CHAPTER2: PROBLEMS AND PROBLEM SPACES

Three major steps are required to build a system to solve a particular problem.

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

2.1 Defining the problem as a state space search

The state space representation forms the basis of many AI methods. 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 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.

Example 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 a 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).

1. $(x, y \mid x < 4) \rightarrow (4, y)$ Fill the 4-gallon jug

2. $(x, y \mid y < 3) \rightarrow (x, 3)$ Fill the 3-gallon jug

3. $(x, y \mid x > 0) \rightarrow (x - d, y)$ Pour some water out of the 4-gallon jug

4. $(x, y \mid y > 0) \rightarrow (x, y - d)$ Pour some water out of the 3-gallon jug

5. $(x, y \mid x > 0) \rightarrow (0, y)$ Empty the 4-gallon jug on the ground

6. $(x, y \mid y > 0) \rightarrow (x, 0)$ Empty the 3-gallon jug on the ground

7. $(x, y \mid x + y >= 4 \wedge y > 0) \rightarrow (4, y - (4 - x))$ Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full

8. $(x, y \mid x + y >= 3 \wedge x > 0) \rightarrow (x - (3 - y), 3)$ Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full

9. $(x, y \mid x + y <= 4 \wedge y > 0) \rightarrow (x + y, 0)$ Pour all the water from the 3-gallon jug into the 4-gallon jug

10. $(x, y \mid x + y <= 3 \wedge x > 0) \rightarrow (0, x + y)$ Pour all the water from the 4-gallon jug into the 3-gallon jug

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.

Clearly, the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to perform.

Gallons in 4-gallon jug	Gallons in 3-gallon jug	Rule applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5
0	2	9
2	0	

For the water jug problem, as with many others, there are several sequences of operations that will solve the problem.

In order to provide a formal description of a problem it is necessary to do the following things:

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 instated assumptions are present in the informal problem description?
   - How general should the rules be made?
   - How much of the work required to solve the problem should be precomputed and represented as 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 need to be performed. 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 that are appropriate for the solution of subparts of a problem can be embedded.

2.1.1 Production Systems

Since search forms the core of many intelligent processes, it is useful to structure A.I. programs in a way that facilitates describing the search process. Production systems provide such structures.

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 action to be performed if the rule is applied.

- One or more databases that contain whatever information is appropriate 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.

In addition to its usefulness as a way to describe search, the production system model has other advantages as a formalism in A.I.:

- It is a good way to model the strong data-driven nature of intelligent action. As new inputs enter the database, the behavior of the system changes.

- New rules can easily be added to account for new situations without disturbing the rest of the system. This is important since no A.I. program is ever completed. Although sometimes confusion arises from interaction among rules, it is often less severe than the corresponding complications of modifying straight-line code.

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 and a set of operators for moving in the 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 will look at the problem of choosing the appropriate control structure for the production system so that the search can be as efficient as possible.

2.1.2 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 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 cause motion.

Consider again the water jug problem.

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.

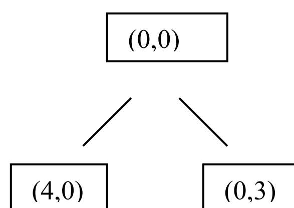
Here is another simple control strategy for the water jug problem: On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But we are likely to arrive at the same state several times during the process and to use many more steps than are necessary.

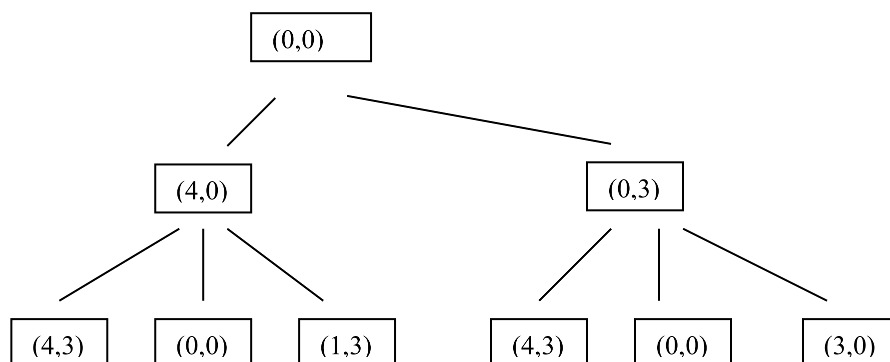
Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution.

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.

Continue this process until some rule produces a goal state.

This process is called *breadth-first search.*





Two Levels of a Breadth-First Search Tree

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 some prespecified depth has been reached and only then go back and explore other branches.

This is called *depth-first search.*

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).

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.

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 so that he travels the shortest possible distance on a round trip, starting at any one of the cities and then returning there.

A simple, motion-causing, and systematic control structure could, in principle, solve this problem. Simply explore the tree of all possible paths 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 (N — 1)!. It takes time proportional to N to examine a single path. So the total time required to perform this search is O(N!). 10! is 3,628,800, already a very large number. The salesman could easily have twenty-five cities to visit. To solve his problem would take more time than he would be willing to wait. This phenomenon is called *combinatorial explosion.* To combat it, a new control strategy is needed.

We can beat the 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.

2.1.3 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.

A *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 interesting directions; they are bad to the extent that they lead into dead ends.

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 the average, they improve the quality of the paths that are explored. Using good heuristics, we can hope to get good (even if 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 algorithm,* which works by selecting the locally superior alternative at each step. Applying it to the traveling salesman problem produces the following procedure:

1. Arbitrarily select a starting city.
2. To select the next city, look at all cities not yet visited. 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 squared, a significant improvement over N!

For general-purpose heuristics, such as the nearest neighbor algorithm, it is often possible to prove such error bounds, which provide reassurance that one is not paying too high a price in accuracy for speed.

In many A.I. 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 goodness 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.

Even in such unstructured situations, it may be possible to say something about the efficiency of the search process.

There are many heuristics, that, although they are not as general as the nearest neighbor algorithm, 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 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.

Without heuristics, we would become hopelessly ensnarled in a combinatorial explosion. 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*. 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 will 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.

Another way that one could define artificial intelligence. *Artificial Intelligence* is the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about the problem domain.

2.2 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:

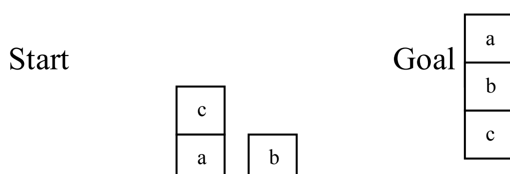
- Is the problem decomposable into a set of independent smaller or easier subproblems?
- Can solution steps be ignored or at least undone if they prove unwise?
- Is the problem's universe predictable?
- Is a good solution to the problem obvious without comparison to all other possible solutions?
- Is the knowledge base to be used for solving the problem internally consistent?
- Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain search?
- Can a computer simply be given the problem and then return with the solution, or will the solution of the problem require interaction between the computer and a person?

2.2.1 Is the Problem Decomposable?

Suppose we want to solve the problem of computing the expression: $x^3 + x^2 + 2x$.
We can solve this problem by breaking it down into three smaller problems.

Using the technique of *problem decomposition,* very large problems can often be solved very easily.

Example (non-decomposable problem) A simple block world problem

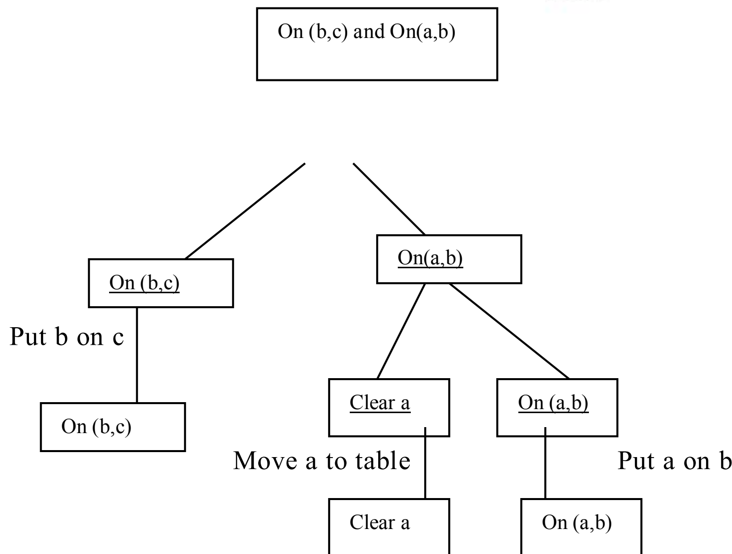


Assume that the following operators are available:

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]

Applying the technique of problem decomposition to this simple blocks world example would lead to a solution tree



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.

Decomposable problems can be solved by the *divide-and-conquer* technique of problem decomposition.

Nondecomposable ones generally cannot, although it is sometimes possible to use such techniques to generate an approximate solution and then patch it to fix bugs caused by interactions.

2.2.2 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. Do we have a problem?

No. Everything we need to know to prove the theorem is still true and in the knowledge base, 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 8 square tiles. The remaining ninth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

can be slid into that space. A game consists of a starting position and a specified goal position.

Start Goal

The goal is to transform the starting position into the goal position by sliding the tiles around.

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.

Now consider 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

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for its 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 pushdown stack, in which decisions are recorded in case they need later to be undone. 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 will discuss below the kinds of problems in which this is possible.

2.2.3 Is the Universe Predictable?

Again suppose that we are playing with the 8-puzzle. Everytime 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.

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, it is also necessary 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.

One of the hardest types of problems to solve is the irrecoverable, uncertain-outcome.

Examples:

- 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.

2.2.4 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 1983 A.D.

Suppose we ask the question "Is Marcus alive?" By representing each of the 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.

		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	axiom 3
7	It is now 1983	axiom 7
9	Marcus' age is 1943 years	3,7
6	No mortal lives longer than 150 years	axiom 6
10	Marcus is dead	8, 6, 9

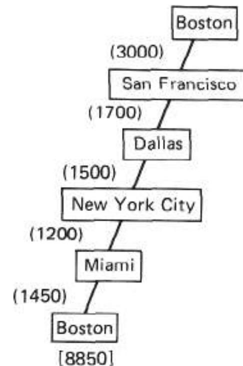
OR

7	It is now 1983	axiom 7
5	All Pompeians died in 79	axiom 5
11	All Pompeians are dead now	7, 5
2	Marcus was a Pompeian	axiom 2
12	Marcus is dead	11, 2

In fact, either of two reasoning paths will lead to the answer. 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.

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 we need to visit and the distances between them are as shown in the following table.

	Boston	New York City	Miami	Dallas	San Francisco
Boston		250	1450	1700	3000
New York City	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
San Francisco	3000	2900	3300	1700	



One place the salesman could start is Boston. If he does that, then one path he might follow is that shown above.

He would then travel a total of 8850 miles. 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.

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.

2.2.5 Is the Knowledge Base Consistent?

Suppose we are given the following set of axioms for a multiplicative group:

1. XY is defined for all XY
2. $X = Y \wedge Y = Z \rightarrow X = Z$
3. $X = X$
4. $(X\ Y)\ Z = X\ (Y\ Z)$
5. $1\ X = X$
6. $X^{-1}\ X = 1$

7. $X = Y \rightarrow ZX = ZY$
8. $X = Y \rightarrow XZ = YZ$

We want to be able to solve problems such as

Prove that $X\ 1 = X$

We can use any of the standard proof procedures of mathematics since the set of axioms is consistent. But now suppose that we want to be able to solve the following problem:

The Target Problem

A man is standing 150 ft from a target. He plans to try to hit the target by shooting a gun that fires bullets with a velocity of 1500 ft/sec. How high above the target should he aim?

One way to solve this problem is to reason as follows. It takes the bullet .1 sec to reach the target, assuming it travels the straight line of length 150 ft from the man to the target. During 0. 1 sec, the bullet falls a distance equal to

$$1/2\ g\ t^2\ \text{ft} = 1/2\ (32)\ (.1)^2\ \text{ft} = .16\ \text{ft}$$

So if the man aims up .16 feet, then in the time it takes the bullet to reach the target it will fall .16 ft and will exactly hit the target. But notice that in solving this problem, we made the assumption that the bullet would travel in a straight line. That assumption clearly conflicts with the conclusion that the bullet will travel in an arc.

These two problems illustrate the difference between problems in a totally consistent world and problems in a world in which inconsistencies may exist in the database. Many reasoning schemes that work well in consistent domains are not appropriate in inconsistent ones. For example, in standard logic, if the database contains A and not A, then it is possible to prove anything. Such a system would be of little help in answering questions with respect to an inconsistent database.

2.2.6 What Is the Role of Knowledge?

Consider 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. In fact, without such knowledge the chess problem is not realistically solvable.

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 to have taxes lowered, you are probably supporting the Republicans.
- The fact that, if the major thing you want to see done is to improve educa tion for minority students, you are probably supporting the Democrats.
- The fact that, if you are opposed to big government, you are probably sup porting 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.

2.2.7 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 all we want is to know that there is a proof, and if the program is capable of finding a proof by itself if one exists, then it does not matter what strategy the program takes to find the proof. It can use, for example, the resolution procedure, 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 it 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 figure. 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 will be given a problem description and will produce an answer, with no intermediate communication and with no demand for an explanation of the reasoning process
- Conversational, in which there will be 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 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 production 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 *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.

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 it clearer what the definitions mean by showing how they relate to specific problems.

	Monotonic	Nonmonotonic
Partially commutative	Theorem proving	Blocks world 8-puzzle
Not partially commutative	Chemical synthesis	Bridge

Thus we come to the second question above, which asked whether there is an interesting relationship between classes of production systems and classes of problems. For any problem, there exists an infinite number of production systems that describe ways to find solutions. Some will be more natural or efficient than others. It turns out that any problem that can be solved by any production system can be solved by a commutative one (our most restricted class). But that 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. shows the four categories of production systems produced by the two dichotomies, monotonic versus nonmonotonic and partially commutative versus 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 was 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 the 8-puzzle. It does not matter

in what order a set of simple motion operators is carried out. The final state will differ from the start state by the sum of the individual movements.

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.

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 Y 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 A is added to B, a stable compound will be formed, so that later addition of C will have no effect; if C is added to B, however, a different stable compound may be formed, so that later addition of A 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.