CHAPTER

2

PROBLEMS, PROBLEM SPACES, AND SEARCH

It's not that I'm so smart, it's just that I stay with problems longer.

—Albert Einstein
(1879 –1955), German-born theoretical physicist

In the last chapter, we gave a brief description of the kinds of problems with which AI is typically concerned, as well as a couple of examples of the techniques it offers to solve those problems. To build a system to solve a particular problem, we need to do four things:

1. Define the problem precisely. This definition must include precise specifications of what the initial situation (s) will be as well as what final situations constitute acceptable solutions to the problem.
2. Analyze the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.
3. Isolate and represent the task knowledge that is necessary to solve the problem.
4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

In this chapter and the next, we discuss the first two and the last of these issues. Then, in the chapters in Part II, we focus on the issue of knowledge representation.

2.1 DEFINING THE PROBLEM AS A STATE SPACE SEARCH

Suppose we start with the problem statement "Play chess". Although there are a lot of people to whom we could say that and reasonably expect that they will do as we intended, as our request now stands it is a very incomplete statement of the problem we want solved. To build a program that could "Play chess," we would first have to specify the starting position of the chess board, the rules that define the legal moves, and the board positions that represent a win for one side or the other. In addition, we must make explicit the previously implicit goal of not only playing a legal game of chess **but** also winning the game, if possible.

For the problem "Play chess," it is fairly easy to provide a formal and complete problem description. The starting position can be described as an 8 × 8 array where each position contains a symbol standing for the appropriate piece in the official chess opening position. We can define as our goal any board position in which the opponent does not have a legal move and his or her king is under attack. The legal moves provide the way of getting from the initial state to a goal state. They can be described easily as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that

describes the change to be made to the board position to reflect the move. There are several ways in which these rules can be written. For example, we could write a rule such as that shown in Fig. 2.1.

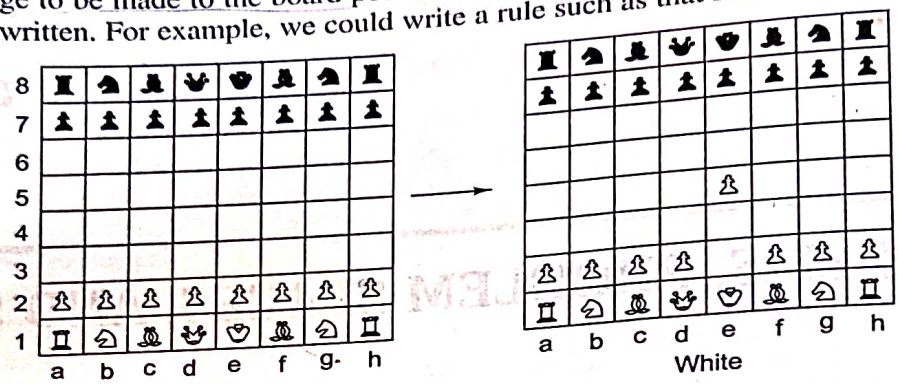

Fig. 2.1 *One Legal Chess Move*

However, if we write rules like the one above, we have to write a very large number of them since there has to be a separate rule for each of the roughly 10^{120} possible board positions. Using so many rules poses two serious practical difficulties:

- No person could ever supply a complete set of such rules. It would take too long and could certainly not be done without mistakes.
- No program could easily handle all those rules. Although a hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules poses serious difficulties.

In order to minimize such problems, we should look for a way to write the rules describing the legal moves in as general a way as possible. To do this, it is useful to introduce some convenient notation for describing patterns and substitutions. For example, the rule described in Fig. 2.1, as well as many like it, could be written as shown in Fig. 2.2.[1] In general, the more succinctly we can describe the rules we need, the less work we will have to do to provide them and the more efficient the program that uses them can be.

White pawn at
Square(file e, rank 2)
AND
Square(file e, rank 3) → move pawn from Square(file e, rank 2) to Square(file e, rank 4)
is empty
AND
Square(file e, rank 4)
is empty

Fig. 2.2 *Another Way to Describe Chess Moves*

We have just defined the problem of playing chess as a problem of moving around in a *state space*, where each state corresponds to a legal position of the board. We can then play chess by starting at an initial state, using a set of rules to move from one state to another, and attempting to end up in one of a set of final states. This state space representation seems natural for chess because the set of states, which corresponds to the set of board positions, is artificial and well-organized. This same kind of representation is also useful for naturally occurring, less well-structured problems, although it may be necessary to use more complex structures than a

[1] To be completely accurate, this rule should include a check for pinned pieces, which have been ignored here.

matrix to describe an individual state. The state space representation forms the basis of most of the AI methods we discuss here. Its structure corresponds to the structure of problem solving in two important ways:

- It allows for a formal definition of a problem as the need to convert some given situation into some desired situation using a set of permissible operations.
- It permits us to define the process of solving a particular problem as a combination of known techniques (each represented as a rule defining a single step in the space) and search, the general technique of exploring the space to try to find some path from the current state to a goal state. Search is a very important process in the solution of hard problems for which no more direct techniques are available.

In order to show the generality of the state space representation, we use it to describe a problem very different from that of chess.

> **A Water Jug Problem:** You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

The state space for this problem can be described as the set of ordered pairs of integers (x, y), such that $x = 0, 1, 2, 3,$ or 4 and $y = 0, 1, 2,$ or 3; x represents the number of gallons of water in the 4-gallon jug, and y represents the quantity of water in the 3-gallon jug. The start state is (0, 0). The goal state is $(2, n)$ for any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug).

The operators[2] to be used to solve the problem can be described as shown in Fig. 2.3. As in the chess problem, they are represented as rules whose left sides are matched against the current state and whose right sides describe the new state that results from applying the rule. Notice that in order to describe the operators completely, it was necessary to make explicit some assumptions not mentioned in the problem statement. We have assumed that we can fill a jug from the pump, that we can pour water out of a jug onto the ground, that we can pour water from one jug to another, and that there are no other measuring devices available. Additional assumptions such as these are almost always required when converting from a typical problem statement given in English to a formal representation of the problem suitable for use by a program.

To solve the water jug problem, all we need, in addition to the problem description given above, is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state. As long as it does not, the cycle continues. Clearly the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to be performed. In Chapter 3, we discuss several ways of making that selection.

For the water jug problem, as with many others, there are several sequences of operators that solve the problem. One such sequence is shown in Fig. 2.4. Often, a problem contains the explicit or implied statement that the shortest (or cheapest) such sequence be found. If present, this requirement will have a significant effect on the choice of an appropriate mechanism to guide the search for a solution. We discuss this issue in Section 2.3.4.

Several issues that often arise in converting an informal problem statement into a formal problem description are illustrated by this sample water jug problem. The first of these issues concerns the role of the conditions that occur in the left sides of the rules. All but one of the rules shown in Fig. 2.3 contain conditions that must be satisfied before the operator described by the rule can be applied. For example, the first rule says, "If the 4-gallon jug is not already full, fill it." This rule could, however, have been written as, "Fill the 4-gallon jug," since it is physically possible to fill the jug even if it is already full. It is stupid to do so since no change in the problem state results, but it is possible. By encoding in the left sides of the rules constraints that are not strictly necessary but that restrict the application of the rules to states in which the rules are most likely to lead to a solution, we can generally increase the efficiency of the problem-solving program that uses the rules.

[2] The word "operator" refers to some representation of an action. An operator usually includes information about what must be true in the world before the action can take place, and how the world is changed by the action.

1	(x, y) if $x < 4$	$\rightarrow$	$(4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	$\rightarrow$	$(x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	$\rightarrow$	$(x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	$\rightarrow$	$(x, y - d)$	Pour some water out of the 3-gallon jug
5	$(x, y\}$ if $x > 0$	$\rightarrow$	$(0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	$\rightarrow$	$(x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	$\rightarrow$	$(4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	$\rightarrow$	$(x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	$\rightarrow$	$(x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	$\rightarrow$	$(0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	$\rightarrow$	$(2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	$\rightarrow$	$(0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Fig. 2.3 *Production Rules for the Water Jug Problem*

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 11

Fig. 2.4 *One Solution to the Water Jug Problem*

The extreme of this approach is shown in the first tic-tac-toe program of Chapter 1. Each entry in the move vector corresponds to a rule that describes an operation. The left side of each rule describes a board configuration and is represented implicitly by the index position. The right side of each rule describes the operation to be performed and is represented by a nine-element vector that corresponds to the resulting board configuration. Each of these rules is maximally specific; it applies only to a single board configuration, and, as a result, no search is required when such rules are used. However, the drawback to this extreme approach is that the problem solver can take no action at all in a novel situation. In fact, essentially no problem *solving* ever really occurs. For a tic-tac-toe playing program, this is not a serious problem, since it is possible to enumerate all the situations (i.e., board configurations) that may occur. But for most problems, this is not the case. In order to solve new problems, more general rules must be available.

A second issue is exemplified by rules 3 and 4 in Fig. 2.3. Should they or should they not be included in the list of available operators? Emptying an unmeasured amount of water onto the ground is certainly allowed by the problem statement. But a superficial preliminary analysis of the problem makes it clear that doing so will never get us any closer to a solution. Again, we see the tradeoff between writing a set of rules that describe just the problem itself, as opposed to a set of rules that describe both the problem and some knowledge about its solution.

Rules 11 and 12 illustrate a third issue. To see the problem-solving knowledge that these rules represent, look at the last two steps of the solution shown in Fig. 2.4. Once the state (4, 2) is reached, it is obvious what to do next. The desired 2 gallons have been produced, but they are in the wrong jug. So the thing to do is to move them (rule 11). But before that can be done, the water that is already in the 4-gallon jug must be emptied out (rule 12). The idea behind these special-purpose rules is to capture the special-case knowledge that can be used at this stage in solving the problem. These rules do not actually add power to the system since the operations they describe are already provided by rule 9 (in the case of rule 11) and by rule 5 (in the case of rule 12). In fact, depending on the control strategy that is used for selecting rules to use during problem solving, the use of these rules may degrade performance. But the use of these rules may also improve performance if preference is given to special-case rules (as we discuss in Section 6.4.3).

We have now discussed two quite different problems, chess and the water jug problem. From these discussions, it should be clear that the first step toward the design of a program to solve a problem must be the creation of a formal and manipulable description of the problem itself. Ultimately, we would like to be able to write programs that can themselves produce such formal descriptions from informal ones. This process is called *operationalization*. It is not at all well-understood how to construct such programs, but see Section 17.3 for a description of one program that solves a piece of this problem. Until it becomes possible to automate this process, it must be done by hand, however. For simple problems, such as chess or the water jug, this is not very difficult. The problems are artificial and highly structured. For other problems, particularly naturally-occurring ones, this step is much more difficult. Consider, for example, the task of specifying precisely what it means to understand an English sentence. Although such a specification must somehow be provided before we can design a program to solve the problem, producing such a specification is itself a very hard problem. Although our ultimate goal is to be able to solve difficult, unstructured problems, such as natural language understanding, it is useful to explore simpler problems, such as the water jug problem, in order to gain insight into the details of methods that can form the basis for solutions to the harder problems.

Summarizing what we have just said, in order to provide a formal description of a problem, we must do the following:

1. Define a state space that contains all the possible configurations of the relevant objects (and perhaps some impossible ones). It is, of course, possible to define this space without explicitly enumerating all of the states it contains.

2. Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the *initial states.*
3. Specify one or more states that would be acceptable as solutions to the problem. These states are called *goal states.*
4. Specify a set of rules that describe the actions (operators) available. Doing this will require giving thought to the following issues:
   - What unstated assumptions are present in the informal problem description?
   - How general should the rules be?
   - How much of the work required to solve the problem should be precomputed and represented in the rules?

The problem can then be solved by using the rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found. Thus the process of search is fundamental to the problem-solving process. The fact that search provides the basis for the process of problem-solving does not, however, mean that other, more direct approaches cannot also be exploited. Whenever possible, they can be included as steps in the search by encoding them into the rules. For example, in the water jug problem, we use the standard arithmetic operations as single steps in the rules. We do not use search to find a number with the property that it is equal to $y - (4 - x)$. Of course, for complex problems, more sophisticated computations will be needed. Search is a general mechanism that can be used when no more direct method is known. At the same time, it provides the framework into which more direct methods for solving subparts of a problem can be embedded.

2.2 PRODUCTION SYSTEMS

Since search forms the core of many intelligent processes, it is useful to structure AI programs in a way that facilitates describing and performing the search process. Production systems provide such structures. A definition of a production system is given below. Do not be confused by other uses of the word *production*, such as to describe what is done in factories. A *production system* consists of:

- A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.[3]
- One or more knowledge/databases that contain whatever information is appropri- ate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.
- A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
- A rule applier.

So far, our definition of a production system has been very general. It encompasses a great many systems, including our descriptions of both a chess player and a water jug problem solver. It also encompasses a family of general production system interpreters, including:

- Basic production system languages, such as OPS5 [Brownston *et al.*, 1985] and ACT* [Anderson, 1983].
- More complex, often hybrid systems called *expert system shells,* which provide complete (relatively speaking) environments for the construction of knowledge- based expert systems.
- General problem-solving architectures like SOAR [Laird *et al.*, 1987], a system based on a specific set of cognitively motivated hypotheses about the nature of problem-solving.

[3] This convention for the use of left and right sides is natural for forward rules. As we will see later, many backward rule systems reverse the sides.

All of these systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved. We discuss production system issues further in Chapter 6.

We have now seen that in order to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state space (including the start and goal states) and a set of operators for moving in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modeled as a production system. In the rest of this section, we look at the problem of choosing the appropriate control structure for the production system so that the search can be as efficient as possible.

2.2.1 Control Strategies

So far, we have completely ignored the question of how to decide which rule to apply next during the process of searching for a solution to a problem. This question arises since often more than one rule (and sometimes fewer than one rule) will have its left side match the current state. Even without a great deal of thought, it is clear that how such decisions are made will have a crucial impact on how quickly, and even whether, a problem is finally solved.

- *The first requirement of a good control strategy is that it causes motion.* Consider again the water jug problem of the last section. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem. We would continue indefinitely filling the 4-gallon jug with water Control strategies that do not cause motion will never lead to a solution.
- *The second requirement of a good control strategy is that it be systematic.* Here is another simple control strategy for the water jug problem: On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But we are likely to arrive at the same state several times during the process and to use many more steps than are necessary. Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for global motion (over the course of several steps) as well as for local motion (over the course of a single step). One systematic control strategy for the water jug problem is the following. Construct a tree with the initial state as its root. Generate all the offspring of the root by applying each of the applicable rules to the initial state. Fig. 2.5 shows how the tree looks at this point. Now for each leaf node, generate all its successors by applying all the rules that are appropriate. The tree at this point is shown in Fig. 2.6.[4] Continue this process until some rule produces a goal state. This process, called *breadth-first search*, can be described precisely as follows.

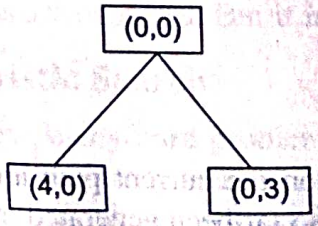

Fig. 2.5 *One Level of a Breadth-First Search Tree*

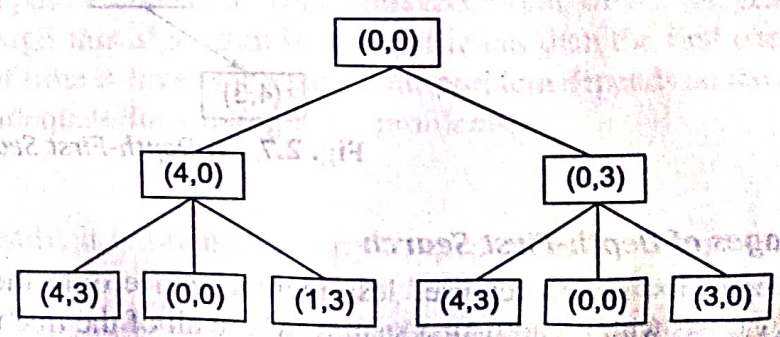

Fig. 2.6 *Two Levels of a Breadth-First Search Tree*

[4] Rule 3, 4, 11, and 12 have been ignored in constructing the search tree.

Algorithm: Breadth-First Search

1. Create a variable called *NODE-LIST* and set it to the initial state.
2. Until a goal state is found or *NODE-LIST* is empty:
   (a) Remove the first element from *NODE-LIST* and call it *E*. If *NODE-LIST* was empty, quit.
   (b) For each way that each rule can match the state described in *E* do:
      (i) Apply the rule to generate a new state,
      (ii) If the new state is a goal state, quit and return this state.
      (iii) Otherwise, add the new state to the end of *NODE-LIST.*

Other systematic control strategies are also available. For example, we could pursue a single branch of the tree until it yields a solution or until a decision to terminate the path is made. It makes sense to terminate a path if it reaches a dead-end, produces a previous state, or becomes longer than some prespecified "futility" limit. In such a case, backtracking occurs. The most recently created state from which alternative moves are available will be revisited and a new state will be created. This form of backtracking is called *chronological backtracking* because the order in which steps are undone depends only on the temporal sequence in which the steps were originally made. Specifically, the most recent step is always the first to be undone. This form of backtracking is what is usually meant by the simple term *backtracking*. But there are other ways of retracting steps of a computation. We discuss one important such way, dependency-directed backtracking, in Chapter 7. Until then, though, when we use the term backtracking, it means chronological backtracking.

The search procedure we have just described is also called *depth-first search*. The following algorithm describes this precisely.

Algorithm: Depth-First Search

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
   (a) Generate a successor, *E*, of the initial state. If there are no more successors, signal failure.
   (b) Call Depth-First Search with *E* as the initial state.
   (c) If success is returned, signal success. Otherwise continue in this loop.

Figure 2.7 shows a snapshot of a depth-first search for the water jug problem. A comparison of these two simple methods produces the following observations:

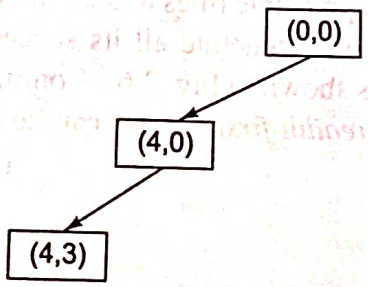

Fig. 2.7 *A Depth-First Search Tree*

Advantages of Depth-First Search

- Depth-first search requires less memory since only the nodes on the current path are stored. This contrasts with breadth-first search, where all of the tree that has so far been generated must be stored.
- By chance (or if care is taken in ordering the alternative successor states), depth-first search may find a solution without examining much of the search space at all. This contrasts with breadth-first search in which all parts of the tree must be examined to level n before any nodes on level $n + 1$ can be examined. This is particularly significant if many acceptable solutions exist. Depth-first search can stop when one of them is found.

Advantages of Breadth-First Search

- Breadth-first search will not get trapped exploring a blind alley. This contrasts with depth-first searching, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors. This is a particular problem in depth-first search if there are loops (i.e., a state has a successor that is also one of its ancestors) unless special care is expended to test for such a situation. The example in Fig. 2.7, if it continues always choosing the first (in numerical sequence) rule that applies, will have exactly this problem.
- If there is a solution, then breadth-first search is guaranteed to find it. Furthermore, if there are multiple solutions, then a minimal solution (i.e., one that requires the minimum number of steps) will be found. This is guaranteed by the fact that longer paths are never explored until all shorter ones have already been examined. This contrasts with depth-first search, which may find a long path to a solution in one part of the tree, when a shorter path exists in some other, unexplored part of the tree.

Clearly what we would like is a way to combine the advantages of both of these methods. In Section 3.3 we will talk about one way of doing this when we have some additional information. Later, in Section 12.5, we will describe an uninformed way of doing so.

For the water jug problem, most control strategies that cause motion and are systematic will lead to an answer. The problem is simple. But this is not always the case. In order to solve some problems during our lifetime, we must also demand a control structure that is efficient.

Consider the following problem.

> **The Traveling Salesman Problem:** A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

A simple, motion-causing and systematic control structure could, in principle, solve this problem. It would simply explore all possible paths in the tree and return the one with the shortest length. This approach will even work in practice for very short lists of cities. But it breaks down quickly as the number of cities grows. If there are N cities, then the number of different paths among them is $1.2...(N-1)$, or $(N-1)!$. The time to examine a single path is proportional to N. So the total time required to perform this search is proportional to $N!$. Assuming there are only 10 cities, 10! is 3,628,800, which is a very large number. The salesman could easily have 25 cities to visit. To solve this problem would take more time than he would be willing to spend. This phenomenon is called *combinatorial explosion*. To combat it, we need a new control strategy.

We can beat the simple strategy outlined above using a technique called *branch- and-bound*. Begin generating complete paths, keeping track of the shortest path found so far. Give up exploring any path as soon as its partial length becomes greater than the shortest path found so far. Using this technique, we are still guaranteed to find the shortest path. Unfortunately, although this algorithm is more efficient than the first one, it still requires exponential time. The exact amount of time it saves for a particular problem depends on the order in which the paths are explored. But it is still inadequate for solving large problems.

2.2.2 Heuristic Search

In order to solve many hard problems efficiently, it is often necessary to compromise the requirements of mobility and systematicity and to construct a control structure that is no longer guaranteed to find the best answer but that will almost always find a very good answer. Thus we introduce the idea of a heuristic.[5] A

[5] The word *heuristic* comes from the Greek word *heuriskein*, meaning "to discover," which is also the origin of *eureka*, derived from Archimedes' reputed exclamation, *heurika* (for "I have found"), uttered when he had discovered a method for determining the purity of gold.

heuristic is a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness. Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions; they are bad to the extent that they may miss points of interest to particular individuals. Some heuristics help to guide a search process without sacrificing any claims to completeness that the process might previously have had. Others (in fact, many of the best ones) may occasionally cause an excellent path to be overlooked. But, on an average, they improve the quality of the paths that are explored. Using good heuristics, we can hope to get good (though possibly nonoptimal) solutions to hard problems, such as the traveling salesman, in less than exponential time. There are some good general-purpose heuristics that are useful in a wide variety of problem domains. In addition, it is possible to construct special-purpose heuristics that exploit domain-specific knowledge to solve particular problems.

One example of a good general-purpose heuristic that is useful for a variety of combinatorial problems is the *nearest neighbor heuristic,* which works by selecting the locally superior alternative at each step. Applying it to the traveling salesman problem, we produce the following procedure:

1. Arbitrarily select a starting city.
2. To select the next city, look at all cities not yet visited, and select the one closest to the current city. Go to it next.
3. Repeat step 2 until all cities have been visited.

This procedure executes in time proportional to N^2, a significant improvement over $N!$, and it is possible to prove an upper bound on the error it incurs. For general-purpose heuristics, such as nearest neighbor, it is often possible to prove such error bounds, which provides reassurance that one is not paying too high a price in accuracy for speed.

In many AI problems, however, it is not possible to produce such reassuring bounds. This is true for two reasons:

- For real world problems, it is often hard to measure precisely the value of a particular solution. Although the length of a trip to several cities is a precise notion, the appropriateness of a particular response to such questions as "Why has inflation increased?" is much less so.
- For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is often impossible to define this knowledge in such a way that a mathematical analysis of its effect on the search process can be performed.

There are many heuristics that, although they are not as general as the nearest neighbor heuristic, are nevertheless useful in a wide variety of domains. For example, consider the task of discovering interesting ideas in some specified area. The following heuristic [Lenat, 1983b] is often useful:

If there is an interesting function of two arguments $f(x, y)$, look at what happens if the two arguments are identical.

In the domain of mathematics, this heuristic leads to the discovery of *squaring iff* is the multiplication function, and it leads to the discovery of an *identity* function if f is the function of set union. In less formal domains, this same heuristic leads to the discovery of *introspection* if f is the function contemplate or it leads to the notion of *suicide iff* is the function kill.

Without heuristics, we would become hopelessly ensnarled in a combinatorial ex-plosion. This alone might be a sufficient argument in favor of their use. But there are other arguments as well:

- Rarely do we actually need the optimum solution; a good approximation will usually serve very well. In fact, there is some evidence that people, when they solve problems, are not optimizers but rather are *satisficers* [Simon, 1981]. In other words, they seek any solution that satisfies some set of requirements, and as soon as they find one they quit. A good example of this is the search for a parking space. Most people stop as soon as they find a fairly good space, even if there might be a slightly better space up ahead.

- Although the approximations produced by heuristics may not be very good in the worst case, worst cases rarely arise in the real world. For example, although many graphs are not separable (or even nearly so) and thus cannot be considered as a set of small problems rather than one large one, a lot of graphs describing the real world are.[6]
- Trying to understand why a heuristic works, or why it doesn't work, often leads to a deeper understanding of the problem.

One of the best descriptions of the importance of heuristics in solving interesting problems is *How to Solve It* [Polya, 1957]. Although the focus of the book is the solution of mathematical problems, many of the techniques it describes are more generally applicable. For example, given a problem to solve, look for a similar problem you have solved before. Ask whether you can use either the solution of that problem or the method that was used to obtain the solution to help solve the new problem. Polya's work serves as an excellent guide for people who want to become better problem solvers. Unfortunately, it is not a panacea for AI for a couple of reasons. One is that it relies on human abilities that we must first understand well enough to build into a program. For example, many of the problems Polya discusses are geometric ones in which once an appropriate picture is drawn, the answer can be seen immediately. But to exploit such techniques in programs, we must develop a good way of representing and manipulating descriptions of those Fig.s. Another is that the rules are very general.

They have extremely underspecified left sides, so it is hard to use them to guide a search—too many of them are applicable at once. Many of the rules are really only useful for looking back and rationalizing a solution after it has been found. In essence, the problem is that Polya's rules have not been operationalized.

Nevertheless, Polya was several steps ahead of AI. A comment he made in the preface to the first printing (1944) of the book is interesting in this respect:

The following pages are written somewhat concisely, but as simply as possible, and are based on a long and serious study of methods of solution. This sort of study, called *heuristic* by some writers, is not in fashion nowadays but has a long past and, perhaps, some future.

There are two major ways in which domain-specific, heuristic knowledge can be incorporated into a rule-based search procedure:

- In the rules themselves. For example, the rules for a chess-playing system might describe not simply the set of legal moves but rather a set of "sensible" moves, as determined by the rule writer.
- As a heuristic function that evaluates individual problem states and determines how desirable they are.

A *heuristic function* is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers. Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution. Sometimes very simple heuristic functions can provide a fairly good estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed. Fig. 2.8 shows some simple heuristic functions for a few problems. Notice that sometimes a high value of the heuristic function indicates a relatively good position (as shown for chess and tic-tac- toe), while at other times a low value indicates an advantageous situation (as shown for the traveling salesman). It does not matter, in general, which way the function is stated. The program that uses the values of the function can attempt to minimize it or to maximize it as appropriate.

[6] For arguments in support of this, see Simon [1981].

Chess	the material advantage of our side over the opponent
Traveling Salesman	the sum of the distances so far
Tic-Tac-Toe	1 for each row in which we could win and in which we already have one piece plus 2 for each such row in which we have two pieces

Fig. 2.8 Some Simple Heuristic Functions

The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available. The more accurately the heuristic function estimates the true merits of each node in the search tree (or graph), the more direct the solution process. In the extreme, the heuristic function would be so good that essentially no search would be required. The system would move directly to a solution. But for many problems, the cost of computing the value of such a function would outweigh the effort saved in the search process. After all, it would be possible to compute a perfect heuristic function by doing a complete search from the node in question and determining whether it leads to a good solution. In general, there is a trade-off between the cost of evaluating a heuristic function and the savings in search time that the function provides.

In the previous section, the solutions to AI problems were described as centering on a search process. From the discussion in this section, it should be clear that it can more precisely be described as a process of heuristic search. Some heuristics will be used to define the control structure that guides the application of rules in the search process. Others, as we shall see, will be encoded in the rules themselves. In both cases, they will represent either general or specific world knowledge that makes the solution of hard problems feasible. This leads to another way that one could define artificial intelligence: the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about the problem domain.

2.3 PROBLEM CHARACTERISTICS

Heuristic search is a very general method applicable to a large class of problems. It encompasses a variety of specific techniques, each of which is particularly effective for a small class of problems. In order to choose the most appropriate method (or combination of methods) for a particular problem, it is necessary to analyze the problem along several key dimensions:

- Is the problem decomposable into a set of (nearly) independent smaller or easier subproblems?
- Can solution steps be ignored or at least undone if they prove unwise?
- Is the problem's universe predictable?
- Is a good solution to the problem obvious without comparison to all other possible solutions?
- Is the desired solution a state of the world or a path to a state?
- Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search?
- Can a computer that is simply given the problem return the solution, or will the solution of the problem require interaction between the computer and a person?

In the rest of this section, we examine each of these questions in greater detail. Notice that some of these questions involve not just the statement of the problem itself but also characteristics of the solution that is desired and the circumstances under which the solution must take place.

2.3.1 Is the Problem Decomposable?

Suppose we want to solve the problem of computing the expression

$$\int (x^2 + 3x + \sin^2 x \cdot \cos^2 x)\, dx$$

We can solve this problem by breaking it down into three smaller problems, each of which we can then solve by using a small collection of specific rules. Figure 2.9 shows the problem tree that will be generated by the process of problem decomposition as it can be exploited by a simple recursive integration program that works as follows: At each step, it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly. If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems. If it can, it creates those problems and calls itself recursively on them. Using this technique of *problem decomposition*, we can often solve very large problems easily.

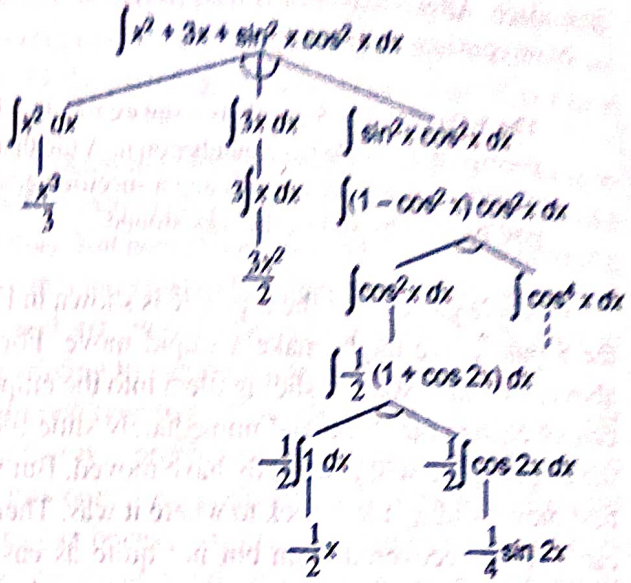

Fig. 2.9 *A Decomposable Problem*

Now consider the problem illustrated in Fig. 2.10. This problem is drawn from the domain often referred to in AI literature as the *blocks world*. Assume that the following operators are available:

1. CLEAR (*x*) [block *x* has nothing on it] → ON (*x*, Table) [pick up *x* and put it on the table]
2. CLEAR (*x*) and CLEAR (*y*) → ON (*x*, *y*) [put *x* on *y*]

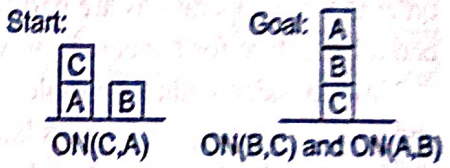

Fig. 2.10 *A Simple Blocks World Problem*

Applying the technique of problem decomposition to this simple blocks world example would lead to a solution tree such as that shown in Fig. 2.11. In the figure, goals are underlined. States that have been achieved are not underlined. The idea of this solution is to reduce the problem of getting B on C and A on B to two separate problems. The first of these new problems, getting B on C, is simple, given the start state. Simply put B on C. The second subgoal is not quite so simple. Since the only operators we have allow us to pick up single blocks at a time, we have to clear off A by removing C before we can pick up A and put it on B. This can easily be done. However, if we now try to combine the two subsolutions into one solution, we will fail. Regardless of which one we do first, we will not be able to do the second as we had planned. In this problem, the two subproblems are not independent. They interact and those interactions must be considered in order to arrive at a solution for the entire problem.

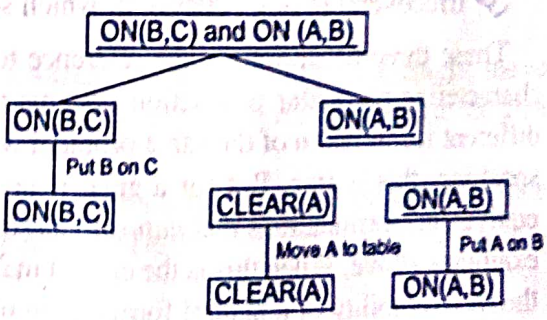

Fig. 2.11 *A Proposed Solution for the Blocks Problem*

These two examples, symbolic integration and the blocks world, illustrate the difference between decomposable and nondecomposable problems. In Chapter 3, we present a specific algorithm for problem decomposition, and in Chapter 13, we look at what happens when decomposition is impossible.

2.3.2 Can Solution Steps Be Ignored or Undone?

Suppose we are trying to prove a mathematical theorem. We proceed by first proving a lemma that we think will be useful. Eventually, we realize that the lemma is no help at all. Are we in trouble?

No. Everything we need to know to prove the theorem is still true and in memory, if it ever was. Any rules that could have been applied at the outset can still be applied. We can just proceed as we should have in the first place. All we have lost is the effort that was spent exploring the blind alley.

Now consider a different problem.

> **The 8-Puzzle:** The 8-puzzle is a square tray in which are placed, eight square tiles. The remaining ninth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

A sample game using the 8-puzzle is shown in Fig. 2.12. In attempting to solve the 8-puzzle, we might make a stupid move. For example, in the game shown above, we might start by sliding tile 5 into the empty space. Having done that, we cannot change our mind and immediately slide tile 6 into the empty space since the empty space will essentially have moved. But we can backtrack and undo the first move, sliding tile 5 back to where it was, Then we can move tile 6. Mistakes can still be recovered from but not quite as easily as in the theorem-proving problem. An additional step must be performed to undo each incorrect step, whereas no action was required to "undo" a useless lemma. In addition, the control mechanism for an 8-puzzle solver must keep track of the order in which operations are performed so that the operations can be undone one at a time if necessary. The control structure for a theorem prover does not *need* to record all that information.

Start

2	8	3
1	6	4
7		5

Goal

1	2	3
8		4
7	6	5

Fig. 2.12 *An Example of the 8-Puzzle*

Now consider again the problem of playing chess. Suppose a chess-playing program makes a stupid move and realizes it a couple of moves later. It cannot simply play as though it had never made the stupid move. Nor can it simply back up and start the game over from that point. All it can do is to try to make the best of the current situation and go on from there.

These three problems—theorem proving, the 8-puzzle, and chess—illustrate the differences between three important classes of problems:

- Ignorable (e.g., theorem proving), in which solution steps can be ignored
- Recoverable (e.g., 8-puzzle), in which solution steps can be undone
- Irrecoverable (e.g., chess), in which solution steps cannot be undone

These three definitions make reference to the steps of the solution to a problem and thus may appear to characterize particular production systems for solving a problem rather than the problem itself. Perhaps a different formulation of the same problem would lead to the problem being characterized differently. Strictly speaking, this is true. But for a great many problems, there is only one (or a small number of essentially equivalent) formulations that *naturally* describe the problem. This was true for each of the problems used as examples above. When this is the case, it makes sense to view the recoverability of a problem as equivalent to the recoverability of a natural formulation of it.

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for the problem's solution. Ignorable problems can be solved using a simple control structure that never backtracks. Such a control structure is easy to implement. Recoverable problems can be solved by a slightly more complicated control strategy that does sometimes make mistakes. Backtracking will be necessary to recover from such mistakes, so the control structure must be implemented using a push-down stack, in which decisions are recorded in case they need to be undone later. Irrecoverable problems, on the other hand, will need to be solved by a system that expends a great deal of effort making each decision since the decision must be final. Some irrecoverable problems can be solved by recoverable style methods used in a *planning* process, in which an entire sequence of steps is analyzed in advance to discover where it will lead before the first step is actually taken. We discuss next the kinds of problems in which this is possible.

2.3.3 Is the Universe Predictable?

Again suppose that we are playing with the 8-puzzle. Every time we make a move, we know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be. We can use planning to avoid having to undo actual moves, although it will still be necessary to backtrack past those moves one at a time during the planning process. Thus a control structure that allows backtracking will be necessary.

However, in games other than the 8-puzzle, this planning process may not be possible. Suppose we want to play bridge. One of the decisions we will have to make is which card to play on the first trick. What we would like to do is to plan the entire hand before making that first play. But now it is not possible to do such planning with certainty since We cannot know exactly where all the cards are or what the other players will do on their turns. The best we can do is to investigate several plans and use probabilities of the various outcomes to choose a plan that has the highest estimated probability of leading to a good score on the hand.

These two games illustrate the difference between certain-outcome (e.g., 8-puzzle) and uncertain-outcome (e.g., bridge) problems. One way of describing planning is that it is problem-solving without feedback from the environment. For solving certain-outcome problems, this open-loop approach will work fine since the result of an action can be predicted perfectly. Thus, planning can be used to generate a sequence of operators that is guaranteed to lead to a solution. For uncertain-outcome problems, however, planning can at best generate a sequence of operators that has a good probability of leading to a solution. To solve such problems, we need to allow for a process of *plan revision* to take place as the plan is carried out and the necessary feedback is provided. In addition to providing no guarantee of an actual solution, planning for uncertain-outcome problems has the drawback that it is often very expensive since the number of solution paths that need to be explored increases exponentially with the; number of points at which the outcome cannot be predicted.

The last two problem characteristics we have discussed, ignorable versus recoverable versus irrecoverable and certain-outcome versus uncertain-outcome, interact in an interesting way. As has already been mentioned, one way to solve irrecoverable problems is to plan an entire solution before embarking on an implementation of the plan. But this planning process can only be done effectively for certain-outcome problems. Thus one of the hardest types of problems to solve is the irrecoverable, uncertain-outcome. A few examples of such problems are:

- Playing bridge. But we can do fairly well since we have available accurate estimates of the probabilities of each of the possible outcomes.
- Controlling a robot arm. The outcome is uncertain for a variety of reasons. Someone might move something into the path of the arm. The gears of the arm might stick. A slight error could cause the arm to knock over a whole stack of things.
- Helping a lawyer decide how to defend his client against a murder charge. Here we probably cannot even list all the possible outcomes, much less assess their probabilities.

2.3.4 Is a Good Solution Absolute or Relative?

Consider the problem of answering questions based on a database of simple facts, such as the following:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 1991 A.D.

Suppose we ask the question "Is Marcus alive?" By representing each of these facts in a formal language, such as predicate logic, and then using formal inference methods we can fairly easily derive an answer to the question.[7] In fact, either of two reasoning paths will lead to the answer, as shown in Fig. 2.13. Since all we are interested in is the answer to the question, it does not matter which path we follow. If we do follow one path successfully to the answer, there is no reason to go back and see if some other path might also lead to a solution.

	Justification
1. Marcus was a man.	axiom 1
4. All men are mortal.	axiom 4
8. Marcus is mortal.	1, 4
3. Marcus was born in 40 A.D.	axiom 3
7. It is now 1991 A.D.	axiom 7
9. Marcus' age is 1951 years.	3, 7
6. No mortal lives longer than 150 years.	axiom 6
10. Marcus is dead.	8, 6, 9
OR	
7. It is now 1991 A.D.	axiom 7
5. All Pompeians died in 79 A.D.	axiom 5
11. All Pompeians are dead now.	7, 5
2. Marcus was a Pompeian.	axiom 2
12. Marcus is dead.	11, 2

Fig. 2.13 *Two Ways of Deciding That Marcus Is Dead*

But now consider again the traveling salesman problem. Our goal is to find the shortest route that visits each city exactly once. Suppose the cities to be visited and the distances between them are as shown in Fig. 2.14.

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

Fig. 2.14 *An Instance of the Traveling Salesman Problem*

One place the salesman could start is Boston. In that case, one path that might be followed is the one shown in Fig. 2.15, which is 8850 miles long. But is this the solution to the problem? The answer is that we cannot be sure unless we also try all other paths to make sure that none of them is shorter. In this case, as can be seen from Fig. 2.16, the first path is definitely not the solution to the salesman's problem.

These two examples illustrate the difference between any-path problems and best- path problems. Best-path problems are, in general, computationally harder than any-path problems. Any-path problems can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore. (See the discussion of best-first search in Chapter 3 for one way of doing this.) If the heuristics are not perfect, the search for a solution may not be as direct as possible, but that does not matter. For true best-path problems, however, no heuristic that could possibly miss the best solution can be used. So a much more exhaustive search will be performed.

[7] Of course, representing these statements so that a mechanical procedure could exploit them to answer the question also requires the explicit mention of other facts, such as "dead implies not alive." We do this in Chapter 5.

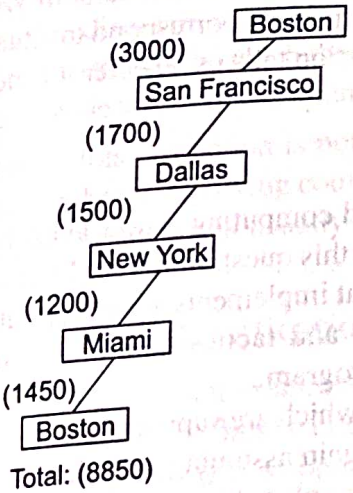

Fig. 2.15 *One Path among the Cities*

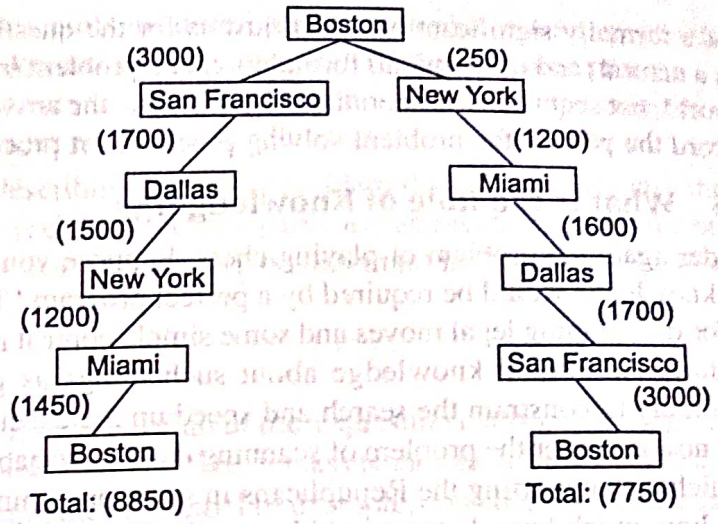

Fig. 2.16 *Two Paths Among the Cities*

2.3.5 Is the Solution a State or a Path?

Consider the problem of finding a consistent interpretation for the sentence

The bank president ate a dish of pasta salad with the fork.

There are several components of this sentence, each of which, in isolation, may have more than one interpretation. But the components must form a coherent whole, and so they constrain each other's interpretations. Some of the sources of ambiguity in this sentence are the following:

The word "bank" may refer either to a financial institution or to a side of a river. But only one of these may have a president.

- The word "dish" is the object of the verb "eat." It is possible that a dish was eaten. But it is more likely that the pasta salad in the dish was eaten.
- Pasta salad is a salad containing pasta. But there are other ways meanings can be formed from pairs of nouns. For example, dog food does not normally contain dogs.
- The phrase "with the fork" could modify several parts of the sentence. In this case, it modifies the verb "eat." But, if the phrase had been "with vegetables," then the modification structure would be different. And if the phrase had been "with her friends," the structure would be different still.

Because of the interaction among the interpretations of the constituents of this sentence, some search may be required to find a complete interpretation for the sentence. But to solve the problem of finding the interpretation we need to produce only the interpretation itself. No record of the processing by which the interpretation was found is necessary.

Contrast this with the water jug problem. Here it is not sufficient to report that we have solved the problem and that the final state is (2, 0). For this kind of problem, what we really must report is not the final state but the path that we found to that state. Thus a statement of a solution to this problem must be a sequence of operations (sometimes called *apian)* that produces the final state.

These two examples, natural language understanding and the water jug problem, illustrate the difference between problems whose solution is a state of the world and problems whose solution is a path to a state. At one level, this difference can be ignored and all problems can be formulated as ones in which only a state is required to be reported. If we do this for problems such as the water jug, then we must redescribe our states so that each state represents a partial path to a solution rather than just a single state of the world. So this question

is not a formally significant one. But, just as for the question of ignorability versus recoverability, there is often a natural (and economical) formulation of a problem in which problem states correspond to situations in the world, not sequences of operations. In this case, the answer to this question tells us whether it is necessary to record the path of the problem-solving process as it proceeds.

2.3.6 What is the Role of Knowledge?

Consider again the problem of playing chess. Suppose you had unlimited computing power available. How much knowledge would be required by a perfect program? The answer to this question is very little—just the rules for determining legal moves and some simple control mechanism that implements an appropriate search procedure. Additional knowledge about such things as good strategy and tactics could of course help considerably to constrain the search and speed up the execution of the program.

But now consider the problem of scanning daily newspapers to decide which are supporting the Democrats and which are supporting the Republicans in some upcoming election. Again assuming unlimited computing power, how much knowledge would be required by a computer trying to solve this problem? This time the answer is a great deal. It would have to know such things as:

- The names of the candidates in each party.
- The fact that if the major thing you want to see done is have taxes lowered, you are probably supporting the Republicans.
- The fact that if the major thing you want to see done is improved education for minority students, you are probably supporting the Democrats.
- The fact that if you are opposed to big government, you are probably supporting the Republicans.
- And so on ...

These two problems, chess and newspaper story understanding, illustrate the difference between problems for which a lot of knowledge is important only to constrain the search for a solution and those for which a lot of knowledge is required even to be able to recognize a solution.

2.3.7 Does the Task Require Interaction with a Person?

Sometimes it is useful to program computers to solve problems in ways that the majority of people would not be able to understand. This is fine if the level of the interaction between the computer and its human users is problem-in solution-out. But increasingly we are building programs that require intermediate interaction with people, both to provide additional input to the program and to provide additional reassurance to the user.

Consider, for example, the problem of proving mathematical theorems. If

1. All we want is to know that there is a proof
2. The program is capable of finding a proof by itself

then it does not matter what strategy the program takes to find the proof. It can use, for example, the *resolution* procedure (see Chapter 5), which can be very efficient but which does not appear natural to people. But if either of those conditions is violated, it may matter very much how a proof is found. Suppose that we are trying to prove some new, very difficult theorem. We might demand a proof that follows traditional patterns so that a mathematician can read the proof and check to make sure it is correct. Alternatively, finding a proof of the theorem might be sufficiently difficult that the program does not know where to start. At the moment, people are still better at doing the high-level strategy required for a proof. So the computer might like to be able to ask for advice. For example, it is often much easier to do a proof in geometry if someone suggests the right line to draw into the Fig.. To exploit such advice, the computer's reasoning must be analogous to that of its human advisor, at least on a few levels. As computers move into areas of great significance to human lives, such as medical diagnosis, people will be very unwilling to accept the verdict of a program whose reasoning they cannot follow. Thus we must distinguish between two types of problems:

- Solitary, in which the computer is given a problem description and produces an answer with no intermediate communication and with no demand for an explanation of the reasoning process
- Conversational, in which there is intermediate communication between a person and the computer, either to provide additional assistance to the computer or to provide additional information to the user, or both

Of course, this distinction is not a strict one describing particular problem domains. As we just showed, mathematical theorem proving could be regarded as either. But for a particular application, one or the other of these types of systems will usually be desired and that decision will be important in the choice of a problem-solving method.

2.3.8 Problem Classification

When actual problems are examined from the point of view of all of these questions, it becomes apparent that there are several broad classes into which the problems fall. These classes can each be associated with a generic control strategy that is appropriate for solving the problem. For example, consider the generic problem of *classification*. The task here is to examine an input and then decide which of a set of known classes the input is an instance of. Most diagnostic tasks, including medical diagnosis as well as diagnosis of faults in mechanical devices, are examples of classification. Another example of a generic strategy is *propose and refine*. Many design and planning problems can be attacked with this strategy.

Depending on the granularity at which we attempt to classify problems and control strategies, we may come up with different lists of generic tasks and procedures. See Chandrasekaran [1986] and McDermott [1988] for two approaches to constructing such lists. The important thing to remember here, though, since we are about to embark on a discussion of a variety of problem-solving methods, is that there is no one single way of solving all problems. But neither must each new problem be considered totally *ab initio*. Instead, if we analyze our problems carefully and sort our problem-solving methods by the kinds of problems for which they are suitable, we will be able to bring to each new problem much of what we have learned from solving other, similar problems.

2.4 PRODUCTION SYSTEM CHARACTERISTICS

We have just examined a set of characteristics that distinguish various classes of problems. We have also argued that production systems are a good way to describe the operations that can be performed in a search for a solution to a problem. Two questions we might reasonably ask at this point are:

1. Can production systems, like problems, be described by a set of characteristics that shed some light on how they can easily be implemented?
2. If so, what relationships are there between problem types and the types of pro- duction systems best suited to solving the problems?

The answer to the first question is yes. Consider the following definitions of classes of production systems. A *monotonic production system* is a production system in which the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected. A *nonmonotonic production system* is one in which this is not true. A *partially commutative production system* is a production system with the property that if the application of a particular sequence of rules transforms state x into state y, then any permutation of those rules that is allowable (i.e., each rule's preconditions are satisfied when it is applied) also transforms state x into state y. A *commutative production system* is a production system that is both monotonic and partially commutative.[8]

[8] This corresponds to the definition of a commutative production system given in Nilsson [1980].

The significance of these categories of production systems lies in the relationship between the categories and appropriate implementation strategies. But before discussing that relationship, it may be helpful to make the meanings of the definitions clearer by showing how they relate to specific problems.

Thus we arrive at the second question above, which asked whether there is an interesting relationship between classes of production systems and classes of problems. For any solvable problem, there exist an infinite number of production systems that describe ways to find solutions. Some will be more natural or efficient than others. Any problem that can be solved by any production system can be solved by a commutative one (our most restricted class), but the commutative one may be so unwieldy as to be practically useless. It may use individual states to represent entire sequences of applications of rules of a simpler, noncommutative system. So in a formal sense, there is no relationship between kinds of problems and kinds of production systems since all problems can be solved by all kinds of systems. But in a practical sense, there definitely is such a relationship between kinds of problems and the kinds of systems that lend themselves naturally to describing those problems. To see this, let us look at a few examples. Fig. 2.17 shows the four categories of production systems produced by the two dichotomies, monotonic versus nonmonotonic and partially commutative versus

	Monotonic	Nonmonotonic
Partially commutative	Theorem proving	Robot navigation
Not partially commutative	Chemical synthesis	Bridge

Fig. 2.17 *The Four Categories of Production Systems*

nonpartially commutative, along with some problems that can naturally be solved by each type of system. The upper left corner represents commutative systems.

Partially commutative, monotonic production systems are useful for solving ignorable problems. This is not surprising since the definitions of the two are essentially the same. But recall that ignorable problems are those for which a *natural* formulation leads to solution steps that can be ignored. Such a natural formulation will then be a partially commutative, monotonic system. Problems that involve creating new things rather than changing old ones are generally ignorable. Theorem proving, as we have described it, is one example of such a creative process. Making deductions from some known facts is a similar creative process. Both of those processes can easily be implemented with a partially commutative, monotonic system.

Partially commutative, monotonic production systems are important from an implementation standpoint because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed. Although it is often useful to implement such systems with backtracking in order to guarantee a systematic search, the actual database representing the problem state need not be restored. This often results in a considerable increase in efficiency, particularly because, since the database will never have to be restored, it is not necessary to keep track of where in the search process every change was made.

We have now discussed partially commutative production systems that are also monotonic. They are good for problems where things do not change; new things get created. Nonmonotonic, partially commutative systems, on the other hand, are useful for problems in which changes occur but can be reversed and in which order of operations is not critical. This is usually the case in physical manipulation problems, such as robot navigation on a flat plane. Suppose that a robot has the following operators: go north (N), go east (E), go south (S), and go west (W). To reach its goal, it does not matter whether the robot executes N-N-E or N-E-N.

Depending on how the operators are chosen, the 8-Puzzle and the blocks world problem can also be considered partially commutative.

Both types of partially commutative production systems are significant from an implementation point of view because they tend to lead to many duplications of individual states during the search process. This is discussed further in Section 2.5.

Production systems that are not partially commutative are useful for many problems in which irreversible changes occur. For example, consider the problem of determining a process to produce a desired chemical compound. The operators available include such things as "Add chemical x to the pot" or "Change the temperature to t degrees." These operators may cause irreversible changes to the potion being brewed. The order in which they are performed can be very important in determining the final output. It is possible that if x is added to y, a stable compound will be formed, so later addition of z will have no effect; if z is added to y, however, a different stable compound may be formed, so later addition of x will have no effect. Nonpartially commutative production systems are less likely to produce the same node many times in the search process. When dealing with ones that describe irreversible processes, it is particularly important to make correct decisions the first time, although if the universe is predictable, planning can be used to make that less important.

2.5 ISSUES IN THE DESIGN OF SEARCH PROGRAMS

Every search process can be viewed as a traversal of a tree structure in which each node represents a problem state and each arc represents a relationship between the states represented by the nodes it connects. For example, Fig. 2.18 shows part of a search tree for a water jug problem. The arcs have not been labeled in the Fig., but they correspond to particular water-pouring operations. The search process must find a path or paths through the tree that connect an initial state with one or more final states. The tree that must be searched could, in principle, be constructed in its entirety from the rules that define allowable moves in the problem space. But, in practice, most of it never is. It is too large and most of it need never be explored. Instead of first building the tree *explicitly* and then searching it, most search programs represent the tree *implicitly* in the rules and generate explicitly only those parts that they decide to explore. Throughout our discussion of search methods, it is important to keep in mind this distinction between implicit search trees and the explicit partial search trees that are actually constructed by the search program.

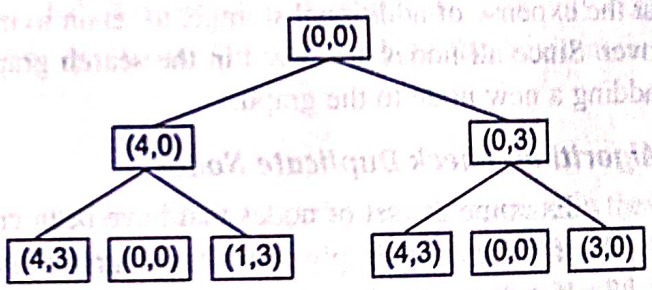

Fig. 2.18 *A Search Tree for the Water Jug Problem*

In the next chapter, we present a family of general-purpose search techniques. But before doing so we need to mention some important issues that arise in all of them:

- The direction in which to conduct the search (*forward* versus *backward* reasoning). We can search forward through the state space from the start state to a goal state, or we can search backward from the goal.
- How to select applicable rules *(matching)*. Production systems typically spend most of their time looking for rules to apply, so it is critical to have efficient procedures for matching rules against states.
- How to represent each node of the search process (the *knowledge representation problem* and the *frame problem)*. For problems like chess, a node can be fully represented by a simple array. In more complex problem solving, however, it is inefficient and/or impossible to represent all of the facts in the world and to determine all of the side effects an action may have.

We discuss the knowledge representation and frame problems further in Chapter 4. We investigate matching and forward versus backward reasoning when we return to production systems in Chapter 6.

One other issue we should consider at this point is that of search trees versus search graphs. As mentioned above, we can think of production rules as generating nodes in a search tree. Each node can be expanded in turn, generating a set of successors. This process continues until a node representing a solution is found. Implementing such a procedure requires little bookkeeping. However, this process often results in the same node being generated as part of several paths and so being processed more than once. This happens because the search space may really be an arbitrary directed graph rather than a tree.

For example, in the tree shown in Fig. 2.18, the node (4,3), representing 4-gallons of water in one jug and 3 gallons in the other, can be generated either by first filling the 4-gallon jug and then the 3-gallon one or by filling them in the opposite order. Since the order does not matter, continuing to process both these nodes would be redundant. This example also illustrates another problem that often arises when the search process operates as a tree walk. On the third level, the node (0, 0) appears. (In fact, it appears twice.) But this is the same as the top node of the tree, which has already been expanded. Those two paths have not gotten Us anywhere. So we would like to eliminate them and continue only along the other branches.

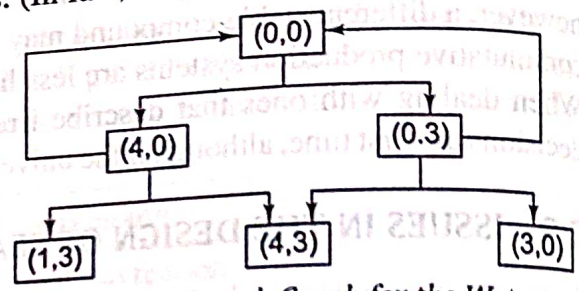

Fig. 2.19 *A Search Graph for the Water Jug Problem*

The waste of effort that arises when the same node is generated more than once can be avoided at the price of additional bookkeeping. Instead of traversing a search tree, we traverse a directed graph. This graph differs from a tree in that several paths may come together at a node. The graph corresponding to the tree of Fig. 2.18 is shown in Fig. 2.19.

Any tree search procedure that keeps track of all the nodes that have been generated so far can be converted to a graph search procedure by modifying the action performed each time a node is generated. Notice that of the two systematic search procedures we have discussed so far, this requirement that nodes be kept track of is met by breadth-first search but not by depth-first search. But, of course, depth-first search could be modified, at the expense of additional storage, to retain in memory nodes that have been expanded and then backed-up over. Since all nodes are saved in the search graph, we must use the following algorithm instead of simply adding a new node to the graph.

Algorithm: Check Duplicate Nodes

1. Examine the set of nodes that have been created so far to see if the new node already exists.
2. If it does not-simply add it to the graph just as for a tree.
3. If it does already exist, then do the following:
   (a) Set the node that is being expanded to point to the already existing-node corresponding to its successor rather than to the new one. The new one can simply be thrown away.
   (b) If you are keeping track of the best (shortest or otherwise least-cost) path to each node, then check to see if the new path is better or worse than the old one. If worse, do nothing. If better, record the new path as the correct path to use to get to the node and propagate the corresponding change in cost down through successor nodes as necessary.

One problem that may arise here is that cycles may be introduced into the search graph. A *cycle* is a path through the graph in which a given node appears more than once. For example, the graph of Fig. 2.19 contains two cycles of length two. One includes the nodes (0, 0) and (4, 0); the other includes the nodes (0, 0) and (0, 3). Whenever there is a cycle, there can be paths of arbitrary length. Thus it may become more difficult to show that a graph traversal algorithm is guaranteed to terminate.

Treating the search process as a graph search rather than as a tree search reduces the amount of effort that is spent exploring essentially the same path several times. But it requires additional effort each time a node is

generated to see if it has been generated before. Whether this effort is justified depends on the particular problem. If it is very likely that the same node will be generated in several different ways, then it is more worthwhile to use a graph procedure than if such duplication will happen only rarely.

Graph search procedures are especially useful for dealing with partially commutative production systems in which a given set of operations will produce the same result regardless of the order in which the operations are applied. A systematic search procedure will try many of the permutations of these operators and so will generate the same node many times. This is exactly what happened in the water jug example shown above.

2.6 ADDITIONAL PROBLEMS

Several specific problems have been discussed throughout this chapter. Other problems have not yet been mentioned, but are common throughout the AI literature. Some have become such classics that no AI book could be complete without them, so we present them in this section. A useful exercise, at this point, would be to evaluate each of them in light of the seven problem characteristics we have just discussed.

A brief justification is perhaps required before this parade of toy problems is presented. Artificial intelligence is not merely a science of toy problems and microworlds (such as the blocks world). Many of the techniques that have been developed for these problems have become the core of systems that solve very nontoy problems. So think about these problems not as defining the scope of AI but rather as providing a core from which much more has developed.

The Missionaries and Cannibals Problem

Three missionaries and three cannibals find themselves on one side of a river. They have agreed that they would all like to get to the other side. But the missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two people at a time. How can everyone get across the river without the missionaries risking being eaten?

The Tower of Hanoi

Somewhere near Hanoi there is a monastery whose monks devote their lives to a very important task. In their courtyard are three tall posts. On these posts is a set of sixty-four disks, each with a hole in the center and each of a different radius. When the monastery was established, all of the disks were on one of the posts, each disk resting on the one just larger than it. The monks' task is to move all of the disks to one of the other pegs. Only one disk may be moved at a time, and all the other disks must be on one of the pegs. In addition, at no time during the process may a disk be placed on top of a smaller disk. The third peg can, of course, be used as a temporary resting place for the disks. What is the quickest way for the monks to accomplish their mission?

Even the best solution to this problem will take the monks a very long time. This is fortunate, since legend has it that the world will end when they have finished.

The Monkey and Bananas Problem

A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey, unfortunately, cannot reach the bananas. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey to take to acquire lunch?

SEND	DONALD	CROSS
+MORE	+GERALD	+ROADS
.....	.....	.....
MONEY	ROBERT	DANGER

Fig. 2.20 *Some Cryptarithmetic Problems*

Cryptarithmetic

Consider an arithmetic problem represented in letters, as shown iff the examples in Fig. 2.20. Assign a decimal digit to each of the letters in such a way that the answer to the problem is correct. If the same letter occurs more than once, it must be assigned the same digit each time. No two different letters may be assigned the same digit.

People's strategies for solving cryptarithmetic problems have been, studied intensively by Newell and Simon [1972].

SUMMARY

In this chapter, we have discussed the first two steps that must be taken toward the design of a program to solve a particular problem:

1. Define the problem precisely. Specify the problem space, the operators for moving within the space, and the starting and goal state(s).
2. Analyze the problem to determine where it falls with respect to seven important issues.

The last two steps for developing a program to solve that problem are, of course:

3. Identify and represent the knowledge required by the task.
4. Choose one or more techniques for problem solving, and apply those techniques to the problem.

Several general-purpose, problem-solving techniques are presented in the next chapter, and several of them have already been alluded to in the discussion of the problem characteristics in this chapter. The relationships between problem characteristics and specific techniques should become even clearer as we go on. Then, in Part II, we discuss the issue of how domain knowledge is to be represented.

EXERCISES

1. In this chapter, the following problems were mentioned:
   - Chess
   - 8-puzzle
   - Missionaries and cannibals
   - Monkey and bananas
   - Bridge
   - Water jug
   - Traveling salesman
   - Tower of Hanoi
   - Cryptarithmetic

   Analyze each of them with respect to the seven problem characteristics discussed in Section 2.3.
2. Before we can solve a problem using state space search, we must define an appropriate state space. For each of the problems mentioned above for which it was not done in the text, find a good state space representation.
3. Describe how the branch-and-bound technique could be used to find the shortest solution to a water jug problem.

4. For each of the following types of problems, try to describe a good heuristic function:
   (a) Blocks world
   (b) Theorem proving
   (c) Missionaries and cannibals
5. Give an example of a problem for which breadth-first search would work better than depth-first search. Give an example of a problem for which depth-first search would work better than breadth-first search.
6. Write an algorithm to perform breadth-first search of a problem *graph.* Make sure your algorithm works properly when a single node is generated at more than one level in the graph.
7. Try to construct an algorithm for solving blocks world problems, such as the one in Fig. 2.10. Do not cheat by looking ahead to Chapter 13.