Chapter 1

What Is Artificial Intelligence?

What exactly is artificial intelligence? Although most attempts to define complex and widely used terms precisely are exercises in futility, it is useful to draw at least an approximate boundary around the concept to provide a perspective on the discussion that follows. To do this, we propose the following by no means universally accepted definition. *Artificial intelligence* (AI) is the study of how to make computers do things which, at the moment, people do better. This definition is, of course, somewhat ephemeral because of its reference to the current state of computer science. And it fails to include some areas of potentially very large impact, namely problems that cannot now be solved well by either computers or people. But it provides a good outline of what constitutes artificial intelligence, and it avoids the philosophical issues that dominate attempts to define the meaning of either *artificial* or *intelligence*. Interestingly, though, it suggests a similarity with philosophy at the same time it is avoiding it. Philosophy has always been the study of those branches of knowledge that were so poorly understood that they had not yet become separate disciplines in their own right. As fields such as mathematics or physics became more advanced, they broke off from philosophy. Perhaps if AI succeeds it can reduce itself to the empty set.

1.1 The AI Problems

What then are some of the problems contained within AI? Much of the early work in the field focused on formal tasks, such as game playing and theorem proving. Samuel wrote a checkers-playing program that not only played games with opponents but also used its experience at those games to improve its later performance. Chess also received a good deal of attention. The Logic Theorist was an early attempt to prove mathematical theorems. It was able to prove several theorems from the first chapter of Whitehead and Russell's *Principia Mathematica*. Gelernter's theorem prover explored another area of mathematics: geometry. Game playing and theorem proving share the property that people who do them well are considered to be displaying intelligence. Despite this, it appeared initially that computers could perform well at those tasks simply by being fast at exploring a large number of solution paths and then selecting the best one. It was thought that this process required very little knowledge and could therefore be programmed easily. As we will see later, this assumption turned out to be false since no computer is fast enough to overcome the combinatorial explosion generated by most problems.

Another early foray into AI focused on the sort of problem solving that we do every day when we decide how to get to work in the morning, often called *commonsense reasoning*. It includes reasoning about physical objects and their relationships to each other (e.g., an object can be in only one place at a time), as well as reasoning about actions and their consequences (e.g., if you let go of something, it will fall to the floor and maybe break). To investigate this sort of reasoning, Newell, Shaw, and Simon built the General Problem Solver (GPS), which they applied to several commonsense tasks as well as to the problem of performing symbolic manipulations of logical expressions. Again, no attempt was made to create a program with a large amount of knowledge about a particular problem domain. Only quite simple tasks were selected.

As AI research progressed and techniques for handling larger amounts of world knowledge were developed, some progress was made on the tasks just described and new tasks could reasonably be attempted. These include perception (vision and speech), natural language understanding, and problem solving in specialized domains such as medical diagnosis and chemical analysis.

Perception of the world around us is crucial to our survival. Animals with much less intelligence than people are capable of more sophisticated visual perception than are current machines. Perceptual tasks are difficult because they involve analog (rather than digital) signals; the signals are typically very noisy and usually a large number of things (some of which may be partially obscuring others) must be perceived at once. The problems of perception are discussed in greater detail in Chapter 21.

The ability to use language to communicate a wide variety of ideas is perhaps the most important thing that separates humans from the other animals. The problem of understanding spoken language is a perceptual problem and is hard to solve for the reasons just discussed. But suppose we simplify the problem by restricting it to written language. This problem, usually referred to as *natural language understanding*, is still extremely difficult. In order to understand sentences about a topic, it is necessary to know not only a lot about the language itself (its vocabulary and grammar) but also a good deal about the topic so that unstated assumptions can be recognized. We discuss this problem again later in this chapter and then in more detail in Chapter 15.

In addition to these mundane tasks, many people can also perform one or maybe more specialized tasks in which carefully acquired expertise is necessary. Examples of such tasks include engineering design, scientific discovery, medical diagnosis, and financial planning. Programs that can solve problems in these domains also fall under the aegis of artificial intelligence. Figure 1.1 lists some of the tasks that are the targets of work in AI.

A person who knows how to perform tasks from several of the categories shown in the figure learns the necessary skills in a standard order. First perceptual, linguistic, and commonsense skills are learned. Later (and of course for some people, never) expert skills such as engineering, medicine, or finance are acquired. It might seem to make sense then that the earlier skills are easier and thus more amenable to computerized duplication than are the later, more specialized ones. For this reason, much of the initial AI work was concentrated in those early areas. But it turns out that this naive assumption is not right. Although expert skills require knowledge that many of us do not have, they

Mundane Tasks

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning (It includes reasoning about physical objects and their relationship to each other)
- Robot control

Formal Tasks

- Games
  - Chess
  - Backgammon
  - Checkers
  - Go
- Mathematics
  - Geometry
  - Logic
  - Integral calculus
  - Proving properties of programs

Expert Tasks

- Engineering
  - Design
  - Fault finding
  - Manufacturing planning
- Scientific analysis
- Medical diagnosis
- Financial analysis

Figure 1.1: Some of the Task Domains of Artificial Intelligence ✓

AI – Com[illegible]

often require much less knowledge than do the more mundane skills and that knowledge is usually easier to represent and deal with inside programs.

As a result, the problem areas where AI is now flourishing most as a practical discipline (as opposed to a purely research one) are primarily the domains that require only specialized expertise without the assistance of commonsense knowledge. There are now thousands of programs called *expert systems* in day-to-day operation throughout all areas of industry and government. Each of these systems attempts to solve part, or perhaps all, of a practical, significant problem that previously required scarce human expertise. In Chapter 20 we examine several of these systems and explore techniques for constructing them.

Before embarking on a study of specific AI problems and solution techniques, it is important at least to discuss, if not to answer, the following four questions:

1. What are our underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level of detail, if at all, are we trying to model human intelligence?
4. How will we know when we have succeeded in building an intelligent program?

The next four sections of this chapter address these questions. Following that is a survey of some AI books that may be of interest and a summary of the chapter.

1.3 What Is an AI Technique?

Artificial intelligence problems span a very broad spectrum. They appear to have very little in common except that they are hard. Are there any techniques that are appropriate for the solution of a variety of these problems? The answer to this question is yes, there are. What, then, if anything, can we say about those techniques besides the fact that they manipulate symbols? How could we tell if those techniques might be useful in solving other problems, perhaps ones not traditionally regarded as AI tasks? The rest of this book is an attempt to answer those questions in detail. But before we begin examining closely the individual techniques, it is enlightening to take a broad look at them to see what properties they ought to possess.

One of the few hard and fast results to come out of the first three decades of AI research is that *intelligence requires knowledge*. To compensate for its one overpowering asset, indispensability, knowledge possesses some less desirable properties, including:

- ✓It is voluminous. big or widely
- ✓It is hard to characterize accurately. correctly
- ✓It is constantly changing.
- ✓It differs from data by being organized in a way that corresponds to the ways it will be used. (organization & usage differ)

So where does this leave us in our attempt to define AI techniques? We are forced to conclude that an AI technique is a method that exploits knowledge that should be represented in such a way that:

- ✓The knowledge captures generalizations. take by force In other words, it is not necessary to represent separately each individual situation. Instead, situations that share important properties are grouped together. If knowledge does not have this property, inordinate amounts of memory and updating will be required. So we usually call something without this property "data" rather than knowledge.
- It can be understood by people who must provide it. Although for many programs, the bulk of the data can be acquired automatically (for example, by taking readings from a variety of instruments), in many AI domains, most of the knowledge a program has must ultimately be provided by people in terms they understand.
- It can easily be modified to correct errors and to reflect changes in the world and in our world view.
- It can be used in a great many situations even if it is not totally accurate or complete.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must usually be considered.

Although AI techniques must be designed in keeping with these constraints imposed y AI problems, there is some degree of independence between problems and problem-olving techniques. It is possible to solve AI problems without using AI techniques (although, as we suggested above, those solutions are not likely to be very good). And it is possible to apply AI techniques to the solution of non-AI problems. This is likely to be a good thing to do for problems that possess many of the same characteristics as do AI problems. In order to try to characterize AI techniques in as problem-independent a way as possible, let's look at two very different problems and a series of approaches for solving each of them.

1.3.1 Tic-Tac-Toe

In this section, we present a series of three programs to play tic-tac-toe. The programs in this series increase in:

- Their complexity
- Their use of generalizations
- The clarity of their knowledge
- The extensibility of their approach

Thus they move toward being representations of what we call AI techniques.

Program 1

Data Structures

Board — A nine-element vector representing the board, where the elements of the vector correspond to the board positions as follows:

1	2	3
4	5	6
7	8	9

An element contains the value 0 if the corresponding square is blank, 1 if it is filled with an X, or 2 if it is filled with an O.

Movetable — A large vector of 19,683 elements (3^9), each element of which is a nine-element vector. The contents of this vector are chosen specifically to allow the algorithm to work.

The Algorithm

To make a move, do the following:

1. View the vector Board as a ternary (base three) number. Convert it to a decimal number.
2. Use the number computed in step 1 as an index into Movetable and access the vector stored there.
3. The vector selected in step 2 represents the way the board will look after the move that should be made. So set Board equal to that vector.

Comments

This program is very efficient in terms of time. And, in theory, it could play an optimal game of tic-tac-toe. But it has several disadvantages:

- It takes a lot of space to store the table that specifies the correct move to make from each board position.
- Someone will have to do a lot of work specifying all the entries in the movetable.
- It is very unlikely that all the required movetable entries can be determined and entered without any errors.
- If we want to extend the game, say to three dimensions, we would have to start from scratch, and in fact this technique would no longer work at all, since 3^{27} board positions would have to be stored, thus overwhelming present computer memories.

The technique embodied in this program does not appear to meet any of our requirements for a good AI technique. Let's see if we can do better.

Program 2

Data Structures

Board	A nine-element vector representing the board, as described for Program 1. But instead of using the numbers 0, 1, or 2 in each element, we store 2 (indicating blank), 3 (indicating X), or 5 (indicating O).
Turn	An integer indicating which move of the game is about to be played; 1 indicates the first move, 9 the last.

The Algorithm

The main algorithm uses three subprocedures:

Make2	Returns 5 if the center square of the board is blank, that is, if Board[5] = 2. Otherwise, this function returns any blank noncorner square (2, 4, 6, or 8).
Posswin(*p*)	Returns 0 if player *p* cannot win on his next move; otherwise, it returns the number of the square that constitutes a winning move. This function will enable the program both to win and to block the opponent's win. Posswin operates by checking, one at a time, each of the rows, columns, and diagonals. Because of the way values are numbered, it can test an entire row (column or diagonal) to see if it is a possible win by multiplying the values of its squares together. If the product is 18 (3 x 3 x 2), then X can win. If the product is 50 (5 x 5 x 2), then O can win. If we find a winning row, we determine which element is blank, and return the number of that square.

Go(*n*)	Makes a move in square *n*. This procedure sets Board[*n*] to 3 if Turn is odd, or 5 if Turn is even. It also increments Turn by one.

The algorithm has a built-in strategy for each move it may have to make. It makes the odd-numbered moves if it is playing X, the even-numbered moves if it is playing O. The strategy for each turn is as follows:

Turn=1	Go(1) (upper left corner).
Turn=2	If Board[5] is blank, Go(5), else Go(1).
Turn=3	If Board[9] is blank, Go(9), else Go(3).
Turn=4	If Posswin(X) is not 0, then Go(Posswin(X)) [i.e., block opponent's win], else Go(Make2).
Turn=5	If Posswin(X) is not 0 then Go(Posswin(X)) [i.e., win] else if Posswin(O) is not 0, then Go(Posswin(O)) [i.e., block win], else if Board[7] is blank, then Go(7), else Go(3). [Here the program is trying to make a fork.]
Turn=6	If Posswin(O) is not 0 then Go(Posswin(O)), else if Posswin(X) is not 0, then Go(Posswin(X)), else Go(Make2).
Turn=7	If Posswin(X) is not 0 then Go(Posswin(X)), else if Posswin(O) is not 0, then Go(Posswin(O)), else go anywhere that is blank.
Turn=8	If Posswin(O) is not 0 then Go(Posswin(O)), else if Posswin(X) is not 0, then Go(Posswin(X)), else go anywhere that is blank.
Turn=9	Same as Turn=7.

Comments

This program is not quite as efficient in terms of time as the first one since it has to check several conditions before making each move. But it is a lot more efficient in terms of space. It is also a lot easier to understand the program's strategy or to change the strategy if desired. But the total strategy has still been figured out in advance by the programmer. Any bugs in the programmer's tic-tac-toe playing skill will show up in the program's play. And we still cannot generalize any of the program's knowledge to a different domain, such as three-dimensional tic-tac-toe.

Program 2′

This program is identical to Program 2 except for one change in the representation of the board. We again represent the board as a nine-element vector, but this time we assign board positions to vector elements as follows:

8	3	4
1	5	9
6	7	2

Notice that this numbering of the board produces a magic square: all the rows, columns, and diagonals sum to 15. This means that we can simplify the process of checking for a possible win. In addition to marking the board as moves are made, we keep a list, for each player, of the squares in which he or she has played. To check for a possible win for one player, we consider each pair of squares owned by that player and compute the difference between 15 and the sum of the two squares. If this difference is not positive or if it is greater than 9, then the original two squares were not collinear and so can be ignored. Otherwise, if the square representing the difference is blank, a move there will produce a win. Since no player can have more than four squares at a time, there will be many fewer squares examined using this scheme than there were using the more straightforward approach of Program 2. This shows how the choice of representation can have a major impact on the efficiency of a problem-solving program.

Comments

This comparison raises an interesting question about the relationship between the way people solve problems and the way computers do. Why do people find the row-scan approach easier while the number-counting approach is more efficient for a computer? We do not know enough about how people work to answer that question completely. One part of the answer is that people are parallel processors and can look at several parts of the board at once, whereas the conventional computer must look at the squares one at a time. Sometimes an investigation of how people solve problems sheds great light on how computers should do so. At other times, the differences in the hardware of the two seem so great that different strategies seem best. As we learn more about problem solving both by people and by machines, we may know better whether the same representations and algorithms are best for both people and machines. We will discuss this question further in Section 1.4.

Program 3

Data Structures

BoardPosition	A structure containing a nine-element vector representing the board, a list of board positions that could result from the next move, and a number representing an estimate of how likely the board position is to lead to an ultimate win for the player to move.

The Algorithm

To decide on the next move, look ahead at the board positions that result from each possible move. Decide which position is best (as described below), make the move that leads to that position, and assign the rating of that best move to the current position.

To decide which of a set of board positions is best, do the following for each of them:

1. See if it is a win. If so, call it the best by giving it the highest possible rating.

2. Otherwise, consider all the moves the opponent could make next. See which of them is worst for us (by recursively calling this procedure). Assume the opponent will make that move. Whatever rating that move has, assign it to the node we are considering.

3. The best node is then the one with the highest rating.

This algorithm will look ahead at various sequences of moves in order to find a sequence that leads to a win. It attempts to maximize the likelihood of winning, while assuming that the opponent will try to minimize that likelihood. This algorithm is called the *minimax procedure*, and it is discussed in detail in Chapter 12.

Comments

This program will require much more time than either of the others since it must search a tree representing all possible move sequences before making each move. But it is superior to the other programs in one very big way: It could be extended to handle games more complicated than tic-tac-toe, for which the exhaustive enumeration approach of the other programs would completely fall apart. It can also be augmented by a variety of specific kinds of knowledge about games and how to play them. For example, instead of considering all possible next moves, it might consider only a subset of them that are determined, by some simple algorithm, to be reasonable. And, instead of following each series of moves until one player wins, it could search for a limited time and evaluate the merit of each resulting board position using some static function.

Program 3 is an example of the use of an AI technique. For very small problems, it is less efficient than a variety of more direct methods. However, it can be used in situations where those methods would fail.

1.3.2 Question Answering

In this section we look at a series of programs that read in English text and then answer questions, also stated in English, about that text. This task differs from the last one in that it is more difficult now to state formally and precisely what our problem is and what constitutes correct solutions to it. For example, suppose that the input text were just the single sentence

Russia massed troops on the Czech border.

Then either of the following question-answering dialogues might occur (and in fact did occur with the POLITICS program [Carbonell, 1980]):

Dialogue 1

Q: Why did Russia do this?

A: Because Russia thought that it could take political control of Czechoslovakia by sending troops.

Q: What should the United States do?

A: The United States should intervene militarily.

Dialogue 2

Q: Why did Russia do this?

A: Because Russia wanted to increase its political influence over Czechoslovakia.

Q: What should the United States do?

A: The United States should denounce the Russian action in the United Nations.

In the POLITICS program, answers were constructed by considering both the input text and a separate model of the beliefs and actions of various political entities, including Russia. When the model is changed, as it was between these two dialogues, the system's answers also change. In this example, the first dialogue was produced when POLITICS was given a model that was intended to correspond to the beliefs of a typical American conservative (circa 1977). The second dialogue occurred when POLITICS was given a model that was intended to correspond to the beliefs of a typical American liberal (of the same vintage).

The general point here is that defining what it means to produce a *correct* answer to a question may be very hard. Usually, question-answering programs define what it means to be an answer by the procedure that is used to compute the answer. Then their authors appeal to other people to agree that the answers found by the program "make sense" and so to confirm the model of question answering defined in the program. This is not completely satisfactory, but no better way of defining the problem has yet been found. For lack of a better method, we will do the same here and illustrate three definitions of question answering, each with a corresponding program that implements the definition.

In order to be able to compare the three programs, we illustrate all of them using the following text:

> Mary went shopping for a new coat. She found a red one she really liked. When she got it home, she discovered that it went perfectly with her favorite dress.

We will also attempt to answer each of the following questions with each program:

Q1: What did Mary go shopping for?

Q2: What did Mary find that she liked?

Q3: Did Mary buy anything?

Program 1

This program attempts to answer questions using the literal input text. It simply matches text fragments in the questions against the input text.

Data Structures

QuestionPatterns	A set of templates that match common question forms and produce patterns to be used to match against inputs. Templates and patterns (which we call *text patterns*) are paired so that if a template matches successfully against an input question then its associated text patterns are used to try to find appropriate answers in the text. For

	example, if the template "Who did x y" matches an input question, then the text pattern "x y z" is matched against the input text and the value of z is given as the answer to the question.
Text	The input text stored simply as a long character string.
Question	The current question also stored as a character string.

The Algorithm

To answer a question, do the following:

1. Compare each element of QuestionPatterns against the Question and use all those that match successfully to generate a set of text patterns.
2. Pass each of these patterns through a substitution process that generates alternative forms of verbs so that, for example, "go" in a question might match "went" in the text. This step generates a new, expanded set of text patterns.
3. Apply each of these text patterns to Text, and collect all the resulting answers.
4. Reply with the set of answers just collected.

Examples

Q1: The template "What did x y" matches this question and generates the text pattern "Mary go shopping for z." After the pattern-substitution step, this pattern is expanded to a set of patterns including "Mary goes shopping for z," and "Mary went shopping for z." The latter pattern matches the input text; the program, using a convention that variables match the longest possible string up to a sentence delimiter (such as a period), assigns z the value, "a new coat," which is given as the answer.

Q2: Unless the template set is very large, allowing for the insertion of the object of "find" between it and the modifying phrase "that she liked," the insertion of the word "really" in the text, and the substitution of "she" for "Mary," this question is not answerable. If all of these variations are accounted for and the question can be answered, then the response is "a red one."

Q3: Since no answer to this question is contained in the text, no answer will be found.

Comments

This approach is clearly inadequate to answer the kinds of questions people could answer after reading a simple text. Even its ability to answer the most direct questions is delicately dependent on the exact form in which questions are stated and on the variations that were anticipated in the design of the templates and the pattern substitutions that the system uses. In fact, the sheer inadequacy of this program to perform the task may make you wonder how such an approach could even be proposed. This program is substantially farther away from being useful than was the initial program we looked at for tic-tac-toe. Is this just a strawman designed to make some other technique look good in comparison? In a way, yes, but it is worth mentioning that the approach that this program uses, namely matching patterns, performing simple text substitutions, and then forming answers using straightforward combinations of canned text and sentence fragments located by the matcher, is the same approach that is used in one of the most famous "AI" programs ever written—ELIZA, which we discuss in Section 6.4.3. But, as you read the rest of this sequence of programs, it should become clear that what we mean by the term "artificial intelligence" does not include programs such as this except by a substantial stretching of definitions.

Program 2

This program first converts the input text into a structured internal form that attempts to capture the meaning of the sentences. It also converts questions into that form. It finds answers by matching structured forms against each other.

Data Structures

EnglishKnow	A description of the words, grammar, and appropriate semantic interpretations of a large enough subset of English to account for the input texts that the system will see. This knowledge of English is used both to map input sentences into an internal, meaning-oriented form and to map from such internal forms back into English. The former process is used when English text is being read; the latter is used to generate English answers from the meaning-oriented form that constitutes the program's knowledge base.
InputText	The input text in character form.
StructuredText	A structured representation of the content of the input text. This structure attempts to capture the essential knowledge contained in the text, independently of the exact way that the knowledge was stated in English. Some things that were not explicit in the English text, such as the referents of pronouns, have been made explicit in this form. Representing knowledge such as this is an important issue in the design of almost all AI programs. Existing programs exploit a variety of frameworks for doing this. There are three important families of such *knowledge representation* systems: production rules (of the form "if *x* then *y*"), slot-and-filler structures, and statements in mathematical logic. We discuss all of these methods later in substantial detail, and we look at key questions that need to be answered in order to choose a method for a particular program. For now though, we just pick one arbitrarily. The one we've chosen is a slot-and-filler structure. For example, the sentence "She found a red one she really liked," might be represented as shown in Figure 1.2. Actually, this is a simplified description of the contents of the sentence. Notice that it is not very explicit about temporal relationships (for example, events are just marked as past tense) nor have we made any real attempt to represent the meaning of the qualifier "really." It should, however, illustrate the basic form that representations such as this take. One of the key ideas in this sort

Event2	
instance :	*Finding*
tense :	*Past*
agent :	*Mary*
object :	*Thing1*
Thing1	
instance :	*Coat*
color :	*Red*
Event2	
instance :	*Liking*
tense :	*Past*
modifier :	*Much*
object :	*Thing1*

Figure 1.2: A Structured Representation of a Sentence

	of representation is that entities in the representation derive their meaning from their connections to other entities. In the figure, only the entities defined by the sentence are shown. But other entities corresponding to concepts that the program knew about before it read this sentence, also exist in the representation and can be referred to within these new structures. In this example, for instance, we refer to the entities *Mary*, *Coat* (the general concept of a coat of which *Thing1* is a specific instance), *Liking* (the general concept of liking), and *Finding* (the general concept of finding).
InputQuestion	The input question in character form.
StructQuestion	A structured representation of the content of the user's question. The structure is the same as the one used to represent the content of the input text.

The Algorithm

Convert the InputText into structured form using the knowledge contained in EnglishKnow. This may require considering several different potential structures, for a variety of reasons, including the fact that English words can be ambiguous, English grammatical structures can be ambiguous, and pronouns may have several possible antecedents.

Then, to answer a question, do the following:

1. Convert the question to structured form, again using the knowledge contained in EnglishKnow. Use some special marker in the structure to indicate the part of the structure that should be returned as the answer. This marker will often correspond

to the occurrence of a question word (like "who" or "what") in the sentence. The exact way in which this marking gets done depends on the form chosen for representing StructuredText. If a slot-and-filler structure, such as ours, is used, a special marker can be placed in one or more slots. If a logical system is used, however, markers will appear as variables in the logical formulas that represent the question.

2. Match this structured form against StructuredText.

3. Return as the answer those parts of the text that match the requested segment of the question.

Examples

Q1: This question is answered straightforwardly with, "a new coat."

Q2: This one also is answered successfully with, "a red coat."

Q3: This one, though, cannot be answered, since there is no direct response to it in the text.

Comments

This approach is substantially more meaning (knowledge)-based than that of the first program and so is more effective. It can answer most questions to which replies are contained in the text, and it is much less brittle than the first program with respect to the exact forms of the text and the questions. As we expect, based on our experience with the pattern recognition and tic-tac-toe programs, the price we pay for this increased flexibility is time spent searching the various knowledge bases (i.e., EnglishKnow, StructuredText).

One word of warning is appropriate here. The problem of producing a knowledge base for English that is powerful enough to handle a wide range of English inputs is very difficult. It is discussed at greater length in Chapter 15. In addition, it is now recognized that knowledge of English alone is not adequate in general to enable a program to build the kind of structured representation shown here. Additional knowledge about the world with which the text deals is often required to support lexical and syntactic disambiguation and the correct assignment of antecedents to pronouns, among other things. For example, in the text

> Mary walked up to the salesperson. She asked where the toy department was.

it is not possible to determine what the word "she" refers to without knowledge about the roles of customers and salespeople in stores. To see this, contrast the correct antecedent of "she" in that text with the correct antecedent for the first occurrence of "she" in the following example:

> Mary walked up to the salesperson. She asked her if she needed any help.

In the simple case illustrated in our coat-buying example, it is possible to deri correct answers to our first two questions without any additional knowledge about stores or coats, and the fact that some such additional information may be necessary to suppo question answering has already been illustrated by the failure of this program to find an answer to question 3. Thus we see that although extracting a structured representatio of the meaning of the input text is an improvement over the meaning-free approach o Program 1, it is by no means sufficient in general. So we need to look at an even mor sophisticated (i.e., knowledge-rich) approach, which is what we do next.

Program 3

This program converts the input text into a structured form that contains the meaning of the sentences in the text, and then it combines that form with other structured forms that describe prior knowledge about the objects and situations involved in the text. I answers questions using this augmented knowledge structure.

Data Structures

WorldModel	A structured representation of background world knowledge. This structure contains knowledge about objects, actions, and situation that are described in the input text. This structure is used to construct IntegratedText from the input text. For example, Figure 1.3 shows ar example of a structure that represents the system's knowledge about shopping. This kind of stored knowledge about stereotypical event is called a *script* and is discussed in more detail in Section 10.2. The notation used here differs from the one normally used in the literature for the sake of simplicity. The prime notation describe an object of the same type as the unprimed symbol that may or may not refer to the identical object. In the case of our text, for example M is a coat and M′ is a red coat. Branches in the figure describe alternative paths through the script.
EnglishKnow	Same as in Program 2.
InputText	The input text in character form.
IntegratedText	A structured representation of the knowledge contained in the input text (similar to the structured description of Program 2) bu combined now with other background, related knowledge.
InputQuestion	The input question in character form.
StructQuestion	A structured representation of the question.

The Algorithm

Convert the InputText into structured form using both the knowledge contained in EnglishKnow and that contained in WorldModel. The number of possible structures will usually be greater now than it was in Program 2 because so much more knowledge is being used. Sometimes, though, it may be possible to consider fewer possibilities by using the additional knowledge to filter the alternatives.

Shopping Script:

roles: C (customer), S (salesperson)
props: M (merchandise), D (dollars)
location: L (a store)

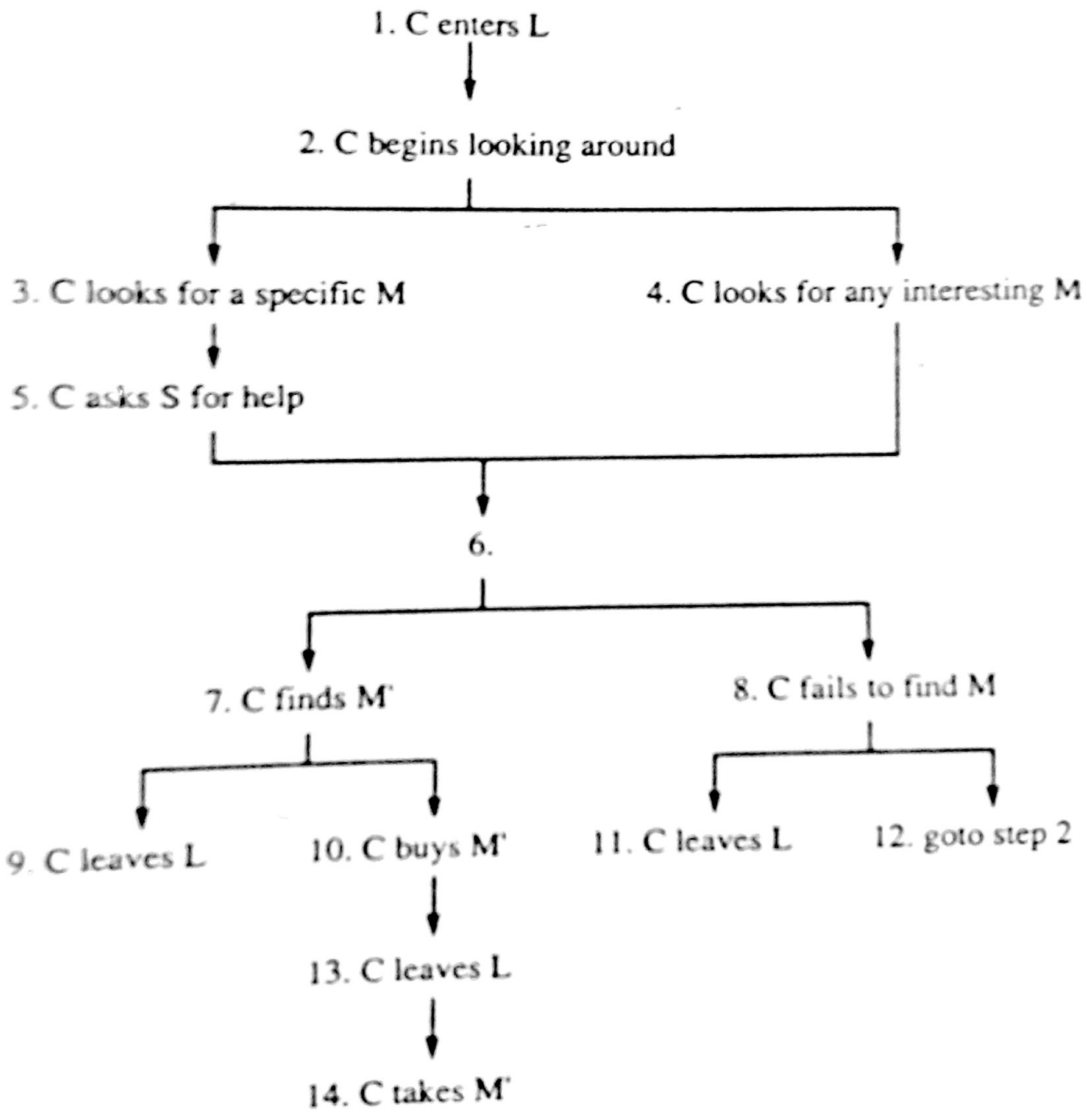


Figure 1.3: A Shopping Script

To answer a question, do the following:

1. Convert the question to structured form as in Program 2 but use WorldModel if necessary to resolve any ambiguities that may arise.
2. Match this structured form against IntegratedText.
3. Return as the answer those parts of the text that match the requested segment of the question.

Examples

Q1: Same as Program 2.

Q2: Same as Program 2.

Q3: Now this question can be answered. The shopping script is instantiated for this text, and because of the last sentence, the path through step 14 of the script is the one that is used in forming the representation of this text. When the script is instantiated M' is bound to the structure representing the red coat (because the script says that M' is what gets taken home and the text says that a red coat is what got taken home). After the script has been instantiated, IntegratedText contains several events that are taken from the script but that are not described in the original text, including the event "Mary buys a red coat" (from step 10 of the script). Thus, using the integrated text as the basis for question answering allows the program to respond "She bought a red coat."

Comments

This program is more powerful than either of the first two because it exploits more knowledge. Thus it, like the final program in each of the other two sequences we have examined, is exploiting what we call AI techniques. But, again, a few caveats are in order. Even the techniques we have exploited in this program are not adequate for complete English question answering. The most important thing that is missing from this program is a general reasoning (inference) mechanism to be used when the requested answer is not contained explicitly even in IntegratedText, but that answer does follow logically from the knowledge that is there. For example, given the text

> Saturday morning Mary went shopping. Her brother tried to call her then, but he couldn't get hold of her.

it should be possible to answer the question

> Why couldn't Mary's brother reach her?

with the reply

> Because she wasn't home.

But to do so requires knowing that one cannot be two places at once and then using that fact to conclude that Mary could not have been home because she was shopping instead. Thus, although we avoided the inference problem temporarily by building IntegratedText, which had some obvious inferences built into it, we cannot avoid it forever. It is simply not practical to anticipate all legitimate inferences. In later chapters, we look at ways of providing a general inference mechanism that could be used to support a program such as the last one in this series.

This limitation does not contradict the main point of this example though. In fact, it is additional evidence for that point, namely, an effective question-answering procedure must be one based soundly on knowledge and the computational use of that knowledge. The purpose of AI techniques is to support this effective use of knowledge.

1.3.3 Conclusion

We have just examined two series of programs to solve two very different problems. In each series, the final program exemplifies what we mean by an AI technique. These two programs are slower to execute than the earlier ones in their respective series, but they illustrate three important AI techniques:

- Search—Provides a way of solving problems for which no more direct approach is available as well as a framework into which any direct techniques that are available can be embedded.
- Use of Knowledge—Provides a way of solving complex problems by exploiting the structures of the objects that are involved.
- Abstraction—Provides a way of separating important features and variations from the many unimportant ones that would otherwise overwhelm any process.

For the solution of hard problems, programs that exploit these techniques have several advantages over those that do not. They are much less fragile; they will not be thrown off completely by a small perturbation in their input. People can easily understand what the program's knowledge is. And these techniques can work for large problems where more direct methods break down.

We have still not given a precise definition of an AI technique. It is probably not possible to do so. But we have given some examples of what one is and what one is not. Throughout the rest of this book, we talk in great detail about what one is. The definition should then become a bit clearer, or less necessary.

1.5 Criteria for Success

One of the most important questions to answer in any scientific or engineering research project is "How will we know if we have succeeded?" Artificial intelligence is no exception. How will we know if we have constructed a machine that is intelligent? That question is at least as hard as the unanswerable question "What is intelligence?" But can we do anything to measure our progress?

In 1950, Alan Turing proposed the following method for determining whether a machine can think. His method has since become known as the *Turing test*. To conduct this test, we need two people and the machine to be evaluated. One person plays the role of the interrogator, who is in a separate room from the computer and the other person. The interrogator can ask questions of either the person or the computer by typing questions and receiving typed responses. However, the interrogator knows them only as A and B and aims to determine which is the person and which is the machine. The goal of the machine is to fool the interrogator into believing that it is the person. If the machine succeeds at this, then we will conclude that the machine can think. The machine is allowed to do whatever it can to fool the interrogator. So, for example, if asked the question "How much is 12,324 times 73,981?" it could wait several minutes and then respond with the wrong answer [Turing, 1963].

The more serious issue, though, is the amount of knowledge that a machine would need to pass the Turing test. Turing gives the following example of the sort of dialogue a machine would have to be capable of:

Interrogator: In the first line of your sonnet which reads "Shall I compare thee to a summer's day," would not "a spring day" do as well or better?

A: It wouldn't scan.

Interrogator: How about "a winter's day." That would scan all right.

A: Yes, but nobody wants to be compared to a winter's day.

Interrogator: Would you say Mr. Pickwick reminded you of Christmas?

A: In a way.

Interrogator: Yet Christmas is a winter's day, and I do not think Mr. Pickwick would mind the comparison.

A: I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

It will be a long time before a computer passes the Turing test. Some people believe none ever will. But suppose we are willing to settle for less than a complete imitation of a person. Can we measure the achievement of AI in more restricted domains?

Often the answer to this question is yes. Sometimes it is possible to get a fairly precise measure of the achievement of a program. For example, a program can acquire a chess rating in the same way as a human player. The rating is based on the ratings of players whom the program can beat. Already programs have acquired chess ratings higher than the vast majority of human players. For other problem domains, a less precise measure of a program's achievement is possible. For example, DENDRAL is a program that analyzes organic compounds to determine their structure. It is hard to get a precise measure of DENDRAL's level of achievement compared to human chemists, but it has produced analyses that have been published as original research results. Thus it is certainly performing competently.

In other technical domains, it is possible to compare the time it takes for a program to complete a task to the time required by a person to do the same thing. For example, there are several programs in use by computer companies to configure particular systems to customers' needs (of which the pioneer was a program called R1). These programs typically require minutes to perform tasks that previously required hours of a skilled engineer's time. Such programs are usually evaluated by looking at the bottom line—whether they save (or make) money.

For many everyday tasks, though, it may be even harder to measure a program's performance. Suppose, for example, we ask a program to paraphrase a newspaper story. For problems such as this, the best test is usually just whether the program responded in a way that a person could have.

If our goal in writing a program is to simulate human performance at a task, then the measure of success is the extent to which the program's behavior corresponds to that performance, as measured by various kinds of experiments and protocol analyses. In this we do not simply want a program that does as well as possible. We want one that fails when people do. Various techniques developed by psychologists for comparing individuals and for testing models can be used to do this analysis.

We are forced to conclude that the question of whether a machine has intelligence or can think is too nebulous to answer precisely. But it is often possible to construct a computer program that meets some performance standard for a particular task. That does not mean that the program does the task in the best possible way. It means only that we understand at least one way of doing at least part of a task. When we set out to design an AI program, we should attempt to specify as well as possible the criteria for success for that particular program functioning in its restricted domain. For the moment, that is the best we can do.

The process of solving the problem can be modeled as production system.

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 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.

- 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.

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

[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.

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 cause 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. Figure 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 Figure 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.

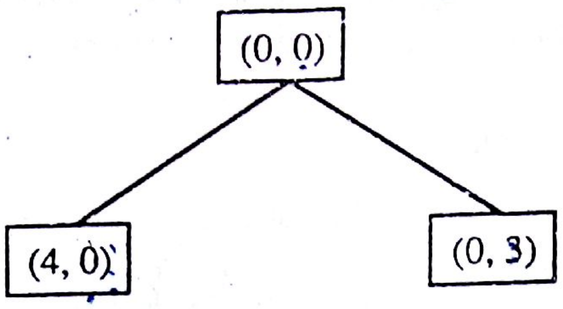
Algorithm: Breadth-First Search

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 do:

   (a) Remove the first element from *NODE-LIST* and call it *E*. If *NODE-LIST* was empty, quit.

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



0 0

0,3

[illegible]

Figure 2.5: One Level of a Breadth-First Search Tree

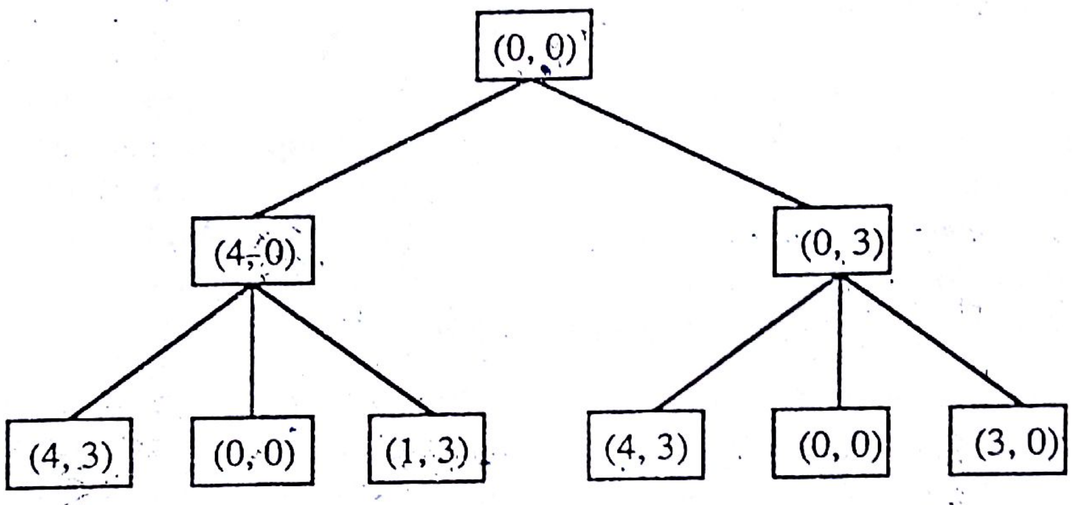


Figure 2.6: Two Levels of a Breadth-First Search Tree

(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.

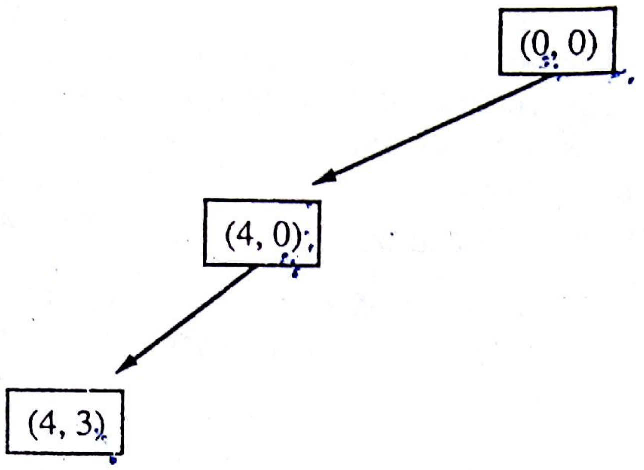


Figure 2.7: A Depth-First Search Tree

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.

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 Figure 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 $1 \cdot 2 \cdots (N-1)$, 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 (nearest neighbour heuristic)

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 *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 the 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.

[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.

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* if f 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* if f is the function kill.

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* [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 figures. Another is that the rules are very general.

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

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. Figure 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.

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.

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

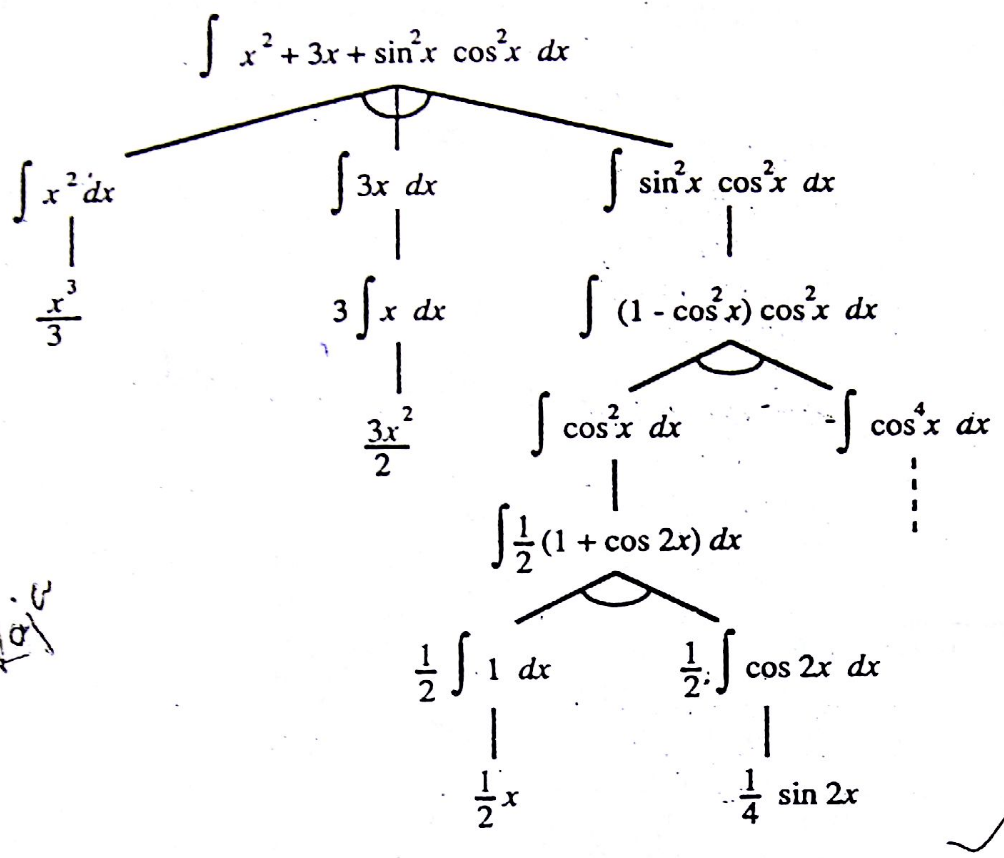
Figure 2.8: Some Simple Heuristic Functions

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?



$\int x^2 + 3x + \sin^2 x \cos^2 x \, dx$
$\int x^2 dx$
$\frac{x^3}{3}$
$\int 3x \, dx$
$3\int x \, dx$
$\frac{3x^2}{2}$
$\int \sin^2 x \cos^2 x \, dx$
$\int (1 - \cos^2 x) \cos^2 x \, dx$
$\int \cos^2 x \, dx$
$-\int \cos^4 x \, dx$
$\int \frac{1}{2}(1 + \cos 2x) \, dx$
$\frac{1}{2}\int 1 \, dx$
$\frac{1}{2}\int \cos 2x \, dx$
$\frac{1}{2}x$
$\frac{1}{4} \sin 2x$

Figure 2.9: A Decomposable Problem

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.

Now consider the problem illustrated in Figure 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:

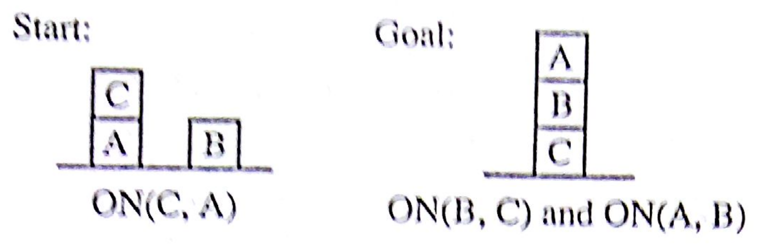


Figure 2.10: A Simple Blocks World Problem

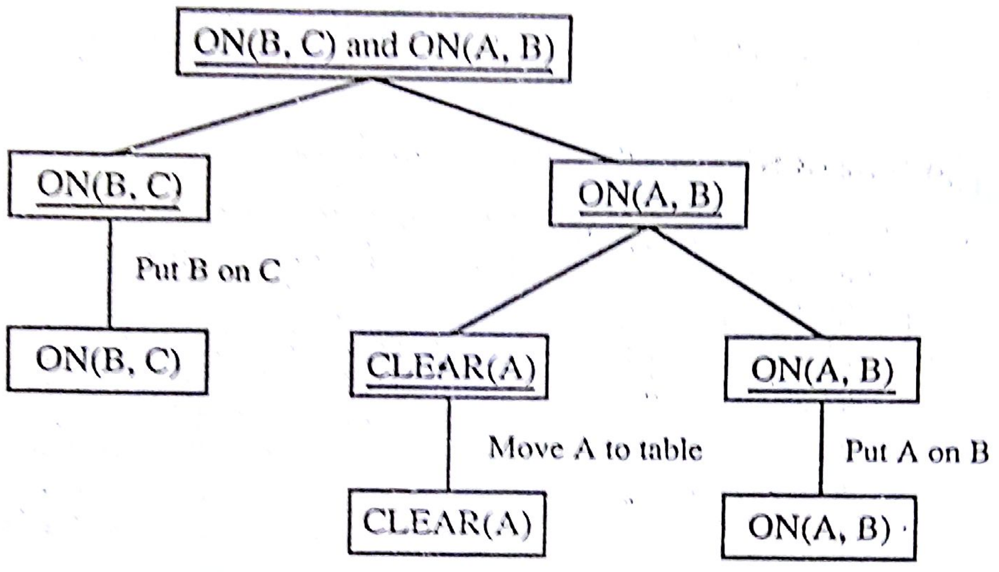


Figure 2.11: A Proposed Solution for the Blocks Problem

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 such as that shown in Figure 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.

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.

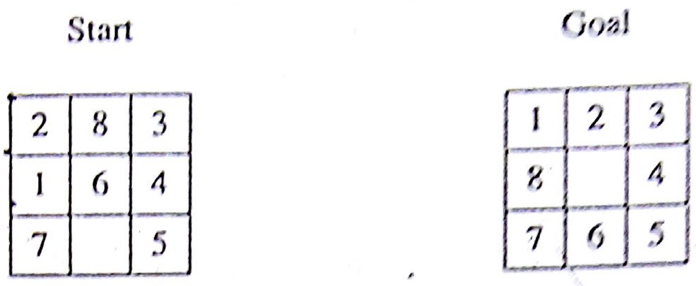


Figure 2.12: An Example of the 8-Puzzle

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 Figure 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.

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.

planning - is the [illegible] [illegible] [illegible] [illegible]
[illegible] environment

puzzle & bridge

→ 8-puzzle

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: (irrecoverable problems)

- 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 n a formal language, such as predicate logic, and then using formal inference methods,

		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

Figure 2.13: Two Ways of Deciding That Marcus Is Dead

	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	

Figure 2.14: An Instance of the Traveling Salesman Problem

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 Figure 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.

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 Figure 2.14.

One place the salesman could start is Boston. In that case, one path that might be followed is the one shown in Figure 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

[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.

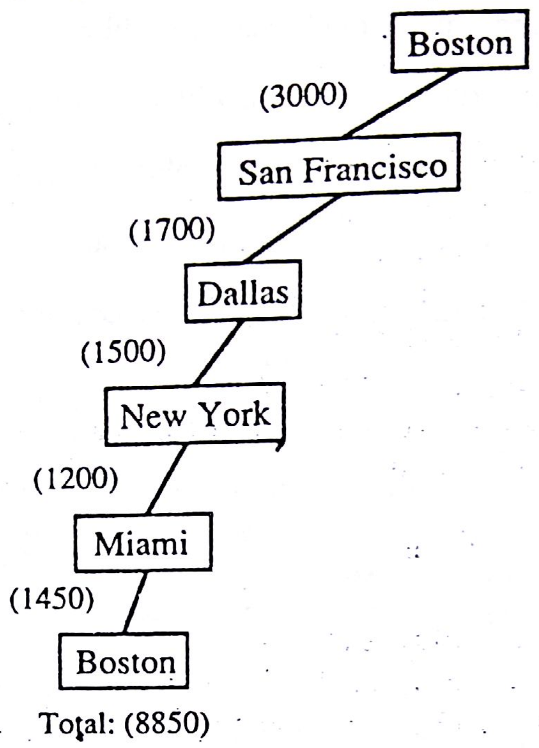


Figure 2.15: One Path among the Cities

other paths to make sure that none of them is shorter. In this case, as can be seen from Figure 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.

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.

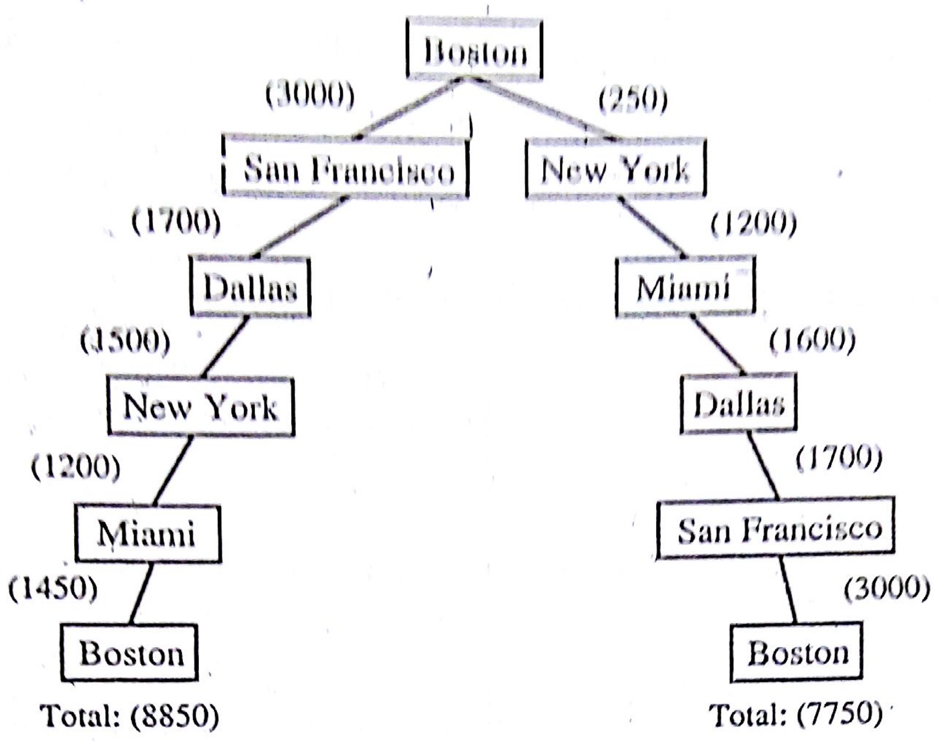


Figure 2.16: Two Paths Among the Cities

- 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 a *plan*) 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 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 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.

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 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 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 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 *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]

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. Figure 2.17 shows the four categories of production systems produced by the two dichotomies, monotonic versus nonmonotonic and partially commutative versus

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

	Monotonic	Nonmonotonic
Partially commutative	Theorem proving ignorable	Robot navigation Recoverable
Not partially commutative	Chemical synthesis [illegible]	Bridge [illegible]

Figure 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