-optimization** problem ensues.

Semi-optimization problems fall into one of two categories. If the boundaries of the acceptance neighborhood are sharply defined (e.g., requiring that the cost of the solution found be within a specified factor of the optimal cost), the task is called **near-optimization.** If we further relax the acceptance criterion and insist only that the solution be near optimal with sufficiently high probability, we have an **approximate-optimization** task on our hands.

Most practical problems are of a semi-optimization type, requiring some reasonable balance between the quality of the solution found and the cost of searching for such a solution. Moreover, because the search effort required for many combinatorial optimization problems can easily reach astronomical figures, relaxing optimality is an economic necessity. The basic problem in handling a semi-optimization task is to devise algorithms that guarantee bounds on both the search effort and the extent to which the optimization objective is compromised. A more ambitious task would be to equip such an algorithm with a set of adjustable parameters so that the user can meet changes in emphasis between cost and performance by controlling the tradeoff between the quality of the solution and the amount of search effort.

1.2.2 Systematic Search and the Split-and-Prune Paradigm

Since every problem-solving activity can be regarded as the task of finding or constructing an object with given characteristics, the most rudimentary requirements for computer-based problem solving are:

1. A **symbol structure** or **code** which can represent each candidate object in the space

2. Computational **tools** that are capable of **transforming** the encoding of one object to that of another in order to scan the space of candidate objects systematically
3. An effective method of **scheduling** these transformations so as to produce the desired object as quickly as possible

In the nomenclature common to the artificial intelligence literature, these three components of problem-solving systems are known as

1. The **database**
2. The **operators** or **production rules**
3. The **control strategy**

A control strategy is said to be **systematic** if it complies with the following colloquially stated directives:

1. Do not leave any stone unturned (unless you are sure there is nothing under it).
2. Do not turn any stone more than once.

The first requirement, called **completeness,** guarantees that we do not miss the opportunity to generate an encoding that represents the desired solution if one exists. The second protects us from the inefficiency of repetitious computations.

Let us first examine what kind of objects the code must be capable of **representing.** Obviously we should have the capabilities of representing those objects that we suspect may themselves constitute the desired solutions to the problem at hand, such as board configurations, paths and strategies. But a code limited to expressing only the final objects will severely restrict the flexibility of search strategies. Although we would still be able to comply with the two basic directives of systemization, such an encoding will prevent us from taking advantage of some useful structural properties of the problem domain.

To illustrate this point, imagine the task of searching for the street address of a telephone number in a typical telephone directory. If we are given only the number, we have no alternative but to scan the book exhaustively. By contrast, if the owner's name is also given, the address can be located in just a few operations. In both cases, the book contains a code for every candidate object, but in the latter case we also use **codes for large subsets of objects.** If we open the directory at a page past the name we seek, we immediately recognize that all the names in the following pages could not qualify for a solution and this entire subset of objects is removed from future considerations by turning the pages back or by placing a book marker in the page just opened. The ability to eliminate or prune large chunks of the candidate space does not exist when we are given just the phone number. True, we can still encode large subsets of objects which we may wish to eliminate, e.g., the set of all phone numbers greater than the one sought, but there is no way to guarantee that future transformations (i.e., page turning operations) will not regenerate elements from the subset we wished eliminated.

This simple example illustrates the need to equip our encoding scheme with facilities to express and manipulate not merely individual solution objects but also **subsets of potential solutions.** Moreover, if we are permitted to apply code-transformations that take us from one subset to another, we should also apply the two requirements of systemization to subsets of potential solutions. The first requirement simply states that all individuals must be included in the collection of subsets expressible in our code, and that each individual be reachable by some operations on the subsets to which it belongs. The second requirement, on the other hand, insists that if subset S_1 has been eliminated from consideration, subsequent operations on other subsets will not generate any member of S_1. Putting these two requirements together implies that the only transformation on subsets we should permit is that of **splitting**. Indeed, by subjecting a given subset to repeated splitting operations, we may eventually reach any individual member of the subset and, at the same time, we cannot generate a new individual not originally included in the parent subset.

Before elaborating on additional features of this **split-and-prune** method, it is instructive to illustrate its basic building blocks in the context of the problems presented in Section 1.1.

In the **8-Queens** problem the individual objects capable of qualifying as solutions can be taken to include all 8×8 board configurations containing eight queens. Upon further reflection, we may narrow down the candidate space by considering only those configurations containing one queen in every row. A partially filled board, like that shown in Figure 1.1, constitutes a code for a *subset of potential solutions* in that it stands for all board configurations whose top three rows coincide with those filled in the partially specified arrangement. Adding a queen to the fourth row further *refines* the parent subset as it constrains the elements represented by the resulting subset to comply with a board of four instead of three specified rows. The three candidate search paths (represented by the letters A, B, and C in Figure 1.1) represent *splitting* of the parent set into a group of three mutually exclusive subsets, each suspected of containing a solution. Note that the subsets corresponding to the remaining five cells on the fourth row need not be considered since they could not possibly contain a legitimate solution.

Our ability to detect and eliminate or **prune** such large subsets of objects at such an early stage of the search is what makes the 8-Queens problem easy to solve. This ability was facilitated partly through the splitting method and partly by the fortunate choice of code. The splitting method guarantees that subsequent operations (i.e., adding new queens to the board) will not regenerate any member of subsets rejected earlier and that no potential solution is overlooked in this process. To appreciate the power of the split-and-prune method, the reader may do well to examine an alternative representation for the 8-Queens problem, one that does not allow splitting. Imagine, for example, that someone decides to solve this puzzle by placing eight queens in some random initial position and then perturb the original position iteratively by moving one queen at a time. In this representation we do have a code for individual objects but not a code for representing subsets of objects. Consequently, as in the case of trying to find an address in a telephone directory without a name, we can reject individual objects but not large chunks of objects.

The availability of an effective code for subset pruning was also significant in facilitating an easy solution to the 8-Queens problem. Choosing a partially filled board as a code for subsets of potential solutions was fortunate because it provided an early detection of some subsets containing no solution (e.g., if two queens in the partially filled board attack each other). The subset code itself, in this case, already possesses qualities of the final solution upon which the final acceptance tests can be performed, and hence early detections of some dead-end subsets are obtained quickly. To appreciate the importance of code-selection, let us now consider an example in which subset splitting is possible, but the code available for representing subsets of potential solutions does not facilitate an early detection of dead ends.

In the **8-Puzzle**, the individual search objects are sequences of legal transformations starting with the initial configuration s. The solution objects are those sequences that lead from s to the goal configuration. A given chain of moves encodes the subset of potential solutions, which includes all those sequences beginning with the chosen chain. Each time we add a new move to a given chain, we narrow the subset of solutions represented by the chain, and since the subsets represented by all legal moves applicable to a given position are mutually exclusive, we are justified in calling the act of listing the consequences of these moves "splitting."

Again, the splitting method theoretically permits us to prune any subset that is proved hopeless, but in the 8-Puzzle no subset is truly hopeless. Subsequent operations can always undo previous operations and place us back on a path to the goal if such a path exists. By contrast, if we change the original problem specifications and require that the goal state be achieved in *no more than K moves*, some subsets of potential solutions do become hopeless. In fact, every sequence of K moves that does not lead to the goal position can simply be discarded because it cannot be turned into a legitimate solution by adding more moves to it. But can we predict this dead-endness early enough simply by looking at an initial segment of such a sequence? Unfortunately we know of no code that would emit early warning signals so vividly as those obtained in the 8-Queens problem. To produce such early warnings, we must, instead, resort to heuristic information that is not explicitly mentioned in the initial problem statement. For example, using heuristic functions such as h_1 and h_2, we can prune away sequences of length shorter than K. Every sequence of $K - h_1(n)$ (or $K - h_2(n)$) moves not leading to the goal can now be discarded, where n is the last position resulting from the sequence.

The use of heuristic functions imposes additional requirements on the **format** of our chosen **code.** We now must make sure that the code we choose for representing candidate subsets not only identifies these subsets unambiguously,

but also facilitates the easy computation of such heuristics. For example, if we choose to codify subsets only by the sequence of splits (or moves) that produced them, such as "up, left, right, down, . . . ," then the computation of h_1 will be very cumbersome. If, on the other hand, we also maintain a (redundant) description of the **state** prevailing after such a sequence, the computation becomes easy. We simply count the number of tiles by which the resulting configuration differs from the goal configuration. And if we also maintain a list of the differing tiles, the computation becomes easier still.

The **Counterfeit Coin** problem introduces a new element. The problem statement requests an identification of a weighing **strategy** that meets certain conditions. A strategy is not simply a sequence of actions but a prescription for choosing an action in response to any possible external event such as an outcome of a test, an action of an adversary, or the output of a complicated computational procedure. Thus, whereas the search objects in the first four examples of Section 1.1 were represented as **paths,** those in the Counterfeit Coin problems are **trees.** A solution to the problem is a tree with depth of at most three which includes all possible outcomes to any of the chosen tests. In addition, each path in the tree must represent a sequence of tests and outcomes that identifies the counterfeit coin unambiguously. To handle such problems, we need, of course, a code for representing subsets of potential solution strategies. A convenient code for such problems is provided by AND/OR graphs, as we will see in Section 1.2.4.

The idea of viewing problem solving as a process of repetitive splitting (also called **branching** or **refinement**) of subsets of potential solutions became popular in operations research and is the basic metaphor behind the celebrated branch-and-bound method (Lawler and Wood, 1966). Artificial intelligence literature rarely mentions this metaphor, preferring to describe problem solving as a **"generate-and-test"** process (Newell and Simon, 1972), **creating** new objects rather than **eliminating** objects from preexisting sets. An artificial intelligence researcher would view a partially developed path, for example, as a building block in the construction of a more elaborate complete solution that will *contain* that path. An operations researcher, on the other hand, views the partial path as (a code for) a huge subset of potential solutions that should eventually be trimmed down to a single element. Note that the use of the word "contain" is reversed in the latter paradigm; the partial path is viewed as an elaborate subset that in itself *contains* complete solutions as structureless points.

The difference between these two approaches is not accidental but reveals a sharp difference of emphasis between the two communities. The split-and-prune paradigm emphasizes the commonality of problem-solving methods over many problems, whereas the object creation paradigm focuses on the unique structure featured by a specific symbolic representation of a given problem. The operations research approach is particularly effective in establishing theoretical guarantees such as completeness and optimality. The artificial intelligence community, on the other hand, has always emphasized the heuristic and implementational aspects of problem solving and its parity with human style over its mathematical foundations. Indeed, in human terms, the acts of adding a new queen to the board (in the 8-Queens problem), extending a tour to pass through one more city (in the Traveling Salesman problem), or deriving a new lemma in theorem proving are definitely perceived more as acts of *creation* and *construction* than acts of *pruning* or *deletion*.

We will use both paradigms interchangeably. The generate-and-test metaphor will be used to *describe* the various steps used in search procedures, whereas the split-and-prune metaphor will mainly be invoked to *justify* them.

1.2.3 State-Space Representation

Returning to the method of subset splitting, the three basic components needed for the method to work are:

1. A symbol structure called a **code** or a **database** that represents subsets of potential solutions (henceforth called **candidate subsets**)
2. A set of operations or **production rules** that modify the symbols in the database, so as to produce more refined sets of potential solutions
3. A search procedure or **control strategy** that decides at any given time which operation is to be applied to the database

If storage space permits, the database should include codes for several candidate subsets simultaneously. This allows the control strategy to decide which subset would be the most promising to split. When memory space is scarce, the database at any given time may contain a code of only one subset with provisions for backtracking and generating alternate refinements of the parent subset in case a solution cannot be found by a given sequence of refinements.

We may now impose **additional requirements on the format** of the code assigned to a given candidate subset. In principle, since a given sequence of refinement operators corresponds to one and only one candidate subset, that sequence in itself could constitute a sufficient parsimonious code for the resultant subset. We have seen in the 8-Puzzle example, however, that such a code would be very awkward. Although this code is adequate for identifying the candidate subset and for protecting against violations of the two rules of systematic search, it tells us nothing about the nature of the problem which remains to be solved (i.e., finding a solution among the members of the encoded subset). The sequence *s*-up-left-up-right, for example, uniquely specifies one tile-configuration in the 8-Puzzle but contains no explicit information regarding the resultant configuration, its proximity to the goal, whether it has been encountered before in some other candidate subsets, or even which operators are next applicable.

For these reasons we find it convenient to include in the code assigned to each candidate subset additional information that **explicitly defines the remaining subproblem** presented by the subset. The code specifying this additional in-

formation is called a **state**. The set of all subproblems obtainable by executing some sequence of refinement operators from a given position is called a **state-space**. If we connect the elements of this space by arcs labeled by the appropriate refinement operators, we obtain a **state-space graph**.

To illustrate this, a typical candidate subset code for the Traveling Salesman problem might look as follows:

$$\underbrace{A \to B \to C \to}_{\text{candidate subset}} \underbrace{D \to \{E, F\} \to A}_{\text{state}}$$

The first sequence, $A \to B \to C \to D$, identifies the subset of all tours that begin at city A and then proceed through cities B, C, and D. The part marked "state" specifies the problem of completing the tour: find a path beginning with D, going once through each city in the set $\{E, F\}$ and finally terminating at A. Note that the former is both necessary and sufficient, whereas the latter part is **redundant** and **incomplete**; that is, a solution of the subproblem represented by the state will not identify the entire tour. A subtour description, on the other hand, completely specifies the remaining problem.

The advantages of keeping the state code explicit are apparent. It is this portion of the code that expedites the computation of the heuristic estimates discussed in Section 1.1.4. Moreover, suppose at a certain state of the search, the database also contains the code:

$$\underbrace{A \to C \to B \to}_{\text{candidate subset}} \underbrace{D \to \{E, F\} \to A}_{\text{state}}$$

Although the candidate subset represented by this code is different from the preceding one, their states are identical, so we can determine that one subset of candidates is superior to the other without solving the state subproblem. Simply by comparing the costs of the two paths from A to D, we can prune the inferior subset from future consideration. This type of pruning is called **pruning by dominance** (Ibaraki, 1977) and can be shown to result in substantial savings of search effort.

1.2.4 Problem-Reduction Representations and AND/OR Graphs

We have seen that both constraint-satisfaction problems and sequence-seeking problems can be represented as tasks of finding paths in a state-space graph. Other problems, like the strategy-seeking task in the Counterfeit Coin problem, may be more conveniently represented by a technique called **problem-reduction** representation. The unique feature of this class of problems is that each residual problem posed by some subset of candidates can be viewed as a **conjunction of several subproblems** that may be solved independently of each other, may require specialized treatments, and should, therefore, be represented as separate entities.

To illustrate this point let us return to the Counterfeit Coin problem of Section 1.1.4 and assume that we have chosen the first alternative, that is, start by weighing a single pair. If we are lucky and the balance tips one way or another, we are left with the easy problem of finding which of the two suspects is the counterfeit coin, knowing that the remaining ten coins are honest. This simple problem can, of course, be solved in a single step, weighing one of the suspects against one of the honest coins. The thing to notice, though, is that this two-suspect problem can be **solved in isolation**, independently of the problems that may ensue in case the balance remains neutral. Moreover, the same simple problem will appear frequently at the end of many elaborate strategies, and so, if we recognize it as an individual entity deserving an individual code, the solution to this subproblem can be found once, remembered, and shared among all strategies that lead to it.

This suggests that we treat **subproblems as individual nodes** in some graph, even though none of these subproblems alone can constitute a complete solution to our original problem. In other words, since every action in our example introduces a triplet of subproblems, all of which must be solved, we prefer to represent the situation resulting from an action not as a single node containing a list of subproblems but rather as a triplet of individual nodes, each residing atop its own specialized set of candidate solution strategies. Figure 1.7 illustrates this problem-reduction representation, contrasting it with the state-space representation of Figure 1.8.

Note that although the individual subproblems are assigned separate nodes, they are still bound by the requirement that *all* must be solved before the parent problem is considered solved. The purpose of the curved arcs in Figure 1.7 is to remind us of this fact. Thus the search-space graph in this representation consists of two types of links:

1. Links that represent alternative approaches in handling the problem node from which they emanate.
2. Links that connect a parent problem node to the individual subproblems of which it is composed.

The former are identical to the links in state-space graphs and are called **OR links**. The latter are unique to problem-reduction and are called **AND links**.

All nodes in an AND/OR graph, like those in a state-space graph, represent subproblems to be solved or subgoals to be achieved, and the start node represents the specification of the original problem. Likewise, each path emanating from the start node identifies a unique candidate subset containing those strategies that comply with the sequence of event-response pairs specified by the path. However, unlike paths in state-space, a solution to the problem represented by the tip node of such a path is no longer sufficient for synthesizing a solution to the original problem. The complete solution is represented by

an AND/OR subgraph, called a **solution graph**, which has the following properties:

1. It contains the start node.
2. All its terminal nodes (nodes with no successors) are primitive problems, known to be solved.
3. If it contains an AND link L, it must also contain the entire group of AND links that are siblings of L.

An AND/OR graph and two of its solution graphs are illustrated in Figure 1.9.

It is sometimes convenient to regard AND/OR graphs as generalizations of ordinary graphs called **hypergraphs**. Instead of links that connect pairs of

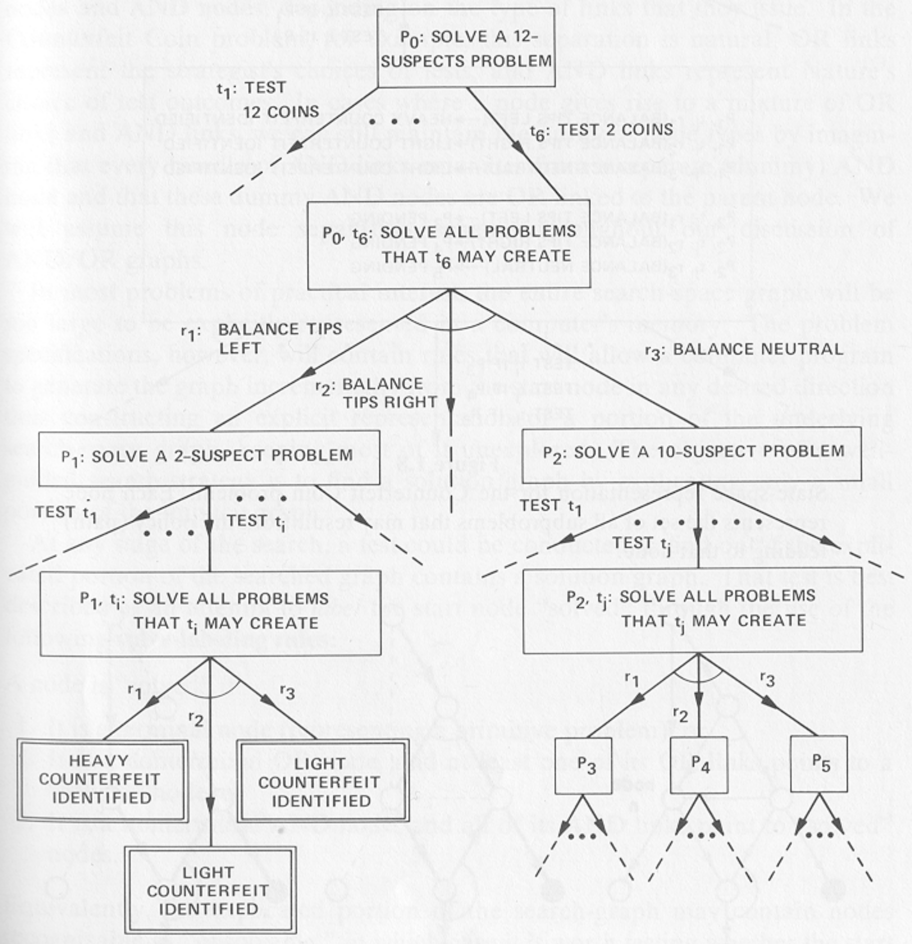
P_0: SOLVE A 12-SUSPECTS PROBLEM
t_1: TEST 12 COINS
t_6: TEST 2 COINS
P_0, t_6: SOLVE ALL PROBLEMS THAT t_6 MAY CREATE
r_1: BALANCE TIPS LEFT
r_2: BALANCE TIPS RIGHT
r_3: BALANCE NEUTRAL
P_1: SOLVE A 2-SUSPECT PROBLEM
P_2: SOLVE A 10-SUSPECT PROBLEM
TEST t_1
TEST t_j
P_1, t_j: SOLVE ALL PROBLEMS THAT t_j MAY CREATE
P_2, t_j: SOLVE ALL PROBLEMS THAT t_j MAY CREATE
r_1
r_2
r_3
HEAVY COUNTERFEIT IDENTIFIED
LIGHT COUNTERFEIT IDENTIFIED
LIGHT COUNTERFEIT IDENTIFIED
P_3
P_4
P_5

Figure 1.7
Problem-reduction representation for the Counterfeit Coin problem. Each node stands for one subproblem, and the curved arcs indicate which sets of subproblems must be solved conjunctively to make up a complete solution.

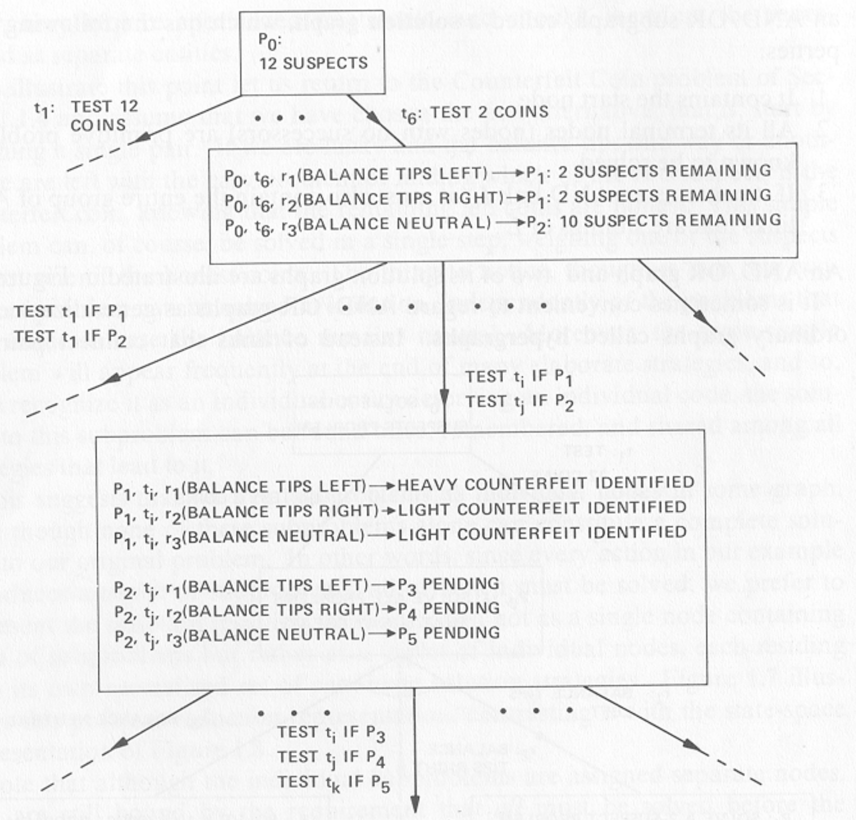
P0: 12 SUSPECTS
t1: TEST 12 COINS
t6: TEST 2 COINS
P0, t6, r1(BALANCE TIPS LEFT)→P1: 2 SUSPECTS REMAINING
P0, t6, r2(BALANCE TIPS RIGHT)→P1: 2 SUSPECTS REMAINING
P0, t6, r3(BALANCE NEUTRAL)→P2: 10 SUSPECTS REMAINING
TEST t1 IF P1
TEST t1 IF P2
TEST ti IF P1
TEST tj IF P2
P1, ti, r1(BALANCE TIPS LEFT)→HEAVY COUNTERFEIT IDENTIFIED
P1, ti, r2(BALANCE TIPS RIGHT)→LIGHT COUNTERFEIT IDENTIFIED
P1, ti, r3(BALANCE NEUTRAL)→LIGHT COUNTERFEIT IDENTIFIED
P2, tj, r1(BALANCE TIPS LEFT)→P3 PENDING
P2, tj, r2(BALANCE TIPS RIGHT)→P4 PENDING
P2, tj, r3(BALANCE NEUTRAL)→P5 PENDING
TEST ti IF P3
TEST tj IF P4
TEST tk IF P5

Figure 1.8

State-space representation for the Counterfeit Coin problem. Each node represents the set of all subproblems that may result from the policy (path) leading to that node.

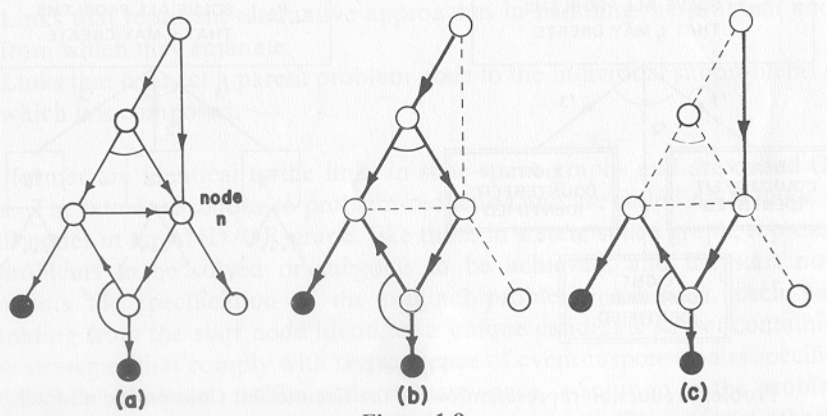

Figure 1.9

An AND/OR graph (*a*) and two of its solution graphs (*b*) and (*c*). Terminal nodes are marked as black dots.

nodes, hypergraphs may contain hyperlinks, called **connectors**, connecting a parent node with a *set* of successor nodes. If we regard each group of AND links as a single connector or hyperlink, the task of finding a solution graph amounts to that of finding a **hyperpath** between the start node and the set of primitive terminal nodes. This analogy is helpful in making concepts and techniques developed for path-finding problems applicable to problem-reduction representations.

In general, an AND link and an OR link may simultaneously point to the same node in an AND/OR graph, such as the node marked (n) in Figure 1.9. In this figure, however, each node issues either OR links or AND links, but not both. When this condition holds, it is convenient to label the nodes **OR nodes** and **AND nodes**, depending on the type of links that they issue. In the Counterfeit Coin problem, for example, this separation is natural; OR links represent the strategist's choices of tests, and AND links represent Nature's choice of test outcomes. In cases where a node gives rise to a mixture of OR links and AND links, we can still maintain the purity of node types by imagining that every bundle of AND links emanates from a separate (dummy) AND node and that these dummy AND nodes are OR linked to the parent node. We will assume this node separation scheme throughout our discussion of AND/OR graphs.

In most problems of practical interest, the entire search-space graph will be too large to be explicitly represented in a computer's memory. The problem specifications, however, will contain rules that will allow a computer program to generate the graph incrementally from the start node in any desired direction thus constructing an explicit representation of a portion of the underlying search-space graph, keeping most of it unexplored. The objective of a well-guided search strategy is to find a solution graph by explicating only a small portion of the implicit graph.

At any stage of the search, a test could be conducted to find out if the explicated portion of the searched graph contains a solution graph. That test is best described as an attempt to *label* the start node "solved" through the use of the following **solve-labeling rules:**

A node is "solved" if:

1. It is a terminal node (representing a primitive problem); or
2. It is a nonterminal OR node, and at least one of its OR links points to a "solved" node, or
3. It is a nonterminal AND node, and all of its AND links point to "solved" nodes.

Equivalently, the explicated portion of the search-graph may contain nodes recognizable as "unsolvable," in which case it is worth testing whether the start node still stands a chance of getting solved by attempting to label it "unsolvable." The labeling rules for "unsolvable" status are similar to those of "solve" except that the roles of AND and OR links are interchanged.

1.2.5 Selecting a Representation

The selection of a representation to fit a given problem is sometimes dictated by the problem specifications and sometimes is a matter of choice.

Generally speaking, problem-reduction representations are more suited to problems in which the final solution is conveniently represented as a tree (or graph) structure. State-space representations are more suited to problems in which the final solution can be specified in the form of a path or as a single node.

Strategy-seeking problems are better represented by AND/OR graphs, where the AND links represent changes in the problem situation caused by external, **uncontrolled conditions** and the OR links represent alternative ways of reacting to such changes. In our Counterfeit Coin problem, the uncontrolled condition was Nature's choice of the weighing results. In games, those conditions are created by the legal moves available to the adversary. In program-synthesis, they consist of the set of outcomes that may result from applying certain computations to unspecified data. In general, a strategy-seeking problem can be recognized by the **format required for presenting the solution.** If the solution is a prescription for responding to observations, we have a strategy-seeking problem on our hands, and an AND/OR graph will be most suitable. If, on the other hand, the solution can be expressed as an unconditional sequence of actions or as a single object with a given set of characteristics, we have a path-finding or constraint-satisfaction problem and a state-space representation may be equally or more appropriate.

Another important class of problems suitable for AND/OR graph representations includes cases in which the solution required is a **partially ordered sequence** of actions. This occurs when each available operator replaces a single component of the database by a set of new components. Take, for example, the following **set-splitting** problem:

> Given a set of numbers, find the cheapest way to separate this set into its individual elements using a sequence of partitioning operators. Each operator partitions one subset into two parts, and the cost of each partition is equal to the sum of the elements in the partitioned subset.

It is not hard to show that the solution to this problem is given by the celebrated Huffman-Code algorithm (Horowitz and Sahni, 1978). However, the feature that makes this example worth mentioning here is that the two subsets created by each partitioning operator can later be handled *in any desired order*, without affecting the cost of the solution. Hence, it would be more natural to represent each candidate solution as an **action-tree**, rather than as an action sequence. Thus, although the solution sought is not a strategy in the strict sense of prescribing a policy of reacting to uncontrolled conditions, it is still in tree form, and, therefore, the problem is well suited for AND/OR graph representation.

A similar structure is possessed by **symbolic-integration** problems (Moses, 1971). Several legal transformations are available in this domain (e.g., integration by parts, long division, etc.) which will split the integrand into a sum of expressions to be integrated separately *in any order.* The set of applicable transformations will be represented as OR links emanating from the node representing the integrand, and the AND links represent individual summands within the integrand, all of which must eventually be integrated.

The tasks of **logical reasoning** and **theorem proving** also give rise to AND/OR structures. We begin with a set of axioms and a set of inference rules that allow us, in each step, to deduce a new statement from a subset of axioms and previously deduced statements. The new statement is added to the database, and the process continues until the desired conclusion (e.g., the theorem) is derived. The solution object pursued by the search is a plan specifying in each step which of the inference rules is to be applied to which subset of statements in the database and what the deduced statement is. This plan, again, is best structured as an **unordered tree** because, when a certain conclusion is derived from a given subset of statements, the internal order in which these statements were themselves derived is of no consequence as long as they reside in the database at the appropriate time. Thus, the solution structure is a tree, and hence, the appropriate search-space would be an AND/OR graph.

In certain cases both state-space and problem-reduction formulations appear equally appropriate and the choice of representation requires additional insight. Take, for instance, the 8-Puzzle (Section 1.1.2) and assume that we wish to formulate it in a problem-reduction representation. Whereas in the state-space approach each tile configuration stood for the problem of transforming the configuration into the goal state, we may attempt now to decompose this problem into a set of more elementary problems. We may regard each condition required to meet the goal criterion as a subproblem on its own right (e.g., tile x_1 on cell c_1, tile x_2 on cell c_2, etc.). However, the highly interdependent nature of these subgoals severely weakens the advantages of the problem-reduction approach. As soon as we accomplish some of these subgoals, we find that they must be violated (at least temporarily) in order to achieve the remaining subgoals. Moreover, the solution found for achieving a given subgoal is **context dependent** and can no longer be saved for future use as we have done in the Counterfeit Coin problem. We conclude, therefore, that a problem-reduction formulation is ineffective for the 8-Puzzle and that state-space is its most natural representation.

This conclusion, however, could not have been drawn by a superficial examination of the problem statement without a detailed analysis of the interactions among the goals. In some sequence-seeking problems, like the 8-Puzzle, the goals are interacting with each other, and yet a problem-reduction representation may substantially reduce the search for a solution. A classical example in this category is the **Tower of Hanoi**:

> There are n disks $D_1, D_2, \ldots, D_n$, of graduated sizes and three pegs 1, 2, and 3. Initially all the disks are stacked on peg 1, with D_1, the smallest, on top and D_n, the

largest, at the bottom as in Figure 1.10. The problem is to transfer the stack to peg 3 given that only one disk can be moved at a time and that no disk may be placed on top of a smaller one.

The problem can easily be represented in state-space by specifying the operators that are permissible from any given disk configuration. Alternatively, the problem may be represented as a list of goals to be achieved: disk D_1 on peg 3, disk D_2 on peg 3, . . . , disk D_n on peg 3. These goals are clearly not independent; placing D_1 initially on peg 3 hinders the transfer of D_2 there. However, this set of goals possesses a unique **linear hierarchy**: once the subset of goals concerning disks $D_k, D_{k+1}, \ldots, D_n$ are completed, the goals of relocating disks $D_1, D_2, \ldots, D_{k-1}$ can be accomplished without undoing the former. Whenever a goal set possesses such a linear structure, the key idea is to begin working on the most critical goal (i.e., setting disk D_n on peg 3).

Having decided to begin with this goal, we must first check whether this can be accomplished by solving a primitive problem. In looking up the list of primitive operators available in our problem and their applicability constraints, we should discover that setting disk D_n on peg 3 is indeed a primitive problem if only two conditions are met:

1. Disk D_n must be clear
2. Peg 3 must be empty.

These two conditions are not satisfied in the initial state and, therefore, they are

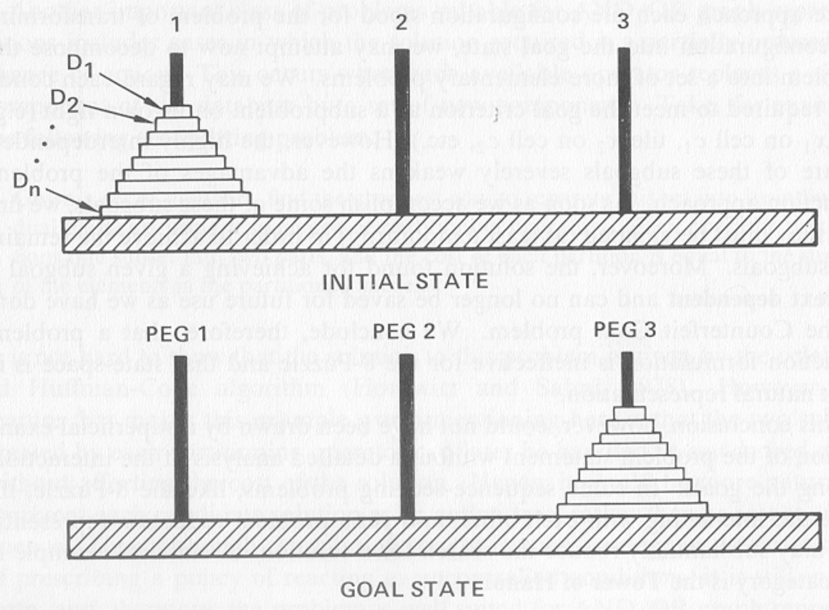

Figure 1.10
The Tower of Hanoi problem.

posited as **auxiliary subgoals,** which can also be formulated as the set of conditions: disk D_1 on peg 2, disk D_2 on peg 2, . . . , disk D_{n-1} on peg 2.

This auxiliary set of subgoals, together with the primitive problem "move D_n onto peg 3" and the set of unaccomplished goals remaining after the latter, now form a conjunction of three subproblems, which are represented by the first level of the tree in Figure 1.11. At this point the power of the problem-reduction representation can be brought into play; although the conjunction of the three subproblems implies an **ordered execution** of their solutions (from left to right), the **search** for these solutions may be conducted **in any order**. That is, we may begin searching for a solution of the rightmost subproblem prior to knowing the solution to the ones on its left; the latter provides no information that is necessary for the former. This property was not present in the 8-Puzzle and for that reason we could not benefit from the problem-reduction representation. In the Tower of Hanoi puzzle, on the other hand, we were able to reduce the original problem involving n disks to three subproblems, one primitive and two involving $n-1$ disks.

This reduction process can be applied to any nonprimitive subproblem until all the nodes in the tree become primitive and then, reading the leaf nodes left to right, we have a complete detailed sequence of elementary moves that constitutes a solution to the initial problem. The tree will undoubtedly be large (exponential in n) simply because the solution requires a large number of elementary moves. However, if we only wish to communicate a parsimonious description of the solution pattern without worrying about its details, we may take the first level of the tree as an **abstract plan** or a blueprint from which the detailed plan can be generated recursively if desired:

To transfer $S = \{D_1, \ldots, D_n\}$ from peg i to peg j, do the following:

1. If S contains a single disk, move that disk from i to j; otherwise
2. Transfer $\{D_1, \ldots, D_{n-1}\}$ from i to k.
3. Move D_n from i to j.
4. Transfer $\{D_1, \ldots, D_{n-1}\}$ from k to j.

The reasoning behind this problem-reduction method is called **means-end analysis**. It is the basis of a program called GPS, the General Problem Solver (Newell, Shaw, and Simon, 1960), subsequent automatic planning programs such as STRIPS (Fikes and Nilsson, 1971) and ABSTRIPS (Sacerdoti, 1974) and a conversational planning system named GODDESS (Pearl, Leal, and Saleh, 1982). The basic difference between this method and the state-space approach is its purposeful behavior: operators are invoked by virtue of their potential in fulfilling a desirable subgoal, and new subgoals are created in order to enable the activation of a desirable operator.

The success of this method depends, of course, on the availability of an ordered set of subgoals $G_1, G_2, \ldots, G_n$, such that the problem is solved when all subgoals are achieved and such that the set $G_1, \ldots, G_k$ can be accomplished

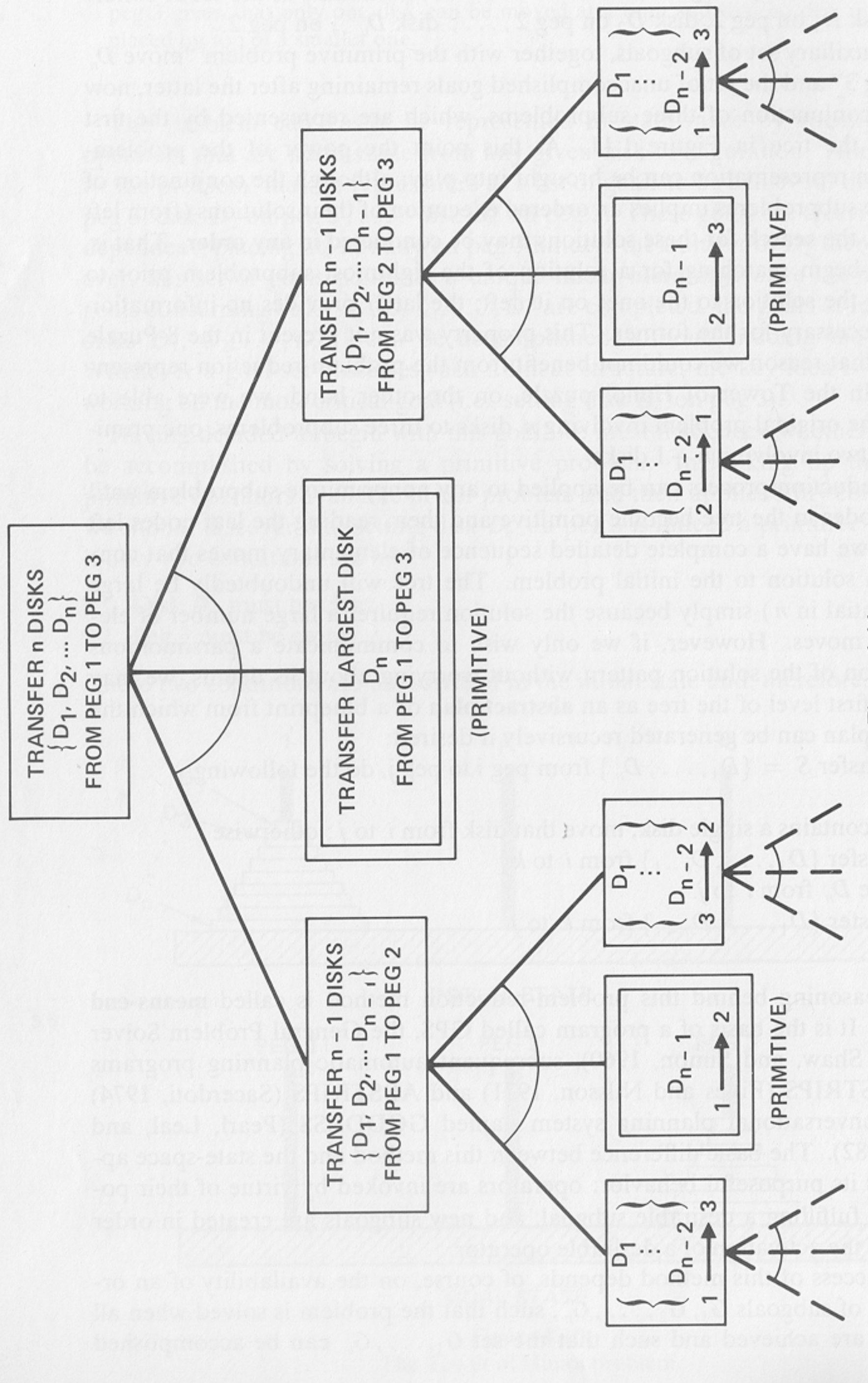

Figure 1.11
Problem-reduction representation of the Tower of Hanoi problem. The problem of transferring n disks is reduced to two subproblems of transferring $n-1$ disks, and one elementary move.

after $G_{k+1}, \ldots, G_n$ without undoing the latter. The problem of finding such an ordered set mechanically for any given problem is extremely hard but seems to be central to the mechanical discovery of heuristics (see Chapter 4).

1.3 BIBLIOGRAPHICAL AND HISTORICAL REMARKS

Some good general books on puzzles, their structure and their solution methods, are those by Gardner (1959, 1961) and by Averbach and Chein (1980). The books by Conway (1976) and Berlekamp, Conway, and Guy (1982) present formal mathematical treatments of the subject. Computer oriented discussions, including descriptions of programs which solve puzzles and games, can by found in Slagle (1971) and Banerji (1980).

The 8-Queens problem or its general n-Queen version, is typical of a class known as constraint-satisfaction or consistent-labeling problems and is often used to demonstrate ideas connected with backtracking search (Section 2.2.1). For example, see Floyd (1967), Fikes (1970), Mackworth (1977), and Gaschnig (1979a). The method of detecting dead ends such as those predicted by the condition $f' = 0$ is known as "forward checking" (Haralick and Elliot, 1980).

The 8-Puzzle is a small version of the 15-Puzzle (using a 4×4 board) first introduced in the 1870s by the American Problematist Sam Loyd (Loyd, 1959). It has served as an arena for testing heuristic methods by many AI researchers, most notably Doran and Michie (1966) and Gaschnig (1979a).

The Traveling Salesman problem (TSP) is typical of scheduling problems arising in operations research (see Wagner, 1975, and Hillier and Lieberman, 1974) and is surveyed in Bellmore and Nemhauser (1968). The use of optimistic cost estimates for finding optimal solutions is the basis of the branch-and-bound method (Lawler and Wood, 1966). The use of the optimal assignment problem to produce lower bounds for the TSP traces back to Eastman (1958), and was shown to be computable in $O(N^3)$ steps by Edmonds and Karp (1972). The use of the MST as a lower bound for the TSP is due to Held and Karp (1971). Additional bounding functions are discussed by Pohl (1973).

The Counterfeit Coin problem is typically used to demonstrate the applications of coding theory (Massey, 1976). A complete solution for the general N-coin problem has been obtained by Linial and Tarsi (1981).

The relations between the split-and-prune paradigm of operations research and the generate-and-test methods of AI are further analyzed by Ibaraki (1978a), Nau, Kumar and Kanal (1982), and Kumar and Kanal (1983). Simon (1983) compares the search paradigm, where the operations are viewed as transformations on data, with the reasoning paradigm of logic, where the elementary operations are viewed as information gathering steps (e.g., accumulating lemmas in proof procedures).

Our treatment of the state-space and problem-reduction representations follows that of Nilsson (1971). The use of AND/OR graphs for representing problems traces back to Amarel (1967) and Slagle (1963). Kowalski (1972) and vanderBrug and Minker (1975) discuss the relation between AND/OR graphs and theorem proving. Amarel (1968, 1982) gives more general expositions of the problem of finding good representations for problems.