CHAPTER 3
Methods of Inference

3.1 INTRODUCTION

This chapter is an introduction to reasoning about knowledge. It is important to separate the meaning of the words used in reasoning from the reasoning itself so we will cover it in a formal way by discussing various methods of inference (Russell 03). In particular we will look at an important type of reasoning in expert systems in which conclusions follow from facts by using rules. It is particularly important in expert systems because making inferences is the fundamental method by which expert systems solve problems (Klir 98).

As mentioned in Chapter 1, expert systems are commonly used when an inadequate algorithm or no algorithmic solution exists. The expert system should make a chain of inferences to come up with a solution just as a person would, especially when faced with incomplete or missing information. A simple computer program that uses an algorithm to solve a problem cannot supply an answer if all the parameters needed are not given.

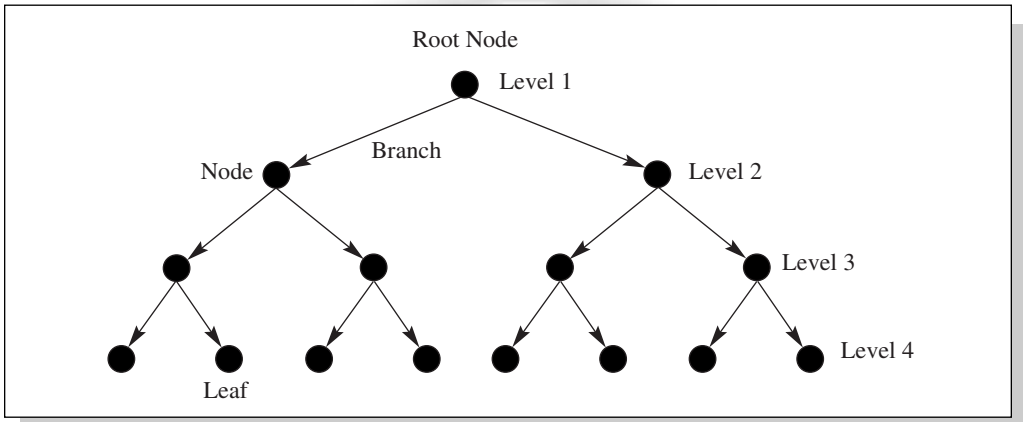
However an expert system makes a best guess just like a human would when a problem needs to be solved and the optimum solution cannot be determined. The idea is to come up with a solution just as people come up with a (hopefully) good solution rather than none at all. While the solution may only be 95% as efficient as the optimum solution, that's still better than no solution. This type of approach is really nothing new. For example, numeric computations in mathematics such as the Runge-Kutta method of solving differential equations have been used for centuries to solve problems for which no analytic solution is possible.

NOTE: Appendices A, B, and C contain useful reference material for this chapter.

3.2 TREES, LATTICES, AND GRAPHS

A **tree** is a hierarchical data structure consisting of **nodes,** which store information or knowledge and **branches,** which connect the nodes. Branches are sometimes called **links** or **edges** and nodes are sometimes called **vertices**. Figure 3.1 shows a general binary tree which has zero, one, or two branches per node. In an **oriented tree** the **root node** is the highest node in the **hierarchy** and the **leaves** are the lowest. A tree can be considered a special type of semantic net in which every node except the root has exactly one **parent** and zero or more **child** nodes. For the usual type of binary tree there is a maximum of two children per node, and the left and right child nodes are distinguished.

Figure 3.1 Binary Tree

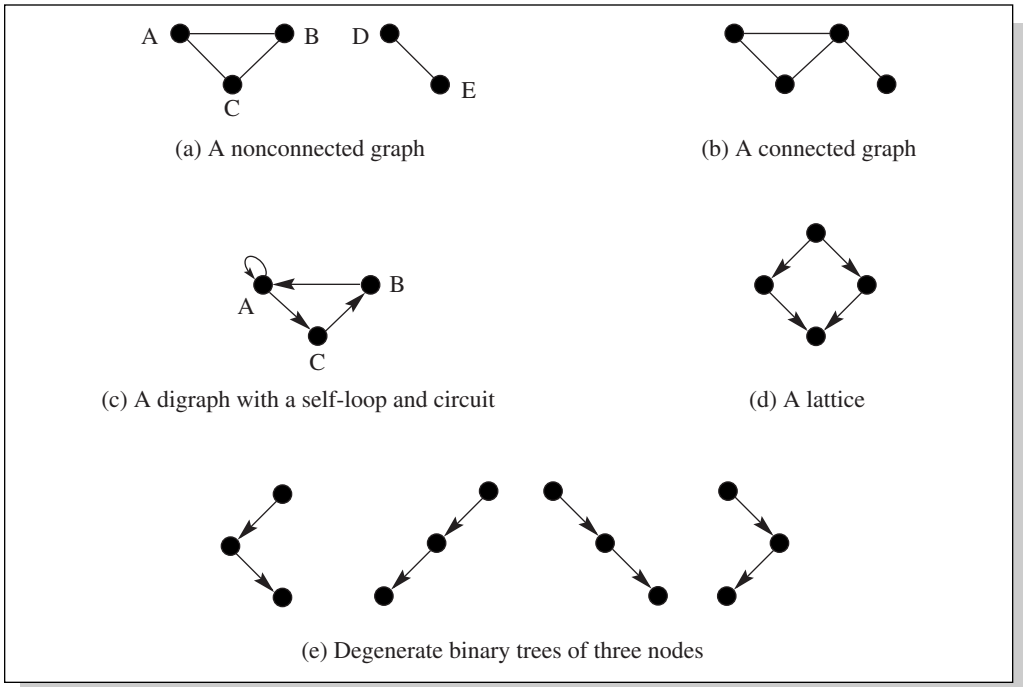


If a node has more than one parent, it is in a network. In Figure 3.1, notice that there is only one sequence of edges or **path** from the root to any node since it is not possible to move against an arrow. In oriented trees the arrows all point downward.

Trees are a special case of a general mathematical structure called a **graph**. The terms *network* or simply *net* are often used synonymously with graph when describing a particular example such as an ethernet network. A graph can have zero or more links between nodes and no distinction between parents and children. A simple example of a graph is a map, where the cities are the nodes and the links are the roads. The links may have arrows or directions associated with them and a **weight** to characterize some aspect of the link. An analogy is one-way streets with limits on how much weight trucks can carry. The weights in a graph can be any type of information. If the graph represents an airline route, the weights can be miles between cities, cost of flying, fuel consumption, and so forth.

A supervised artificial neural network is another example of a graph with cycles since during training there is feedback of information from one layer of the

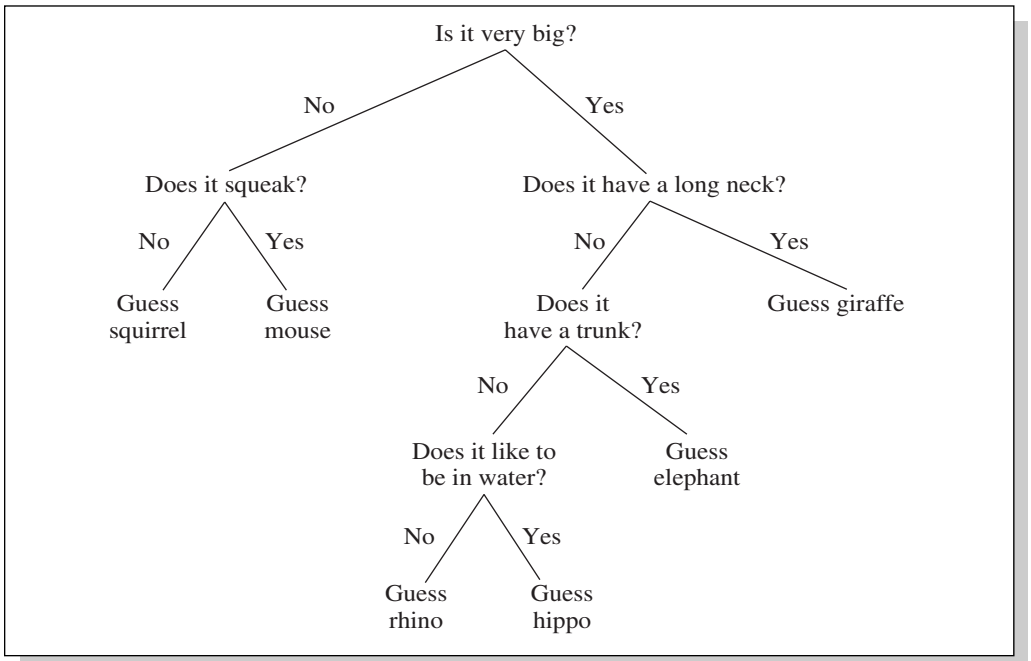
Figure 3.2 Simple Graphs



net to another, which modifies the weights. A simple graph has no links that come immediately back on the node itself, as shown in Figure 3.2a. A **circuit** or **cycle** is a path through a graph that begins and ends on the same node, as does the path ABCA in Figure 3.2a An **acyclic graph** has no cycles. A **connected graph** has links to all its nodes as shown in Fig. 3.2b. A graph with directed links, called a **digraph,** and a **self-loop** is shown in Figure 3.2c. A directed acyclic graph is a **lattice**, and an example is shown in Figure 3.2d. A tree with only a single path from the root to its one leaf is a **degenerate tree**. The degenerate binary trees of three nodes are shown in Figure 3.2e. Generally in a tree, the arrows are *not* explicitly shown because they are assumed to be pointing down.

Trees and lattices are useful for classifying objects because of their hierarchical nature, with parents above children. An example is a family tree, which shows the relationships and ancestry of related people. Another application of trees and lattices is making decisions; these are called **decision trees** or **decision lattices** and are very common for simple reasoning. We will use the term **structure** to refer to both trees and lattices. A decision structure is both a knowledge representation scheme and a method of reasoning about its knowledge. An example of a decision tree for classifying animals is shown in Figure 3.3. This example is for the classic game of twenty questions. The nodes contain questions, the branches "yes" or "no" responses to the questions, and the leaves contain the guesses of what the animal is.

Figure 3.3 Decision Tree Showing Knowledge About Animals

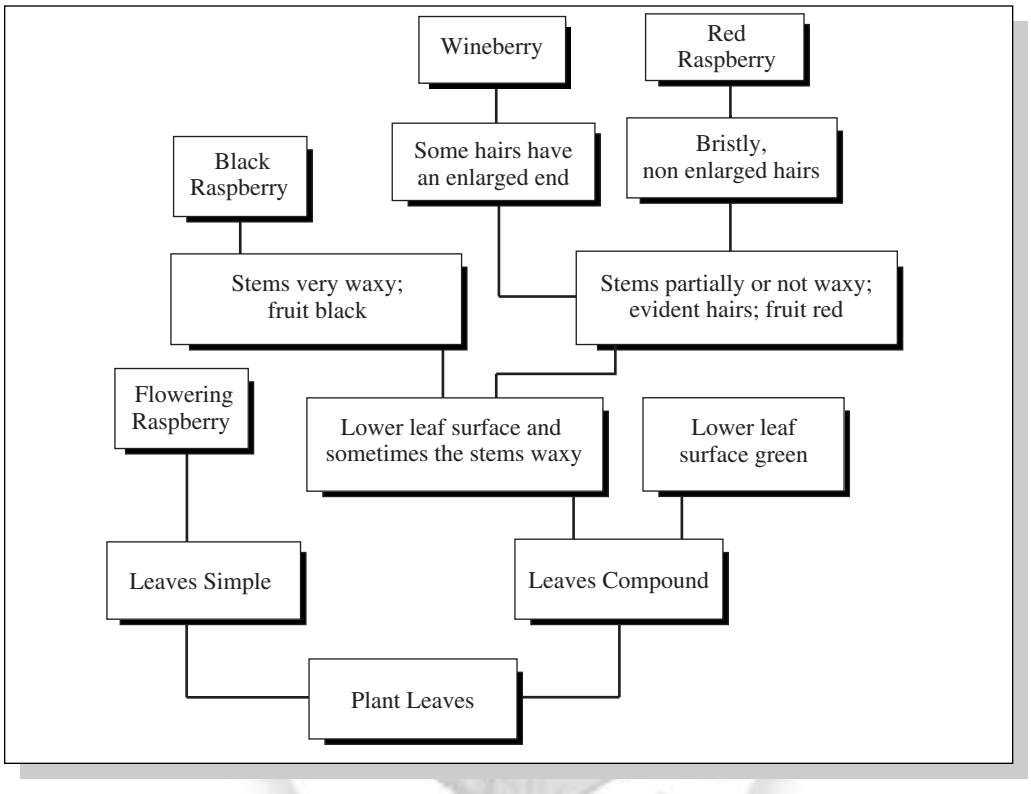


A small portion of a decision tree to classify raspberries is shown in Figure 3.4. Unlike Computer Science trees, classification trees may be drawn with the root down. Not shown is the root which has a branch to the "Leaves Simple" node and another branch to the "Leaves Compound" node. The decision process starts at the bottom by identifying gross features, such as whether the leaves are simple or compound. More specific details requiring closer observation are used as we travel up the tree. That is, larger sets of alternatives are examined first and then the decision process starts narrowing down the possibilities to smaller sets. This is a good way of organizing the decisions in terms of the time and effort to carry out more detailed observations.

If the decisions are binary, then a binary decision tree is easy to construct and is very efficient. Every question goes down one level in the tree. One question can decide one of two possible answers. Two questions can decide one of four possible answers. Three questions can decide one of eight possible answers and so on. If a binary tree is constructed such that all the leaves are answers and all the nodes leading down are questions, there can be a maximum of 2^N answers for N questions. For example, ten questions can classify one of 1,024 animals while twenty questions can classify one of 1,048,576 possible answers.

Another useful feature of decision trees is that they can be made **self-learning**. If the guess is wrong, a procedure can be called to query the user for a new, correct classification question and the answers to the "yes" and "no" responses. A new node, branches, and leaves can then be dynamically created and added to the tree. In the original Animals program written in BASIC, knowledge was stored in DATA statements. When the user taught the program a new animal, automatic

Figure 3.4 Portion of a Decision Tree for Species of Raspberries



learning took place as the program generated new DATA statements containing information about the new animal. For maximum efficiency, the animal knowledge is stored in trees. Using CLIPS, new rules can be built automatically as the program learns new knowledge using the **build** keyword (Giarratano 93). This **automated knowledge acquisition** is very useful since it can circumvent the knowledge acquisition bottleneck described in Chapter 6 in simple cases.

Decision structures can be mechanically translated into production rules. This can easily be done by a breadth-first search of the tree and generating IF-THEN rules at every node. For example, the decision tree of Figure 3.3 could be translated into rules as follows:

```
IF QUESTION = "IS IT VERY BIG?" AND RESPONSE = "NO"
      THEN  QUESTION := "DOES IT SQUEAK?"

IF QUESTION = "IS IT VERY BIG?" AND RESPONSE = "YES"
      THEN  QUESTION := "DOES IT HAVE A LONG NECK?"
```

and so forth for the other nodes. A leaf node would generate an ANSWER response rather than a question. Appropriate procedures would also query the user for input and construct new nodes if wrong.

Although decision structures are very powerful classification tools, they are limited because they cannot deal with variables as an expert system can. Expert systems are general-purpose tools rather than simply classifiers.

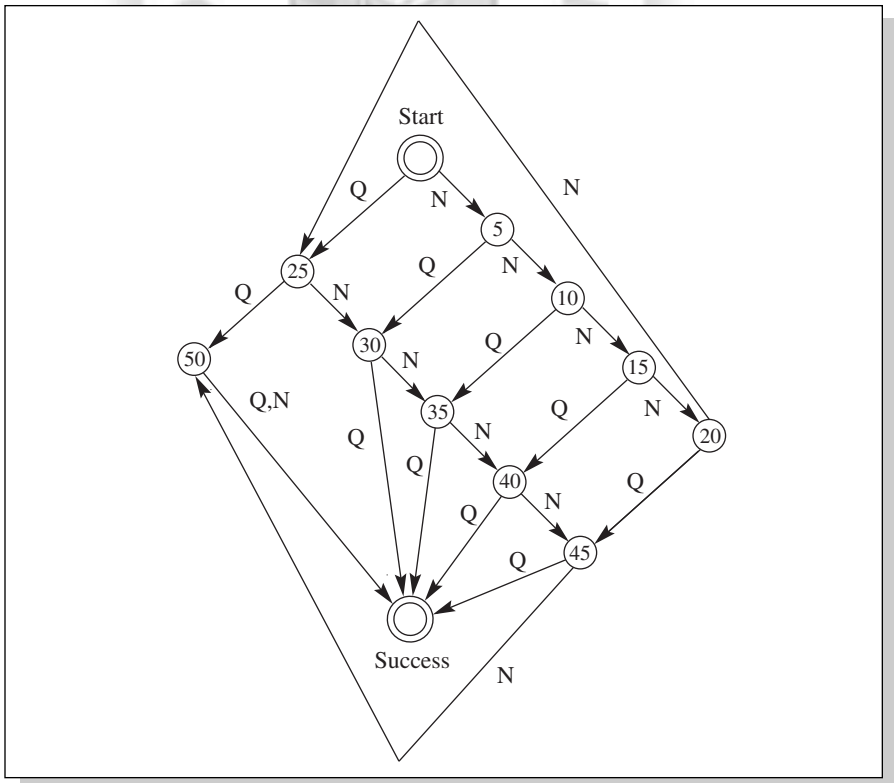
3.3 STATE AND PROBLEM SPACES

Graphs can be applied to many practical problems. A useful method of describing the behavior of an object is to define a graph called the **state space**. A state is a collection of characteristics that can be used to define the status or **state** of an object. The state space is the set of states showing the **transitions** between states that the object can experience. A transition takes an object from one state to another.

Examples of State Spaces

As a simple example of state spaces, consider the purchase of a soft drink from a machine. As you put coins into the machine, it makes a transition from one state to another. Figure 3.5 illustrates the state space assuming that only quarters and nickels are available and 55¢ is required for a drink. Adding other coins such as dimes and fifty-cent pieces makes the diagram more complicated and is not shown here.

Figure 3.5 State Diagram for a Soft Drink Vending Machine Accepting Quarters (Q) and Nickels (N)



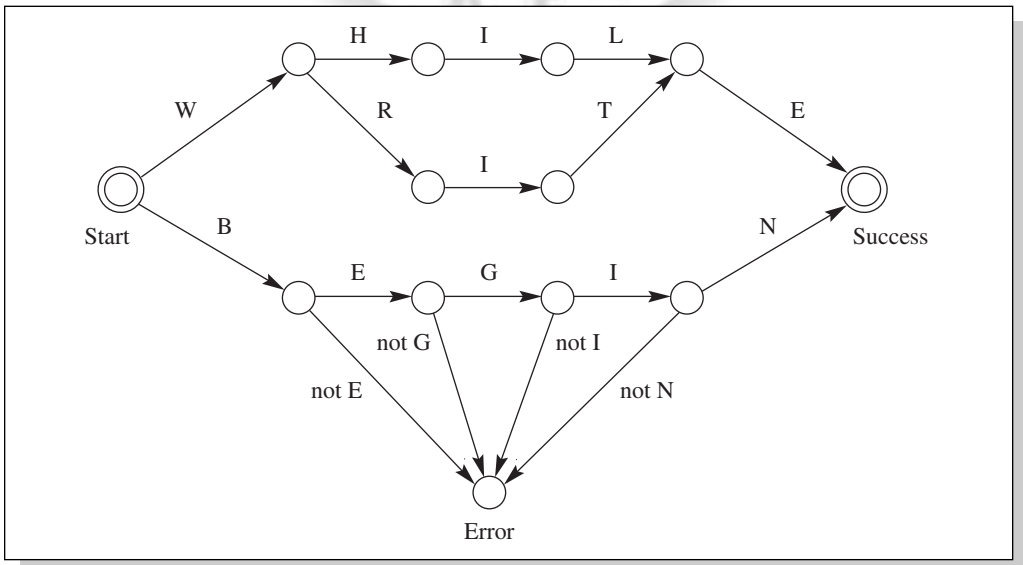
The start and success states are drawn as double circles to make them easier to identify. The states are shown as circles and the possible transitions to other states are drawn as arrows. Notice that this diagram is a weighted digraph, where the weights are the possible coins that can be input to the machine in every state.

This diagram is also called a **finite state machine diagram** because it describes the finite number of states of a machine. The term *machine* is used in a very general sense. The machine can be a real object, an algorithm, a concept, and so forth. Associated with every state are the actions that drive it to another state. At any time, this machine can be in only one state. As the machine accepts input to a state, it progresses from that state to another. If the correct inputs are given, the machine will progress from the start to the success or final state. If a state is not designed to accept a certain input, the machine will become hung up in that state. For example, the soft drink machine has no provision to accept dimes. If someone puts a dime in the machine, the response is undefined. A good design will include the possibility of invalid inputs from every state and provide for transitions to an error state. The error state is designed to give appropriate error messages and take whatever action is necessary.

Finite state machines are often used in compilers and other programs to determine the validity of an input. For example, Figure 3.6 shows part of a finite state machine to test input strings for validity. Characters of the input are examined one at a time. Only the character strings WHILE, WRITE, and BEGIN will be accepted. Arrows are shown from the BEGIN state for successful input and also for erroneous input going to the error state. For efficiency, some states, such as the one pointed to by "L" and "T," are used for testing both WHILE and WRITE.

NOTE: Only some of the error state transitions are shown.

Figure 3.6 Part of a Finite State Machine for Determining Valid Strings WHILE, WRITE, and BEGIN.



State diagrams are also useful in describing solutions to problems. In these kinds of applications we can think of the state space as a **problem space**, where some states correspond to intermediate stages in problem solving and some states correspond to answers. In a problem space there may be multiple success states corresponding to possible solutions. Finding the solution to a problem in a problem space involves finding a valid path from start (problem statement) to success (answer). The animal decision tree can be viewed as a problem space where the yes/no responses to questions determine the state transition.

Another example of a problem space occurs in the classic Monkey and Bananas problem shown in Figure 3.7. The problem is to give instructions to a monkey telling how to retrieve some bananas hanging from the ceiling. The bananas are out of reach. Inside the room are a couch and a ladder. The initial starting configuration is typically with the monkey on the couch. Instructions might be:

```
jump off couch
move to ladder
move ladder under bananas' position
climb ladder
grab bananas
```

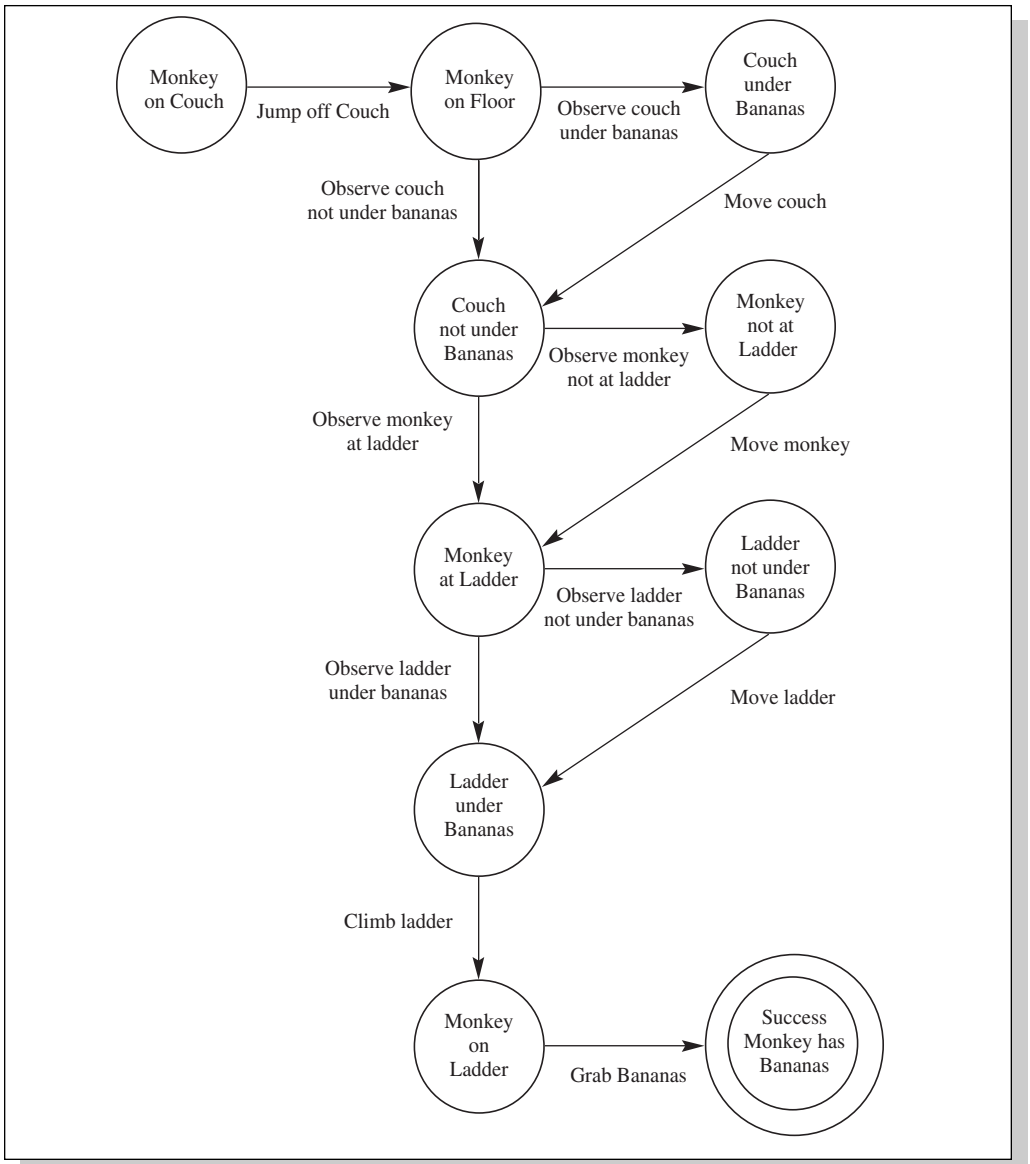
The instructions will vary depending on the initial configuration of monkey, couch, and ladder. Since there are a number of initial start states, the special double circles for the start are not shown. For example, another possible starting state is with the monkey on the couch under the bananas. The monkey will then have to push the couch out of the way before moving the ladder under the bananas. In the simplest starting state, the monkey is already on the ladder under the bananas.

Although this problem seems obvious to a human, it involves a considerable amount of reasoning. A practical application of a reasoning system like this is giving instructions to a robot concerning the solution of a task. Rather than assuming that all objects in the environment are fixed in place, a general solution is a reasoning system that can deal with a variety of situations. A rule-based solution to the Monkey and Bananas problem is distributed with the CLIPS CDROM.

Another useful application of graphs is exploring paths to find a problem's solution. Figure 3.8a shows a simple net for the Traveling Salesman problem. In this example, assume the problem is to find a complete path from node A that visits all other nodes. As usual in the Traveling Salesman problem, no node can be visited twice. Figure 3.8b shows all the possible paths starting from node A in the form of a tree. The correct paths ABDCA and ACDBA are shown as thick lines in this graph.

Depending on the search algorithm, the exploration of paths to find the correct one may involve a considerable amount of backtracking. For example, the path ABA may be first searched unsuccessfully and then backtracked to B. From B, the paths CA, CB, CDB, and CDC will be unsuccessfully searched. Next the path BDB will be unsuccessfully searched until the first correct path ABDCA is found.

Figure 3.7 The State Space for the Monkey and Bananas Problem

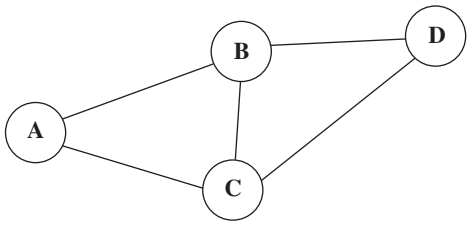


Ill-Structured Problem Spaces

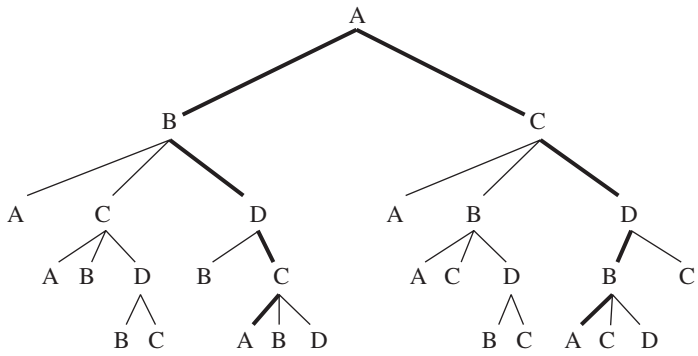
A useful application of state spaces is in characterizing ill-structured problems. In Chapter 1 an ill-structured problem was defined to have uncertainties associated with it. These uncertainties can be specified more precisely by a problem space.

As an example of an ill-structured problem, let's consider again the case of a person who is thinking about traveling and an online search doesn't help, so

Figure 3.8 A Traveling Salesman Problem



(a) Graph of a Traveling Salesman Problem



(b) Search Paths (optimal path shown by thick edges)

visits a travel agent as discussed in Chapter 1. Table 3.1 lists some characteristics of this ill-structured problem as a problem space, indicated by the person's responses to the travel agent's questions.

If you compare Table 3.1 to Table 1.10 in Chapter 1, you'll see that the concept of a problem space lets us specify more precisely the characteristics of an ill-structured problem. It is essential to characterize these parameters precisely to determine if a solution is feasible, and if so, what is needed for a solution. A problem is not necessarily ill-structured just because it has one, some, or even all of these characteristics since much depends on the severity. For example, all theorem-proving problems have an infinite number of potential solutions, but this does not make theorem proving an ill-structured problem.

As you can see from Table 3.1, there are many uncertainties and yet travel agents cope with them every day. While not all cases may be as bad as this, it indicates why an algorithmic solution would be very difficult.

A well-formed problem is one in which we know the explicit problem, goal, and operators that can be applied to go from one state to another. A well-formed problem is **deterministic** because when an operator is applied to a state, we are sure of the next state. The problem space is bounded and the states are discrete. This means that there are a finite number of states and each state is well defined.

Table 3.1 Example of an Ill-Structured Problem for Travel

Characteristic	Response
Goal not explicit	I'm thinking about going somewhere
Problem space unbounded	I'm not sure where to go
Problem states not discrete	I just like to travel; the destination is not important
Intermediate states difficult to achieve	I don't have enough money to go
State operators unknown	I don't know how to get the money
Time constraint	I must go soon

In the travel problem, the states are unbounded because there are infinitely many possible destinations that a traveler might go to. An analogous situation occurs with an analog meter, which may indicate an infinite number of possible readings. If we consider each reading of the meter to be a state, then there are an infinite number of states and they are not well defined because they correspond to real numbers. Since there are an infinite number of real numbers between any two real numbers, the states are not discrete because the next state differs only infinitesimally. In contrast, the readings of a digital meter are bounded and discrete.

3.4 AND-OR TREES AND GOALS

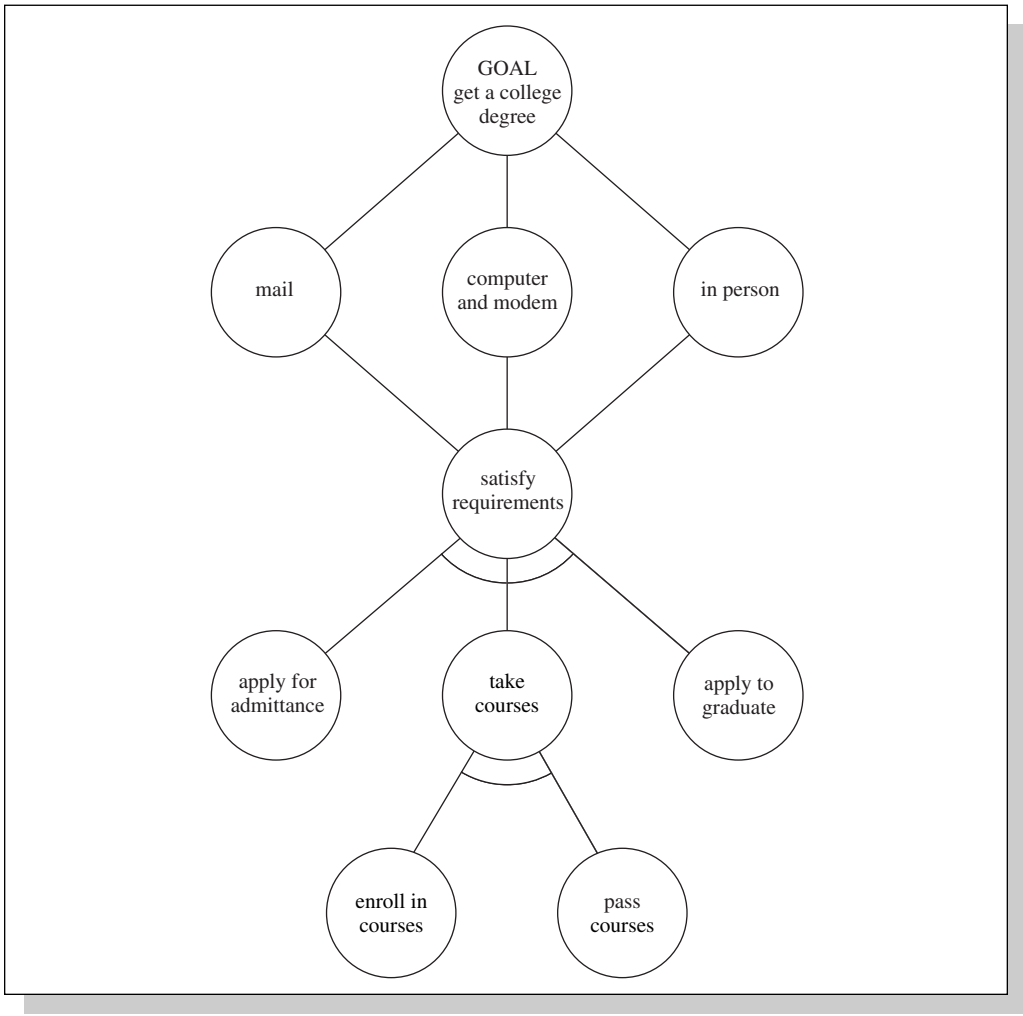
Many types of expert systems use backward chaining to find solutions to problems. PROLOG is a good example of a backward-chaining system that tries to solve a problem by breaking it up into smaller subproblems and solving them. During the 1990s PROLOG started to become widely used for commercial applications in business and industry (http://www.ddj.com/documents/s=9064/ddj0212ai004/0212aie004.htm). Solving a problem is considered by optimists as a goal to be achieved. In order to accomplish a significant goal, many subgoals may need to be accomplished.

One type of tree or lattice that is useful in representing backward-chaining problems is the AND-OR tree. Figure 3.9 shows a simple example of an AND-OR lattice to solve the goal of obtaining a college degree. To accomplish this goal, you can either attend a college in person or through correspondence courses. With correspondence courses, work can be performed either by mailing assignments or electronically using a home computer and modem.

In order to satisfy requirements for the degree, three subgoals must be accomplished: (1) apply for admittance, (2) take courses, and (3) apply to graduate. Notice that there is an arc through the edges from the Satisfy Requirements goal to these three subgoals. The arc through the edges indicates that Satisfy Requirements is an AND-node that can be satisfied only if all three of its subgoals are satisfied. Goals without the arc such as Mail, Computer and Modem, and In Person are OR-nodes in which accomplishing any of these subgoals satisfies its parent goal of Get a College Degree.

This diagram is a lattice because the Satisfy Requirements subgoal has three parent nodes: (1) Mail, (2) Computer and Modem, and (3) In Person. Notice that it would be possible to draw this diagram as a tree by simply duplicating the

Figure 3.9 AND-OR Lattice Showing How to Obtain a College Degree

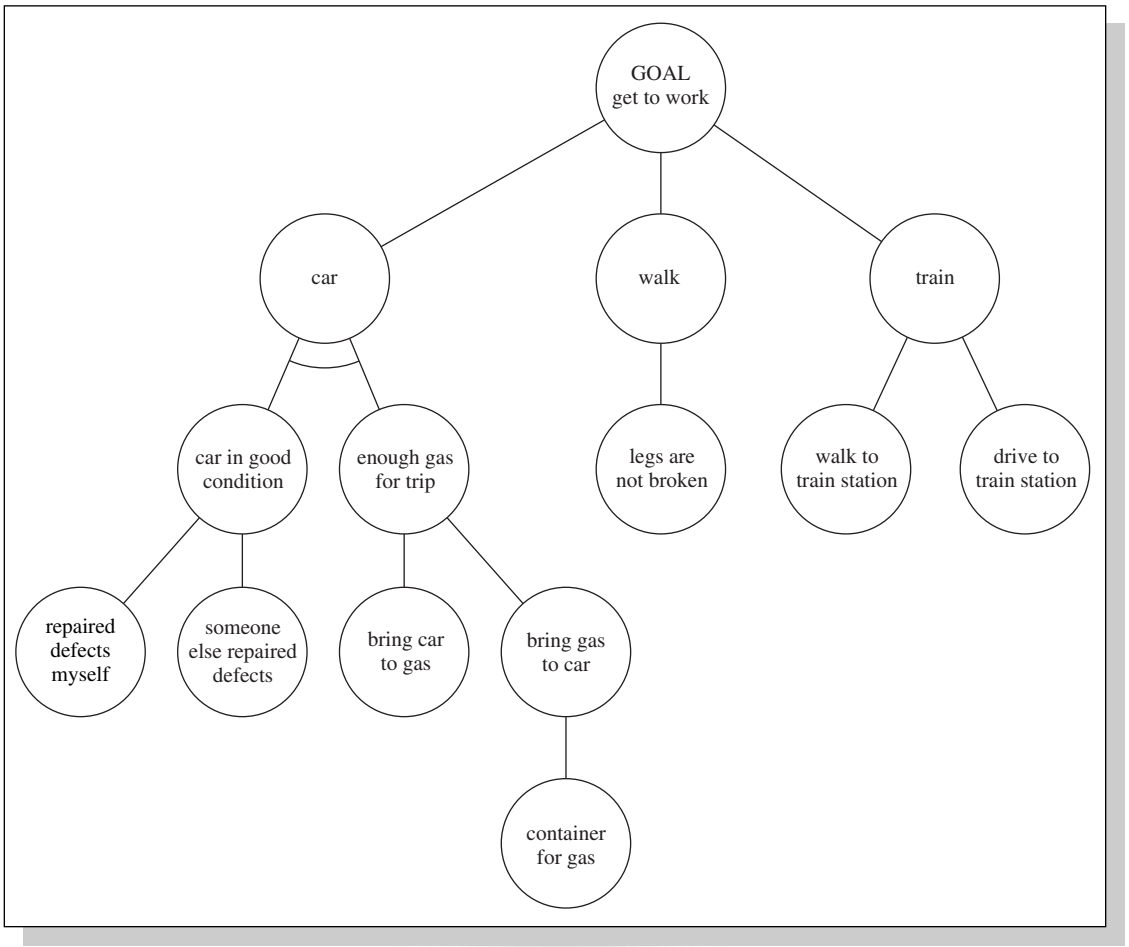


subgoal Satisfy Requirements and its subtree of goals for the Mail, Computer and Modem, and In Person goals. However, since the Satisfy Requirements is the same for each of its parents there is no real advantage, and it uses more paper to draw the tree.

As another simple example, Figure 3.10 shows an AND-OR tree for the problem of getting to work by different possible ways. For completeness, this could also be converted into a lattice. For example, an edge could be added from the node Drive to Train Station to the Car node and from Walk to Train Station to the Walk node. Figure 3.11 shows an AND-exclusive OR type lattice.

Another way of describing problem solutions is an AND-OR-NOT lattice, which uses logic gate symbols instead of the AND-OR tree type notation. The logic gate symbols for AND, OR, and NOT are shown in Figure 3.12. These

Figure 3.10 A Simple AND-OR Tree Showing Methods of Getting to Work



gates implement the truth tables for AND, OR, and NOT discussed in Chapter 2. Figure 3.13 shows Figure 3.9 implemented with AND and OR gates.

AND-OR trees and decision trees have the same basic advantages and disadvantages. The main advantage of AND-OR-NOT lattices is their potential implementation in hardware for fast processing speeds. These lattices can be custom designed for fabrication as integrated circuits. In practice, one type of logic gate such as the NOT-AND or NAND is used for reasons of manufacturing economy rather than separate AND, OR, and NOT gates. From logic it can be proved that any logic function can be implemented by a NAND gate. An integrated circuit with one type of device is cheaper to manufacture than one with multiple types of logic gates.

A chip using forward chaining can compute the answer very quickly as a function of its inputs since processing proceeds in parallel. Chips like this can be used for real-time monitoring of sensor data and make an appropriate response

Figure 3.11 AND-OR Lattice for Car Selling/Repair Decision

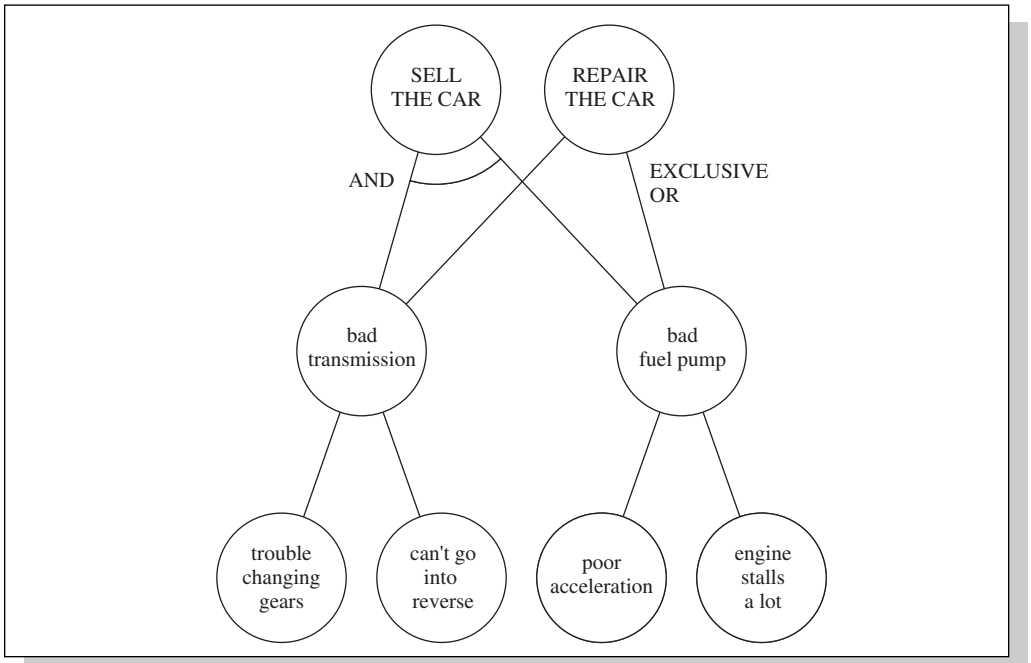


Figure 3.12 AND, OR, and NOT Logic Gate Symbols

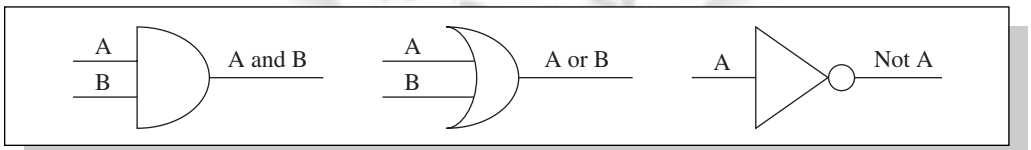
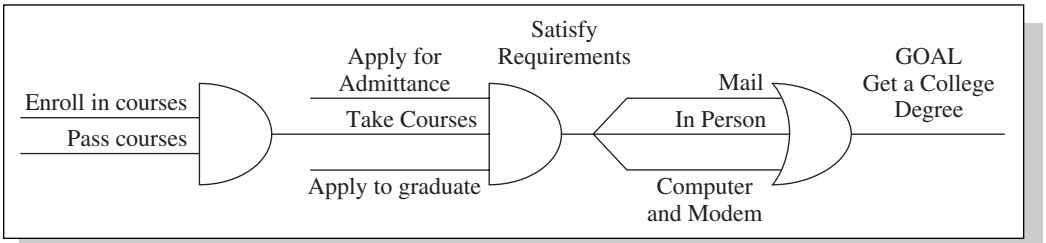


Figure 3.13 AND-OR Logic Gate Representation for Figure 3.9



depending on the inputs. The main disadvantage is that like other decision structures, a chip designed for logic cannot handle situations it was not designed for. However, an ANS implemented on a chip can handle unexpected inputs.

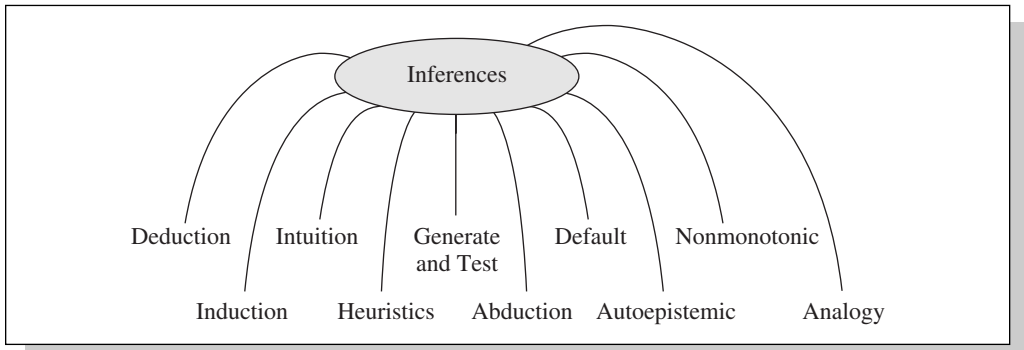
3.5 DEDUCTIVE LOGIC AND SYLLOGISMS

In Chapter 2, we discussed the representation of knowledge by logic. Now you will see how inferences are made to derive new knowledge or information. In the remainder of this chapter we will discuss different methods of inference. Figure 3.14 is an overview of the methods of inference. A brief summary is as follows:

- *Deduction*. Logical reasoning in which conclusions must follow from their premises.
- *Induction*. Inference from the specific case to the general. Induction is one of the main methods of machine learning in which computers learn without human intervention. Three popular methods of machine learning are the connectionist Hopfield network, the symbolic ID3/ID5R, and evolution-based genetic algorithms (http://ai.bpa.arizona.edu/papers/mlir93/ mlir93.html).
- *Intuition*. No proven theory. The answer just appears, possibly by unconsciously recognizing an underlying pattern. Expert systems do not implement this type of inference yet. ANS may hold promise for this type of inference since they can extrapolate from their training rather than just provide a conditioned response or interpolation. That is, a neural net will always give its best guess for a solution.
- *Heuristics*. Rules of thumb based on experience.
- *Generate and test*. Trial and error. Often used with planning for efficiency.
- *Abduction*. Reasoning back from a true conclusion to the premises that may have caused the conclusion.
- *Default.* In the absence of specific knowledge, assume general or common knowledge by default.
- *Autoepistemic*. Self-knowledge, such as what color the sky appears to you.
- *Nonmonotonic.* Previous knowledge may be incorrect when new evidence is obtained.
- *Analogy*. Inferring a conclusion based on the similarities to another situation. Neural networks do this by recognizing patterns in the data and then extrapolating to a new situation.

Although not explicitly shown in Figure 3.14, **commonsense knowledge** is a combination of all of these types based on our experience. Commonsense reasoning is the type that people use in ordinary situations, and is very difficult for computers. The major attempt to build a huge database of commonsense knowledge suitable for computer use has been underway by Doug Lenat since the 1980s and practical applications are now being made (http://www.OpenCyc.org). This database consists of many assertions and rules that may be

Figure 3.14 Types of Inference



used for applications ranging from better speech understanding to more efficient construction of ontologies. The interesting thing is that the adage, "Bigger is better," does not always hold. In the following communication from Doug Lenat (personal email) he explains why:

> "3 million rules have been entered by hand; through generalization and factoring we have that number down to 1.5 million today. We try to make the number as small as possible, not as large—we could easily expand this into over a billion rules, e.g. Consider 2 types of animals—no rat is also a camel. Knowing 10,000 animals, one could write 100,000,000 rules of that sort. Instead, Cyc has 10,001 rules: Linnaean taxon relationships plus a rule that says that two taxons that aren't known to stand in a genl/spec relationship are disjoint. The "tens of thousands of rules" is talking about a very special type of rule, namely one of a certain level of complexity, through which NONE of the assertions I just mentioned would count as rules at all.
>
> We have a couple million assertions because that's as few as we can get away with for the content. We don't TRY to make the number large, because it would slow things down, but if we WANTED it to be large we could take 10,001 of the (2 million) assertions in the current knowledge base, and equivalently replace them with 100,000,000 only slightly less general-seeming rules. That would be bad."

The application of fuzzy logic to commonsense reasoning is discussed in Chapter 5.

One of the most often-used methods of drawing inferences is **deductive logic**, which has been used since ancient times to determine the validity of an **argument**. Although people commonly use the word argument to describe an angry "exchange of views," it has a very different meaning in logic. A logical

argument is a group of statements in which the last is claimed to be justified on the basis of the previous ones in the **chain of reasoning**. One type of logical argument is the syllogism, which was discussed in Chapter 2. As an example of a syllogism:

```
Premise:    Anyone who can program is intelligent
Premise:    John can program
Conclusion: Therefore, John is intelligent
```

In an argument the premises are used as evidence to support the conclusions. The premises are also called the **antecedent** and the conclusion is called the **consequent**. The essential characteristic of deductive logic is that the true conclusion *must* follow from true premises. A line is customarily drawn to separate the premises from the conclusion, as shown above, so that it is not necessary to explicitly label the premises and conclusion.

The argument could have been written more briefly as:

```
Anyone who can program is intelligent
John can program
∴ John is intelligent
```

where the three dots, ∴, mean "therefore."

Let's take a closer look at syllogistic logic now. The main advantage of studying syllogisms is that it is a simple, well-understood branch of logic that can be completely proven. Also, syllogisms are often useful since they can be expressed in terms of IF-THEN rules. For example, the previous syllogism can be rephrased as:

```
IF   Anyone who can program is intelligent and
     John can program
THEN John is intelligent
```

In general, a syllogism is any valid deductive argument having two premises and a conclusion. The classic syllogism is a special type called a **categorical syllogism**. The premises and conclusions are defined as categorical statements of the following four forms, as shown in Table 3.2.

Table 3.2 Categorical Statements

Form	Schema	Meaning
A	All S is P	universal affirmative
E	No S is P	universal negative
I	Some S is P	particular affirmative
O	Some S is not P	particular negative

Note that in logic, the term *schema* specifies the logical form of the statement. This also illustrates another use of the word schema, which is different from its AI use, discussed in Chapter 2. In logic, the word schema is used to show the essential form of an argument. Schemata may also specify the logical form of an entire syllogism as in:

```
All M is P
All S is M
∴ All S is P
```

The subject of the conclusion, S, is called the **minor term** while the predicate of the conclusion, P, is called the **major term**. The premise containing the major term is called the **major premise** and the premise containing the minor term is called the **minor premise**. For example:

```
Major Premise:   All M is P
Minor Premise:   All S is M
Conclusion: All S is P
```

is a syllogism said to be in **standard form**, with its major and minor premises identified. The **subject** is the object that is being described while the **predicate** describes some property of the subject. For example, in the statement:

```
All microcomputers are computers
```

the subject is "microcomputers" and the predicate is "computers." In the statement:

```
All computers with 1 Gigabyte of RAM
are computers with a lot of memory
```

the subject is "computers with 1 Gigabyte" and the predicate is "computers with a lot of memory."

The forms of the categorical statements have been identified since ancient times by the letters A, E, I, and O. The A and I indicate affirmative and are thought to come from the first two vowels of the Latin word *affirmo* (I affirm); while E and O come from *nego* (I negate). The A and I forms are said to be **affirmative in quality** by affirming that the subjects are included in the predicate class. The E and O forms are **negative in quality** because the subjects are excluded from the predicate class.

The verb *is* comes from **copula** in Latin, which means to connect. The copula connects the two parts of the statement. In the standard categorical syllogism, the copula is the present tense form of the verb *to be*. So another version is:

```
All S are P
```

The third term of the syllogism, M, is called the **middle term** and is common to both premises. The middle term is essential because a syllogism is de-

fined such that the conclusion cannot be inferred from either of the premises alone. So the argument:

```
All A is B
All B is C
∴ All A is B
```

is not a valid syllogism since it follows from the first premise alone.

The **quantity** or **quantifier** describes the portion of the class included. The quantifiers *All* and *No* are **universal** because they refer to entire classes. The quantifier *Some* is called **particular** because it refers to just part of the class.

The **mood** of a syllogism is defined by the three letters that give the form of the major premise, minor premise, and conclusion, respectively. For example, the syllogism:

```
All M is P
All S is M
∴ All S is P
```

is an AAA mood.

There are four possible patterns of arranging the S, P, and M terms, as shown in Table 3.3. Each pattern is called a figure, with the number of the **figure** specifying its type.

Table 3.3 Patterns of Categorical Statements

	Figure 1	Figure 2	Figure 3	Figure 4
Major Premise	M P	P M	M P	P M
Minor Premise	S M	S M	M S	M S

So the previous example is completely described as an AAA-1 syllogism type. Just because an argument has a syllogistic form does not mean that it is a valid syllogism. Consider the AEE-1 syllogism form:

```
All M is P
No S is M
∴ No S is P
```

which is not a valid syllogism, as can be seen from the example:

```
All microcomputers are computers
No mainframe is a microcomputer
∴ No mainframe is a computer
```

Rather than trying to think up examples to prove the validity of syllogistic arguments, there is a **decision procedure** that can be used. A decision procedure

is a method of proving validity. A decision procedure is some general mechanical method or algorithm by which the process of determining validity can be automated. While there are decision procedures for syllogistic logic and propositional logic, Church showed in 1936 that there is none for predicate logic. Instead, people or computers must apply creativity to generate proofs. In the 1970s, programs such as the Automated Mathematician and Eurisko by Doug Lenat rediscovered mathematical proofs of the Goldbach's Conjecture and the Unique Factorization Theorem. Mathematics journals have not been keen on publishing creative works by computers in the same way literary journals and magazines have not exactly encouraged the submission of poems or novels by computers. (They're only willing to pay a little bit.)

The decision procedure for propositions is simply constructing a truth table and examining it for tautology. The decision procedure for syllogisms can be done using Venn diagrams with three overlapping circles representing S, P, and M, as shown in Figure 3.15a. For the syllogism form AEE-1:

All M is P
No S is M
∴ No S is P

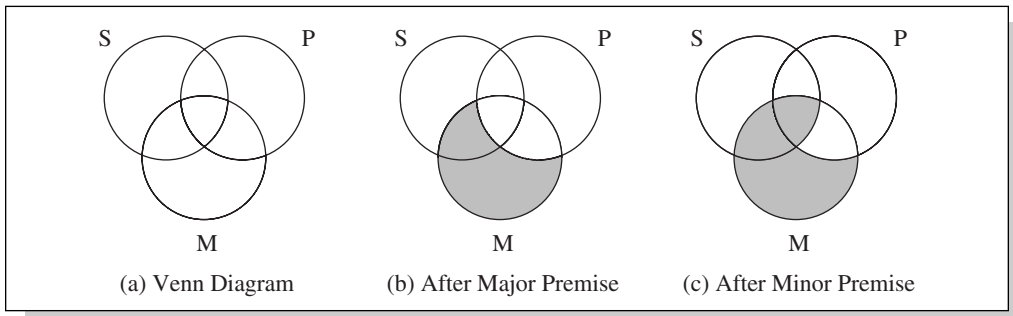
the major premise is illustrated in Figure 3.15b. The lined section of M indicates that there are no elements in that portion. In Figure 3.15c the minor premise is included by lining its portion with no elements. From Figure 3.15c it can be seen that the conclusion of AEE-1 is false since there are some S in P.

As another example, the following EAE-1 is valid as can be seen from Figure 3.16c:

No M is P
All S is M
∴ No S is P

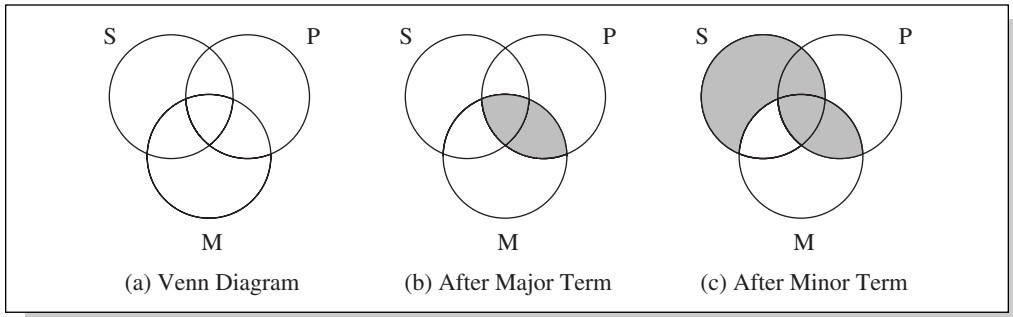
Venn diagrams that involve "some" quantifiers are a little more difficult to draw. The general rules for drawing categorical syllogisms under the Boolean view that there may be no members in the A and E statements are:

Figure 3.15 Decision Procedure for Syllogism AEE-1



(a) Venn Diagram (b) After Major Premise (c) After Minor Premise

Figure 3.16 Decision Procedure for Syllogism EAE-1



1. If a class is empty, it is shaded.
2. Universal statements, A and E, are always drawn before particular ones.
3. If a class has at least one member, mark it with an *.
4. If a statement does not specify in which of two adjacent classes an object exists, place an * on the line between the classes.
5. If an area has been shaded, no * can be put in it.

As an example,

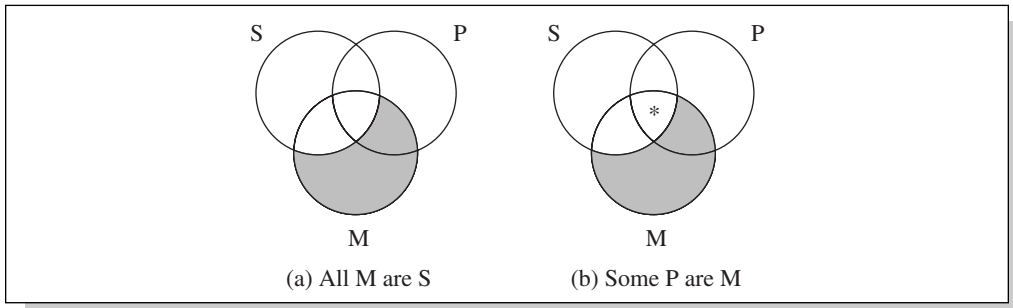
```
Some computers are laptops
All laptops are transportable
∴ Some transportables are computers
```

which can be put in IAI-4 type:

```
Some P are M
All M are S
∴ Some S are P
```

Following the rules 2 and 1 for Venn diagrams, we start with the universal statement for the minor premise and shade it, as shown in Figure 3.17a. Next, rule 3 is applied to the particular major premise and a * is drawn, as shown in Figure 3.17b. Since the conclusion "Some transportables are computers" is shown in the diagram, it follows that the argument IAI-4 is a valid syllogism.

Figure 3.17 Syllogism of IAI-4 Type



3.6 RULES OF INFERENCE

Although Venn diagrams are a decision procedure for syllogisms, the diagrams are inconvenient for more complex arguments because they become more difficult to read. However, there is a more fundamental problem with syllogisms because they address only a small portion of the possible logical statements. In particular, categorical syllogisms only address categorical statements of the A, E, I, and O form.

Propositional logic offers another means of describing arguments. In fact, we often use propositional logic without realizing it. For example, consider the following propositional argument:

```
If there is power, the computer will work
There is power
∴ The computer will work
```

This argument can be expressed in a formal way by using letters to represent the propositions as follows:

```
A = There is power
B = The computer will work
```

and so the argument can be written as:

```
A → B
A
∴ B
```

Arguments like this occur often. A general schema for representing arguments of this type is:

```
p → q
p
∴ q
```

where p and q are logical variables that can represent any statements. The use of logical variables in propositional logic allows more complex types of statements than the four syllogistic forms A, E, I, and O. Inference schema of this propositional form is called by a variety of names: **direct reasoning**, **modus ponens**, **law of detachment,** and **assuming the antecedent** (Lakoff 00).

Notice that this example can also be expressed in the syllogistic form:

```
All computers with power will work
This computer has power
∴ This computer will work
```

which demonstrates that *modus ponens* is really a special case of syllogistic logic. Modus ponens is important because it forms the basis of rule-based expert systems.

However, rule-based systems do not rely exclusively on logic because people use more than logic to solve problems. In the real world there may be several competing rules, not just the single rules of syllogisms. The expert system rule-engine must decide which is the appropriate rule to execute just as a person must decide, "Should I eat that last piece of candy and satisfy my craving or not eat it and stay slim?"

It is not enough to just write a bunch of rules in C and have the power of an expert system tool, although this may be all that is required for very simple applications (http://www.ddj.com/documents/s=9064/ddj0301aie001/0301aie001.htm). The real power of an expert systems tool comes into play when there are hundreds or thousands of rules and they conflict. The Rete pattern-matching algorithm is one of the most powerful and yet efficient methods of resolving rule conflicts and is the basis of CLIPS pattern matching (http://www.ddj.com/documents/s=9064/ddj0212ai002/0212aie002.htm).

The compound proposition $p \rightarrow q$ corresponds to the rule while the p corresponds to the pattern that must match the antecedent for the rule to be satisfied. However, as discussed in Chapter 2, the conditional $p \rightarrow q$ is not exactly equivalent to a rule because the conditional is a logical definition defined by a truth table and there are many possible definitions of the conditional.

Generally we will follow the convention of logic theory of using uppercase letters such as A, B, C . . . to represent constant propositions such as "There is power." Small letters such as p, q, r . . . will represent logical variables, which can stand for different constant propositions. Note that this convention is opposite to that of PROLOG, which uses uppercase letters for variables.

This modus ponens schema could also have been written with differently named logical variables as:

$$\begin{array}{l} r \rightarrow s \\ \underline{r\quad\quad} \\ \therefore\ s \end{array}$$

and the schema would still mean the same.

Another notation for this schema is:

$$r,\ r \rightarrow s;\ \therefore\ s$$

where the comma is used to separate one premise from another and the semicolon indicates the end of the premises. Although so far we have looked at arguments with only two premises, a more general form of an argument is:

$$P_1,\ P_2,\ \ldots\ P_N;\ \therefore\ C$$

where the uppercase letters P_i represent premises such as r, $r \rightarrow s$, and C is the conclusion. Notice how this resembles the goal satisfaction statement of PROLOG discussed in Chapter 2.:

```
p :- p1, p2, ... pN.
```

The goal, p, is satisfied if all the subgoals p_1, p_2, . . . p_N are satisfied. An analogous argument for production rules can be written in the general form:

$$C_1 \wedge C_2 \wedge \ldots C_N \rightarrow A$$

which means that if each condition, C_i, of a rule is satisfied, then the action, A, of the rule is done. As discussed previously, a logical statement of the form above is strictly not equivalent to a rule because the logical definition of the conditional is not the same as a production rule. However, this logical form is a useful intuitive aid in thinking about rules.

The notation of the logic operators AND and OR have different forms in PROLOG compared to the usual $\wedge$ and $\vee$. The comma between subgoals in PROLOG means a conjunction, $\wedge$, while the disjunction, $\vee$, is indicated with a semicolon. For example,

```
p :- p1; p2.
```

means that p is satisfied if p_1 or p_2 is satisfied. Conjunctions and disjunctions can be mixed. For example,

```
p :- p1, p2; p3, p4.
```

is the same as the two PROLOG statements:

```
p :- p1, p2.
p :- p3, p4.
```

Although PROLOG has a powerful built-in reasoning strategy, rules created this way are not always suitable for custom use. However it is possible to customize the rule execution using PROLOG so that the knowledge representation and inference strategy can be tuned to the application, such as one that automates pediatric offices (www.visualdatallc.com) and is described in more detail in (Merritt 04).

In general, if the premises and conclusion are all schemata, the argument:

$$P_1, P_2, \ldots P_N; \therefore C$$

is a formally valid deductive argument if and only if:

$$P_1 \wedge P_2 \wedge \ldots P_N \rightarrow C$$

is a tautology. As an example,

$$(p \wedge q) \rightarrow p$$

is a tautology because it is true for any values, T or F, of p and q. You can verify this by constructing the truth table.

The argument of modus ponens,

$$\begin{array}{l} p \rightarrow q \\ \underline{p\quad\quad} \\ \therefore\ q \end{array}$$

is valid because it can be expressed as a tautology:

$$(p \rightarrow q) \wedge p \rightarrow q$$

Note that we are assuming that the arrow has lower precedence than conjunction and disjunction. This saves writing additional parentheses such as:

$$((p \rightarrow q) \wedge p) \rightarrow q$$

The truth table for modus ponens is shown in Table 3.4. It is a tautology because the values of the argument, shown in the rightmost column, are all true no matter what the values of its premises. Notice that in the third, fourth, and fifth columns, the truth values are written under certain operators such as the $\rightarrow$ and $\wedge$. These are called the main connectives because they connect the two main parts of a compound proposition.

Table 3.4 Truth Table for Modus Ponens

p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge p$	$(p \rightarrow q) \wedge p \rightarrow q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Although this method of determining valid arguments works, it does require checking every row of the truth table. The number of rows is 2^N, where N is the number of premises and so the rows increase very rapidly with the number of premises. For example, five premises would require 32 rows to be checked while ten premises require 1,024 rows. A shorter method of determining a valid argument is to consider only those rows of the truth table in which the premises are all true. The equivalent definition of a valid argument states that it is valid if and only if the conclusion is true for each of these rows. That is, the conclusion is tautologically implied by the premises. For modus ponens, the $p \rightarrow q$ premise and p premise are both true only in the first row, and so is the conclusion. Hence, modus ponens is a valid argument. If there were any other row in which the premises were all true and the conclusion false, then the argument would be invalid.

The shorter way of expressing the truth table for modus ponens is shown in Table 3.5, where all the rows are explicitly shown. In practice, only those rows that have true premises, such as the first row, need be considered.

Table 3.5 Alternate Short-Form Truth Table for Modus Ponens

		Premises		Conclusion
p	q	$p \rightarrow q$	p	q
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	F	F

The truth table for modus ponens shows that it is valid because the first row has true premises and a true conclusion, and there are no other rows that have true premises and a false conclusion.

Arguments can be deceptive. To show this, first consider the following valid example of modus ponens:

```
If there are no bugs, then the program compiles
There are no bugs
∴ The program compiles
```

Compare this with the following argument that somewhat resembles modus ponens:

```
If there are no bugs, then the program compiles
The program compiles
∴ There are no bugs
```

Is this a valid argument? The schema for arguments of this type is:

$$\begin{array}{l} p \rightarrow q \\ \underline{q} \\ \therefore\ p \end{array}$$

and its short-form truth table is shown in Table 3.6.

Table 3.6 Short-Form Truth Table of $p \rightarrow q$, q; $\therefore p$

		Premises		Conclusion
p	**q**	**p → q**	**q**	**p**
T	T	T	T	T
T	F	F	F	T
F	T	T	T	F
F	F	T	F	F

Notice that this argument is not valid. Although the first row does show that the conclusion is true if all the premises are true, the third row shows that if the premises are true, the conclusion is false. Thus this argument fails the if and only if criteria of a valid argument. Although many programmers wish arguments like this were true, logic (and experience) proves it a fallacy or invalid argument. This particular fallacious argument is called the **fallacy of the converse.** The converse is defined in Table 3.9.

As another example, the argument schema:

$$\begin{array}{l} p \rightarrow q \\ \underline{\sim q} \\ \therefore\ \sim p \end{array}$$

is valid since Table 3.7 shows the conclusion is true only when the premises are true.

Table 3.7 Short-Form Truth Table of $p \rightarrow q$, $\sim q$; $\therefore \sim p$

		Premises		Conclusion
p	**q**	**p → q**	**~q**	**~p**
T	T	T	F	F
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

This particular schema is called by a variety of names: **indirect reasoning**, **modus tollens**, and **law of contraposition**.

Modus ponens and modus tollens are rules of inference, sometimes called **laws of inference**. Table 3.8 shows some of the laws of inference.

The Latin name *modus* means "way," while *ponere* means "assert," and *tollere* means to "deny." The real names of the modus rules and their literal meanings are shown in Table 3.9. Modus ponens and modus tollens are short for

Table 3.8 Some Rules of Inference for Propositional Logic

Law of Inference	Schemata	
1. Law of Detachment	$p \rightarrow q$ p $\therefore q$	
2. Law of the Contrapositive	$p \rightarrow q$ $\therefore \sim q \rightarrow \sim p$	
3. Law of Modus Tollens	$p \rightarrow q$ $\sim q$ $\therefore \sim p$	
4. Chain Rule (Law of the Syllogism)	$p \rightarrow q$ $q \rightarrow r$ $\therefore p \rightarrow r$	
5. Law of Disjunctive Inference	$p \vee q$ $\sim p$ $\therefore q$	$p \vee q$ $\sim q$ $\therefore p$
6. Law of the Double Negation	$\sim(\sim p)$ $\therefore p$	
7. De Morgan's Law	$\sim(p \wedge q)$ $\therefore \sim p \vee \sim q$	$\sim(p \vee q)$ $\therefore \sim p \wedge \sim q$
8. Law of Simplification	$p \wedge q$ $\therefore p$	$\sim(p \vee q)$ $\therefore q$
9. Law of Conjunction	p q $\therefore p \wedge q$	
10. Law of Disjunctive Addition	p $\therefore p \vee q$	
11. Law of Conjunctive Argument	$\sim(p \wedge q)$ p $\therefore \sim q$	$\sim(p \wedge q)$ q $\therefore \sim p$

Table 3.9 The Modus Meanings

Rule of Inference Number	Name	Meaning "mood which by..."
1	modus ponendo ponens	affirming, affirms
3	modus tollendo tollens	denying, denies
5	modus tollendo ponens	denying, affirms
11	modus ponendo tollens	affirming, denies

the first two types (Stebbing 50). The Rule of Inference numbers correspond to those in Table 3.8.

The rules of inference can be applied to arguments with more than two premises. For example, consider the following argument:

```
Chip prices rise only if the yen rises.
The yen rises only if the dollar falls and
   if the dollar falls then the yen rises.
Since chip prices have risen,
   the dollar must have fallen.
```

Let the propositions be defined as follows:

```
C = chip prices rise
Y = yen rises
D = dollar falls
```

Recall from Section 2.12 that one of the meanings of the conditional is "p, only if q." A proposition such as "The yen rises only if the dollar falls" has this meaning and so is represented as $C \rightarrow Y$. The entire argument has the following form:

$$
\begin{array}{l}
C \rightarrow Y \\
(Y \rightarrow D) \wedge (D \rightarrow Y) \\
\underline{C\quad\quad} \\
\therefore D
\end{array}
$$

The second premise has an interesting form, which can be further reduced by using a variant of the conditional. The conditional $p \rightarrow q$ has several variants, which are the converse, **inverse**, and **contrapositive**. These are listed with the conditional, for completeness, in Table 3.10.

Table 3.10 The Conditional and Its Variants

conditional	$p \rightarrow q$
converse	$q \rightarrow p$
inverse	$\sim p \rightarrow \sim q$
contrapositive	$\sim q \rightarrow \sim p$

As usual, it is assumed that the negation operator has a higher priority than the other logical operators and so no parentheses are used around ~p and ~q.

If the conditional $p \rightarrow q$ and its converse $q \rightarrow p$ are both true, then p and q are equivalent. That is, $p \rightarrow q \wedge q \rightarrow p$ is equivalent to the **biconditional** $p \leftrightarrow q$ or **equivalence** $p \equiv q$. Notice that the ordinary assignment or equality symbols

are written with two short horizontal bars, =, while the equivalence is written with three, ≡. In other words, p and q always take the same truth values. If p is T then q is true and if p is F then q is F. The argument becomes:

```
(1) C → Y
(2) Y ≡ D
(3) C___
    ∴ D
```

where numbers are now used to identify the premises. Since Y and D are equivalent from (2), we can substitute D for Y in (1) to yield:

```
(4) C → D
```

where (4) is an inference made on the basis of (1) and (2). Premises (3) and (4) and the conclusion are:

```
(4) C → D
(3) C___
    ∴ D
```

which can be recognized as a schema of modus ponens. Hence the argument is valid.

The substitution of one variable that is equivalent to another is a rule of inference called the **rule of substitution**. The rules of modus ponens and substitution are two basic rules of deductive logic.

A formal logic proof is usually written by numbering the premises, conclusion, and inferences as follows:

```
1. C → Y
2. (Y → D) ∧ (D → Y)
3. C                        / ∴ D
4. Y ≡ D                    2 Equivalence
5. C → D                    1 Substitution
6. D                        3,5 Modus Ponens
```

Lines 1, 2, and 3 are the premises and conclusion while 4, 5, and 6 are the inferences obtained. The right-hand column lists the rule of inference and line numbers used to justify the inference.

3.7 LIMITATIONS OF PROPOSITIONAL LOGIC

Consider our familiar classic argument:

```
All men are mortal
Socrates is a man
Therefore, Socrates is mortal
```

We know that this is a valid argument since it is a valid syllogism. Can we prove its validity using propositional logic? To answer this question, let's first write the argument as a schema:

```
p = All men are mortal
q = Socrates is a man
r = Socrates is mortal
```

and so the argument schema is:

```
p
q
∴ r
```

Notice that there are no logical connectives in the premises or conclusions and so each premise and each conclusion must have a different logical variable. Also, propositional logic has no provision for quantifiers and so there is no way to represent the quantifier "all" in the first premise. The only representation of this argument in propositional logic is thus the schema above of three independent variables.

To determine if this is a valid argument, consider the truth table of three independent variables for all possible combinations of T and F, shown in Table 3.11. The second row of this truth table shows the argument to be invalid because the premises are true while the conclusion is false.

Table 3.11 Truth Table for the Schema p, q; ∴ r

p	q	∴r
T	T	T
T	T	F
T	F	T
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F

The invalidity of this argument should *not* be interpreted as meaning the conclusion is incorrect. Any person would recognize this as a correct argument. The invalidity simply means that *the argument cannot be proven under propositional logic*. The argument can be proven valid if we examine the internal structure of the premises. For example, we would have to attribute some meaning to "all" and recognize "men" as the plural of "man." However, syllogisms and the propositional calculus do not allow the internal structure of propositions to be examined. This limitation is overcome by predicate logic, and this argument is a

valid argument under predicate logic. In fact, all of syllogistic logic is a valid subset of first-order predicate logic and can be proven valid under it.

The only valid syllogistic form of the proposition is:

```
If Socrates is a man, then Socrates is mortal
Socrates is a man
Therefore, Socrates is mortal
```

Let:

```
p = Socrates is a man
q = Socrates is mortal
```

The argument becomes:

```
p → q
p
∴ q
```

which is a valid syllogistic form of modus ponens.

As another example, consider the following classic argument:

```
All horses are animals
Therefore, the head of a horse
   is the head of an animal
```

We know that this argument is correct and yet it cannot be proved under propositional logic, although it can be proven under predicate logic (see Problem 3.12).

3.8 FIRST-ORDER PREDICATE LOGIC

Syllogistic logic can be completely described by predicate logic. Table 3.12 shows the four categorical statements and their representation in predicate logic. In addition to the rules of inference previously discussed, predicate logic has rules that deal with quantifiers.

Table 3.12 Representation of the Four Categorical Syllogisms Using Predicate Logic

Type	Schema	Predicate Representation
A	All S is P	$(\forall x)\,(S(x) \rightarrow P(x))$
E	No S is P	$(\forall x)\,(S(x) \rightarrow \sim P(x))$
I	Some S is P	$(\exists x)\,(S(x) \wedge P(x))$
O	Some S is not P	$(\exists x)\,(S(x) \wedge \sim P(x))$

The Rule of Universal Instantiation essentially states that an individual may be substituted for a universal. For example, if ϕ is any proposition or **propositional function:**

$$\frac{(\forall\ x)\ \phi(x)}{\therefore\ \phi(a)}$$

is a valid inference, where *a* is an instance. That is, *a* refers to a specific individual while *x* is a variable that ranges over all individuals. For example, this can be used to prove that Socrates is human:

$$\frac{(\forall\ x)\ H(x)}{\therefore\ H(\text{Socrates})}$$

where H(x) is the propositional function that says *x* is a human. The above states that for every *x*, that *x* is human, and so by inference Socrates is human.

Other examples of the Rule of Universal Instantiation are:

$$\frac{(\forall\ x)\ A(x)}{\therefore\ A(c)}$$

$$\frac{(\forall\ y)\ (B(y) \vee C(b))}{\therefore\ B(a) \vee C(b)}$$

$$\frac{(\forall\ x)\ [A(x) \wedge (\exists\ x)\ (B(x) \vee C(y))]}{\therefore\ A(b) \wedge (\exists\ x)\ (B(x) \vee C(y))}$$

In the first example, the instance *c* is substituted for x. In the second example, notice that the instance *a* is substituted for *y* but not for *b* since *b* is not included in the **scope** of the quantifier. That is, a quantifier such as ∀ x applies only to x variables. The variables such as x and y used with quantifiers are called **bound** while the others are called **free**. In the third example, the quantifier x has as its scope only A(x). That is, ∀ x does not apply to the existential quantifier ∃ x and its scope over B(x) ∨ C(y). The convention of nested quantifiers such as this is that the scope ends when a new quantifier is used, even if it uses the same variable, such as *x*. The formal proof of the syllogism:

$$\frac{\begin{array}{l}\text{All men are mortal}\\ \text{Socrates is a man}\end{array}}{\therefore\ \text{Socrates is mortal}}$$

is shown following, where H = man, M = mortal, and *s* = Socrates:

1. (∀ x) (H(x) → M(x))	
2. H(s)	/∴ M(s)
3. H(s) → M(s)	1 Universal Instantiation
4. M(s)	2,3 Modus Ponens

3.9 LOGIC SYSTEMS

A **logic system** is a collection of objects such as rules, axioms, statements, and so forth organized in a consistent manner. The logic system has several goals.

The first goal is to specify the forms of arguments. Since logical arguments are meaningless in a semantic sense, a valid form is essential if the validity of the argument is to be determined. Thus, one important function of a logic system is to determine the **well-formed formulas** (**wffs**) that are used in arguments. Only wffs can be used in logic arguments. For example, in syllogistic logic,

```
All S is P
```

could be a wff, but:

```
All
All is S P
Is S all
```

are not wffs. Although the symbols of the alphabet are meaningless, the sequence of symbols that make up the wff is meaningful.

The second goal of a logic system is to indicate the rules of inference that are valid. The third goal of a logic system is to extend itself by discovering new rules of inference and so extend the range of arguments that can be proven. By extending the range of arguments, new wffs, called **theorems**, can be proven by a logic argument.

When a logic system is well developed it can be used to determine the validity of arguments in a way that is analogous to calculations in systems such as arithmetic, geometry, calculus, physics, and engineering. Logic systems have been developed such as the Sentential or Propositional Calculus, the Predicate Calculus, and so forth. Each system relies on formal definitions of its **axioms** or **postulates**, which are the fundamental definitions of the system. From these axioms people (and sometimes computer programs such as the Automated Mathematician) try to determine what can be proven. Anyone who has studied Euclidian geometry in high school is familiar with axioms and the derivation of geometric theorems. Just as geometric theorems can be derived from geometric axioms, so can logic theorems be derived from logic axioms.

An axiom is simply a fact or **assertion** that cannot be proven from within the system. Sometimes we accept certain axioms because they make "sense" by appealing to commonsense or observation. Other axioms, such as "parallel lines meet at infinity," do not make intuitive sense because they appear to contradict Euclid's axiom of parallel lines as never meeting. However, this axiom about parallel lines meeting at infinity is just as reasonable from a purely logical viewpoint as Euclid's and is the basis of one type of non-Euclidean geometry.

A formal system requires the following:

1. An alphabet of symbols.
2. A set of finite strings of these symbols, the wffs.
3. Axioms, the definitions of the system.

4. Rules of inference, which enable a wff, A, to be deduced as the conclusion of a finite set, G, of other wffs where G= {A1, A2 ... An}. These wffs must be axioms or other theorems of the logic system. For example, a propositional logic system can be defined using only modus ponens to derive new theorems.

If the argument:

$A_1, A_2, \ldots A_N; \therefore A$

is valid, then A is said to be a **theorem** of the formal logic system and is written with the symbol ⊢. For example, $\Gamma \vdash A$ means that A is a theorem of the set of wffs, G. A more explicit schema of a proof that A is a theorem is the following:

$A_1, A_2, \ldots A_N \vdash A$

The symbol ⊢, which indicates that the following wff is a theorem, is not a symbol of the system. Instead, ⊢ is a **metasymbol,** because it is used to describe the system itself. An analogy is a computer language such as Java. Although programs can be specified using Java's syntax, there is no syntax in Java for indicating a valid program.

A rule of inference in a formal system specifies exactly how new assertions, the theorems, can be obtained from axioms and previously derived theorems. An example of a theorem is our syllogism about Socrates, written in predicate logic form:

$(\forall x)\ (H(x) \rightarrow M(x)), H(s) \vdash M(s)$

where H is the predicate function for man and M is the predicate function for mortal. Since M(s) can be proven from its axioms on the left, it is a theorem of these axioms. However, note that M(Zeus) would not be a theorem since Zeus, the Greek god, is not a man, and there is no alternative way of showing M(Zeus).

If a theorem is a tautology, it follows that Γ is the null set since the wff is always true and so does not depend on any other axioms or theorems. A theorem that is a tautology is written with the symbol ⊨ as in:

$\models A$

For example, if $A \equiv p \vee \sim p$, then:

$\models p \vee \sim p$

states that $p \vee \sim p$ is a theorem, which is a tautology. Notice that whatever values are assigned to p, either T or F, the theorem $p \vee \sim p$ is always true. An assignment of truth values is an **interpretation** of a wff. A **model** is an interpretation in which the wff is true. For example, a model of $p \rightarrow q$ is p = T and q = T. A wff is called **consistent** or **satisfiable** if there is an interpretation that makes it

true, and **inconsistent** or **unsatisfiable** if the wff is false in *all* interpretations. An example of an inconsistent wff is $p \wedge \sim p$.

A wff is **valid** if it is true in all interpretations; else it is **invalid**. For example, the wff $p \vee \sim p$ is valid, while $p \rightarrow q$ is an invalid wff since it is not true for p = T and q = F. A wff is **proved** if it can be shown to be valid. All propositional wffs can be proved by the truth table method since there is only a finite number of interpretations for wffs and so the propositional calculus is **decidable**. However, the predicate calculus is not decidable since there is no general method of proof like truth tables for all predicate calculus wffs.

One example of a valid predicate calculus wff that can be proved is that for any predicate B,

$$(\exists\ x)\ B(x) \rightarrow \sim\ [(\forall\ x)\ \sim B(x)]$$

which shows how the existential quantifier can be replaced by the universal quantifier. This predicate calculus wff is therefore a theorem.

There is a big difference between an expression like $\vdash$ A and $\nvdash$ B. The *A* is a theorem and so can be proven from the axioms by the rules of inference. The *B* is a wff and there may be no known proof to show its derivation. While propositional logic is decidable, predicate logic is not. That is, there is no mechanical procedure or algorithm for finding the proof of a predicate logic theorem in a finite number of steps. In fact, there is a theoretical proof that there is no decision procedure for predicate logic. However, there are decision procedures for subsets of predicate logic like syllogisms and propositional logic. Sometimes predicate logic is referred to as **semidecidable** because of this.

Note that PROLOG is based on predicate logic. In the first rush of enthusiasm toward it in the 1970s, the Japanese announced plans for their 5th Generation ultra-advanced computer system. This system would have allowed natural language input and output and really understood what was being said. However there was not enough computerized commonsense knowledge available nor were the microprocessors of the 1980s sufficiently powerful enough to allow voice recognition with a high degree of accuracy. Now with the Open.Cyc project for commonsense ontologies and good voice recognition with no training, such a system has a stronger chance of success (although not with PROLOG).

As a very simple example of a complete formal system, define the following:

Alphabet : The single symbol "1"
Axiom : The string "1" (which happens to be the same as the symbol 1)
Rule of Inference : If any string $ is a theorem, then so is the string $11. This rule can be written as a Post production rule,

```
$ → $11.
   If $ = 1 then this rule gives $11 = 111.
   If $ = 111 then the rule gives $11 = 11111 and in
      general
      1, 111, 11111, 1111111, ...
```

The strings above are the theorems of this formal system.

Although strings like 11111 do not look like the types of theorems we are used to seeing, they are perfectly valid logic theorems. These particular theorems also have a semantic meaning because they are the odd numbers expressed in a **unary number system** of the single symbol 1. Just as the binary number system has only the alphabet symbols 0 and 1, the unary number system has only the single symbol 1. Numbers in the unary and decimal system are expressed as:

Unary	Decimal
1	1
11	2
111	3
1111	4
11111	5

and so forth.

Notice that because of our rule of inference and axiom, the strings 11, 1111, and so forth cannot be expressed in our formal system. That is, 11 and 1111 are certainly strings from our formal alphabet, but they are not theorems or wffs because they cannot be proven using only the rule of inference and the axiom. This formal system allows only the derivation of the odd numbers, not the even numbers. The axiom "11" must be added in order to be able to derive the even numbers.

Another property of a formal system is **completeness**. A set of axioms is **complete** if every wff can either be proved or **refuted**. The term *refute* means to prove some assertion is false. In a complete system, every logically valid wff is a theorem. However, since predicate logic is not decidable, coming up with a proof depends on our luck and cleverness. Of course, another possibility is writing a computer program that will try to derive proofs and let it grind away.

A further desirable property of a logical system is that it be **sound**. A sound system means that every theorem is a logically valid wff. In other words, a sound system will not allow a conclusion to be inferred that is not a logical consequence of its premises. No invalid arguments will be inferred as valid.

There are different **orders** of logic. A **first-order** language is defined so that the quantifiers operate on objects that are variables such as $\forall$ x. A **second-order** language would have additional features such as two kinds of variables and quantifiers. In addition to the ordering variables and quantifiers, the second-order logic can have quantifiers that range over function and predicate symbols. An example of second-order logic is the **equality axiom**, which states that two objects are equal if all predicates of them are equal. If P is any predicate of one argument, then:

$$x = y \equiv (\forall P)\ [P(x) \leftrightarrow P(y)]$$

is a statement of the equality axiom using a second-order quantifier, $\forall$ P, which ranges over all predicates.

3.10 RESOLUTION

The very powerful **resolution** rule of inference introduced by Robinson in 1965 is commonly implemented in theorem-proving AI programs. In fact, resolution is the primary rule of inference in PROLOG. Instead of many different inference rules of limited applicability such as modus ponens, modus tollens, merging, chaining, and so forth, PROLOG uses the one general-purpose inference rule of resolution. This application of resolution makes automatic theorem provers such as PROLOG practical tools for solving problems. Instead of having to try different rules of inference and hoping one succeeds, the single rule of resolution can be applied. This approach can greatly reduce the search space.

As a way of introducing resolution, let's first consider the syllogism about Socrates expressed in PROLOG as follows, where comments are shown with a percent sign:

```
mortal(X) :- man(X).     % All men are mortal
man(socrates).           % Socrates is a man
:- mortal(socrates).     % Query - is Socrates mortal?
yes                      % PROLOG answers yes
```

PROLOG uses a **quantifier-free** notation. Notice that the universal quantifier, ∀, is implied in the statement that all men are mortal.

PROLOG is based on first-order predicate logic. However it also has a number of extensions to make it easier for programming applications. These special programming features violate pure predicate logic and are called **extralogical features**: input/output, cut (which alters the search space), and assert/retract (to alter truth values without any logical justification).

Before resolution can be applied, the wff must be in a **normal** or standard form. The three main types of normal forms are **conjunctive normal form**, clausal form, and its Horn clause subset. The basic idea of normal form is to express wffs in a standard form that uses only the ∧, ∨, and possibly ~. The resolution method is then applied to normal form wffs in which all other connectives and quantifiers have been eliminated. This conversion to normal form is necessary because resolution is an operation on pairs of **disjuncts**, which produces new disjuncts, which simplifies the wff.

The following illustrates a wff in conjunctive normal form, which is defined as the conjunction of disjunctions that are **literals:**

$$(P_1 \vee P_2 \vee \ldots) \wedge (Q_1 \vee Q_2 \vee \ldots) \wedge \ldots (Z_1 \vee Z_2 \vee \ldots)$$

Terms such as P_i must be literals, which mean that they contain no logical connectives such as the conditional and **biconditional**, or quantifiers. A literal is an atomic formula or a negated atomic formula. For example, the following wff:

$$(A \vee B) \wedge (\sim B \vee C)$$

is in conjunctive normal form. The terms within parentheses are clauses:

$$A \vee B \text{ and } \sim B \vee C$$

As will be shown later, any predicate logic wff, which includes propositional logic as a special case, can be written as clauses. The full **clausal form** can express any predicate logic formula but may not be as natural or readable for a person. The syntax of PROLOG is the Horn clause subset, which makes mechanical theorem proving by PROLOG much easier and efficient to implement than standard predicate logic notation or full clausal form. As mentioned in Chapter 1, PROLOG allows only one head. A full clausal form expression is generally written in a special form called Kowalski clausal form:

$$A_1, A_2, \ldots A_N \rightarrow B_1, B_2, \ldots B_M$$

which is interpreted as saying that if all the subgoals A_1, A_2, . . . A_N are true, then one or more of B_1 or B_2 . . . or B_M are true also. Note that sometimes the direction of the arrow is reversed in this notation. This clause, written in standard predicate notation, is:

$$A_1 \wedge A_2 \ldots A_N \rightarrow B_1 \vee B_2 \ldots B_M$$

This can be expressed in **disjunctive form** as the disjunction of literals using the equivalence:

$$p \rightarrow q \equiv \sim p \vee q$$

so:

$$\begin{aligned} A_1 \wedge A_2 \ldots A_N &\rightarrow B_1 \vee B_2 \ldots B_M \\ &\equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee (B_1 \vee B_2 \ldots B_M) \\ &\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee B_1 \vee B_2 \ldots B_M \end{aligned}$$

where de Morgan's law:

$$\sim(p \wedge q) \equiv \sim p \vee \sim q$$

is used to simplify the last expression.

As discussed in Chapter 1, PROLOG uses a restricted type of clausal form, the Horn clause, in which only one head is allowed:

$$A_1, A_2, \ldots A_N \rightarrow B$$

which is written in PROLOG syntax as:

$$B \text{ :- } A_1, A_2, \ldots A_N$$

The problem with trying to prove a theorem directly is the difficulty of deducing it using only the rules of inference and axioms of the system. It may take a very long time to derive a theorem or we may not be clever enough to derive it at all. To prove a theorem is true the classical method of **reductio ad absurdum**, or *method of contradiction*, is used. In this method we try to prove the negated wff is a theorem. If a contradiction results, then the original non-negated wff is a theorem.

The basic goal of resolution is to infer a new clause, the **resolvent**, from two other clauses called **parent clauses**. The resolvent will have fewer terms than the parents. By continuing the process of resolution, eventually a contradiction will be obtained or the process is terminated because no progress is being made. A simple example of resolution is shown in the following argument:

$$\begin{array}{l} A \vee B \\ \underline{A \vee \sim B} \\ \therefore A \end{array}$$

One way of seeing how the conclusion follows is by writing the premises as:

$$(A \vee B) \wedge (A \vee \sim B)$$

One of the Axioms of Distribution is:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

Applying this to the premises gives:

$$(A \vee B) \wedge (A \vee \sim B) \equiv A \vee (B \wedge \sim B) \equiv A$$

where the last step follows since $(B \wedge \sim B)$ is always false. This follows from the Law of the Excluded Middle, which states that something cannot be both true and false. In fuzzy logic, discussed in Chapter 5, we'll see that this law does not hold. Another way of writing this uses the term **nil** or **null**, which means empty, nothing, or false. For example, a null pointer in C points to nothing, while the Law of the Excluded Middle states

$$(B \wedge \sim B) \equiv \text{nil}.$$

The example of resolution shows how the parent clauses $(A \vee B)$ and $(A \vee \sim B)$ can be simplified into the resolvent A. Table 3.13 summarizes some basic parent clauses and their resolvents in clause notation, where the comma separating clauses means $\wedge$.

3.11 RESOLUTION SYSTEMS AND DEDUCTION

Given wffs $A_1, A_2, \ldots A_N$ and a logical conclusion or theorem C, we know:

$$A_1 \wedge A_2 \ \ldots \ A_N \vdash C$$

Table 3.13 Clauses and Resolvents

Parent Clauses	Resolvent	Meaning
$p \rightarrow q$, p or $\sim p \vee q$, p	q	Modus Ponens
$p \rightarrow q$, $q \rightarrow r$ or $\sim p \vee q$, $\sim q \vee r$	$p \rightarrow r$ or $\sim p \vee r$	Chaining or Hypothetical Syllogism
$\sim p \vee q$, $p \vee q$	q	Merging
$\sim p \vee \sim q$, $p \vee q$	$\sim p \vee p$ or $\sim q \vee q$	TRUE (a tautology)
$\sim p$, p	nil	FALSE (a contradiction)

is equivalent to stating that:

$$(1)\ A_1 \wedge A_2 \ \ldots\ A_N \rightarrow C \equiv \sim(A_1 \wedge A_2 \ \ldots\ A_N) \vee C \equiv \sim A_1 \vee \sim A_2 \ \ldots\ \sim A_N \vee C$$

is valid. Suppose we take the negation as follows:

$$\sim[A_1 \wedge A_2 \ \ldots\ A_N \rightarrow C]$$

Now:

$$p \rightarrow q \equiv \sim p \vee q$$

and so the above becomes:

$$\sim[A_1 \wedge A_2 \ \ldots\ A_N \rightarrow C] \equiv \sim[\sim(A_1 \wedge A_2 \ \ldots\ A_N) \vee C]$$

From de Morgan's laws:

$$\sim(p \vee q) \equiv \sim p \wedge \sim q$$

and so the above becomes:

$$(2)\ \sim[A_1 \wedge A_2 \ \ldots\ A_N \rightarrow C] \equiv [\sim\sim(A_1 \wedge A_2 \ \ldots\ A_N) \wedge \sim C] \equiv A_1 \wedge A_2 \ \ldots\ A_N \wedge \sim C$$

Now if (1) is valid, then its negation (2) must be invalid. In other words, if (1) is a tautology then (2) must be a contradiction. Formulas (1) and (2) represent two equivalent ways of proving that a formula C is a theorem. Formula (1) can be used to prove a theorem by checking to see if it is true in all cases. Equivalently, formula (2) can be used to prove a theorem by showing (2) leads to a contradiction.

As mentioned in the previous section, proving a theorem by showing its negation leads to a contradiction is proof by reductio ad absurdum. The primary

part of this type of proof is the **refutation**. To refute something means to prove it false. Resolution is a sound rule of inference that is also **refutation complete** because the empty clause will always be the eventual result if there is a contradiction in the set of clauses. Essentially this means that **resolution refutation** will terminate in a finite number of steps if there is a contradiction. Although resolution refutation can't tell us how to *produce* a theorem, it will definitely tell us if a wff *is* a theorem.

As a simple example of proof by resolution refutation, consider the argument:

$$\begin{array}{l} A \rightarrow B \\ B \rightarrow C \\ \underline{C \rightarrow D} \\ \therefore\ A \rightarrow D \end{array}$$

To prove that the conclusion $A \rightarrow D$ is a theorem by resolution refutation, first convert it to disjunctive form using the equivalence:

$$p \rightarrow q \equiv \sim p \vee q$$

So:

$$A \rightarrow D \equiv \sim A \vee D$$

and its negation is:

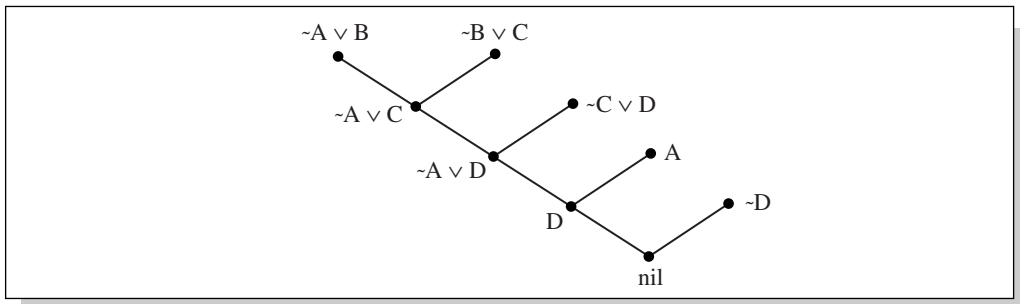
$$\sim(\sim A \vee D) \equiv A \wedge \sim D$$

The conjunction of the disjunctive forms of the premises and the negated conclusion gives the conjunctive normal form suitable for resolution refutation.

$$(\sim A \vee B) \wedge (\sim B \vee C) \wedge (\sim C \vee D) \wedge A \wedge \sim D$$

The resolution method can now be applied to the premises and conclusion. Figure 3.18 shows a method of representing resolution refutation in the form of a **resolution refutation tree diagram**, where clauses on the same level are

Figure 3.18 Resolution Refutation Tree



resolved. The root, which is the final resolvent, is nil, as can be verified from the last row of Table 3.13 for ~p, p, and so the original conclusion A → D is a theorem.

3.12 SHALLOW AND CAUSAL REASONING

Resolution systems and production rule systems are two popular paradigms for proving theorems. Although most people think of a theorem in the mathematical sense, we have seen that a theorem is actually the conclusion of a valid logical argument. Now consider an expert system that uses an inference chain. In general, a longer chain represents more causal or deep knowledge, while shallow reasoning commonly uses a single rule or a few inferences. Besides the length of the inference chain, the quality of knowledge in the rules is also a major factor in determining deep and shallow reasoning. Sometimes another definition of shallow knowledge is used, called **experiential knowledge**, which is knowledge based on experience.

The conclusion of an inference chain is a theorem because it is proven by the chain of inference, as demonstrated by the previous example:

$$A \rightarrow B,\ B \rightarrow C,\ C \rightarrow D \vdash A \rightarrow D$$

In fact, expert systems that use an inference chain to establish a conclusion are really using theorems. This result is very important because otherwise we could not use expert systems for causal inference. Instead, expert systems would be restricted to shallow inferences of single rules with no chaining.

Let's look at some rules now in order to better contrast shallow and deep reasoning. As a first example, consider the following rule, where the number in parentheses is for identification purposes only:

```
(1) IF a car has
       a good battery
       good sparkplugs
       gas
       good tires
    THEN the car can move
```

This is a perfectly good rule that could be used in an expert system.

One of the important features of an expert system is the explanation facility, as discussed in Chapter 1. Rule-based expert systems make it easy for the system to explain its reasoning. In this case, if the user asked how the car can move, the expert system could respond by listing its conditional elements:

```
a good battery
good sparkplugs
gas
good tires
```

This is an elementary type of explanation facility since the system lists only the conditional elements of the rule. More sophisticated explanation facilities can be designed to list previous rules that have fired and resulted in the current rule firing. Other explanation facilities may allow the user to ask "What if" type questions to explore alternative reasoning paths.

This rule is also an example of **shallow reasoning**. That is, there is little or no understanding of cause and effect in shallow reasoning because there is little or no inference chain. The previous rule is essentially a heuristic in which all the knowledge is contained in the rule. The rule becomes activated when its conditional elements are satisfied and not because there is any understanding by the expert system of what function the conditional elements perform. In shallow reasoning there is little or no **causal chain** of cause and effect from one rule to another. In the simplest case, the cause and effect are contained in one with no relationship to any other rule. If you think of rules in terms of the chunks of knowledge discussed in Chapter 1, shallow reasoning makes no connections between chunks and so is like a simple reflex reaction.

The advantage of shallow reasoning compared to causal reasoning is the ease of programming. Easier programming means the development time is shorter and the program is smaller, faster, and costs less to develop.

Frame are useful for causal or **deep reasoning**. The term *deep* is often used synonymously for causal reasoning to imply a deep understanding of the subject. However, a deep understanding implies that besides understanding the causal chain by which a process occurs, you also understand the process in an abstract sense.

We can add simple causal reasoning to our rule by defining additional rules such as:

```
(2) IF the battery is good
    THEN there is electricity

(3) IF there is electricity
       and the sparkplugs are good
    THEN the sparkplugs will fire

(4) IF the sparkplugs fire
       and there is gas
    THEN the engine will run

(5) IF the engine runs and
       there are good tires
    THEN the car will move
```

Notice that with causal reasoning, the explanation facility can give a good explanation of what each car component does since each element is specified by a rule. Such a causal system also makes it easier to write a diagnostic system to determine what effect a bad component will have. Causal reasoning may be used for an arbitrary refinement of the operation of a system limited by the speed of execution, memory size, and increasing development cost.

Causal reasoning can be used to construct a **model** of the real system that behaves in all respects like the real system. Such a model can be used for simulation to explore hypothetical reasoning of the "What if" type of queries. However, causal models are neither always necessary nor desirable. For example, the classic MUD expert system serves as a consultant to drilling fluid or mud engineers. Drilling fluid, called mud because of its resemblance to mud, is an important aid in drilling for a number of reasons, such as cooling and lubrication of the bit. MUD diagnoses problems with the mud and suggests treatments.

A causal system would not be of much use because the drilling engineer cannot normally observe the causal chain of events occurring far below the ground. Instead, the engineer can observe only the symptoms on the surface and not the unobservable intermediate events of potential diagnostic significance.

The situation is very different in medicine where physicians have a wide range of diagnostic tests that can be used to verify intermediate events. For example, if a person complains of feeling ill, the physician can check for fever. If there is a fever, there may be an infection and so a blood test may be done. If the blood test reveals a tetanus infection, the physician may check the person for recent cuts from a rusty object. In contrast, if the drilling fluid becomes salty, the drilling engineer may suspect the drill has gone through a salt dome. However, there is no simple way to check this since it's not possible to go into the hole (except by robot), and a seismic test is expensive and not always reliable. Because drilling engineers cannot normally test intermediate hypotheses the way that physicians can, they do not approach diagnostic problems the way that physicians do, and MUD reflects this approach.

Another reason for not using causal reasoning in MUD is that there are a limited number of diagnostic possibilities and symptoms. Most of the relevant tests used by MUD are conducted routinely and input in advance. There is little advantage in an interactive query session with an engineer to explore alternate diagnostic paths if the system knows all the relevant tests and diagnostic paths. If there were many possible diagnostic paths to follow with a verifiable intermediate hypothesis, there would be an advantage to causal knowledge because the engineer could work with the system to prune the search to likely paths.

Because of the increased requirements for causal reasoning, it may become necessary to combine certain rules into a shallow reasoning one. The resolution method with refutation can be used to prove that a single rule is a true conclusion of multiple rules. The single rule is the theorem that will be proved by resolution.

As an example, suppose we want to prove that rule (1) is a logical conclusion of rules (2) – (5). Using the following propositional definitions, the rules can be expressed as follows:

```
B = battery is good       C = car will move
E = there is electricity  F = sparkplugs will fire
G = there is gas          R = engine will run
S = sparkplugs are good   T = there are good tires
```

(1) B ∧ S ∧ G ∧ T → C
(2) B → E
(3) E ∧ S → F
(4) F ∧ G → R
(5) R ∧ T → C

The first step in applying resolution refutation is to negate the conclusion or goal rule:

(1') ~(B ∧ S ∧ G ∧ T → C) = ~[~(B ∧ S ∧ G ∧ T) ∨ C]
= ~[~B ∨ ~S ∨ ~G ∨ ~T ∨ C]

Now each of the other rules is expressed in disjunctive form using equivalences such as:

p → q ≡ ~p ∨ q and ~(p ∧ q) ≡ ~p ∨ ~q

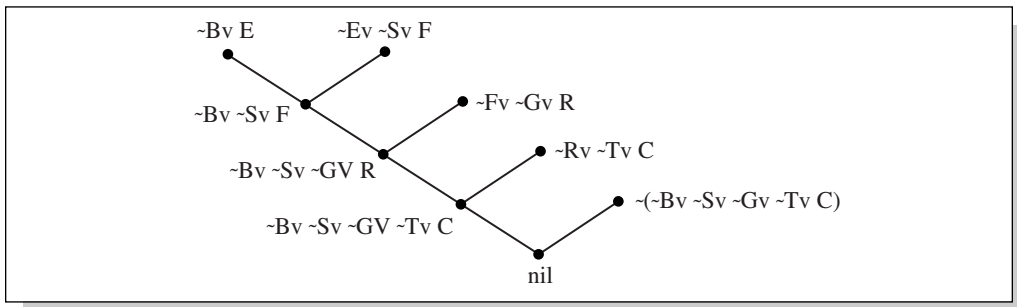
to yield the following new versions of (2) – (5):

(2') ~B ∨ E
(3') ~(E ∧ S) ∨ F = ~E ∨ ~S ∨ F
(4') ~(F ∧ G) ∨ R = ~F ∨ ~G ∨ R
(5') ~(R ∧ T) ∨ C = ~R ∨ ~T ∨ C

As described in the previous section, a convenient way of representing the successive resolvents of (1') – (5') is with a resolution refutation tree, as shown in Figure 3.19. Starting at the top of the tree, the clauses are represented as nodes, which are resolved to produce the resolvent below. For example,

~B ∨ E and ~E ∨ ~S ∨ F

Figure 3.19 Resolution Refutation Tree for the Car Example



are resolved to infer:

$$\sim B \vee \sim S \vee F$$

which is then resolved with:

$$\sim F \vee \sim G \vee R$$

to infer:

$$\sim B \vee \sim S \vee \sim G \vee R$$

and so forth. For simplicity in drawing, the last resolvents are implied, rather than drawing in every single one.

Since the root of the tree is nil, this is a contradiction. By refutation, the original conclusion:

$$B \wedge S \wedge G \wedge T \rightarrow C$$

is a theorem since its negation leads to a contradiction. Thus rule (1) does logically follow from rules (2) – (5).

3.13 RESOLUTION AND FIRST-ORDER PREDICATE LOGIC

The resolution method is also used with the first-order predicate logic. In fact, it is the primary inference mechanism of PROLOG. However, before resolution can be applied, a wff must be put in clausal form. As an example,

Some programmers hate all failures
No programmer hates any success
∴ No failure is a success

Define the following predicates:

P(x) = x is a programmer
F(x) = x is a failure
S(x) = x is a success
H(x,y) = x hates y

The premises and negated conclusion are written as:

$$(1)\ (\exists x)\ [P(x) \wedge (\forall y)\ (F(y) \rightarrow H(x,y))]$$
$$(2)\ (\forall x)\ [P(x) \rightarrow (\forall y)\ (S(y) \rightarrow \sim H(x,y))]$$
$$(3)\ \sim(\forall y)\ (F(y) \rightarrow \sim S(y))$$

where the conclusion has been negated in preparation for resolution.

Conversion to Clausal Form

The following nine steps are an algorithm to convert first-order predicate wffs to clausal form. This procedure is illustrated using wff (1) from the previous page.

1. Eliminate conditionals, →, using the equivalence:

   $p \rightarrow q \equiv \sim p \vee q$

   so the wff (1) becomes:

   $(\exists x)\ [P(x) \wedge (\forall y)\ (\sim F(y) \vee H(x,y))]$

2. Wherever possible, eliminate negations or reduce the scope of negation to one atom. Use the equivalences shown in Appendix A such as:

   $\sim\sim p \equiv p$
   $\sim(p \wedge q) \equiv \sim p \vee \sim q$
   $\sim(\exists x)\ P(x) \equiv (\forall x)\ \sim P(x)$
   $\sim(\forall x)\ P(x) \equiv (\exists x)\ \sim P(x)$

3. Standardize variables within a wff so that the bound or dummy variables of each quantifier have unique names. Note that the variable names of a quantifier are dummies. That is,

   $(\forall x)\ P(x) \equiv (\forall y)\ P(y) \equiv (\forall z)\ P(z)$

   and so the standardized form of:

   $(\exists x)\ \sim P(x) \vee (\forall x)\ P(x)$

   is:

   $(\exists x)\ \sim P(x) \vee (\forall y)\ P(y)$

4. Eliminate existential quantifiers, ∃, by using **Skolem functions**, named after the Norwegian logician, Thoralf Skolem. Consider the wff:

   $(\exists x)\ L(x)$

   where L(x) is defined as the predicate, which is true if x is < 0. This wff can be replaced by:

   $L(a)$

   where a is a constant such as -1 that makes L(a) true. The a is called a Skolem constant, which is a special case of the Skolem function. For the case in which there is a universal quantifier in front of an existential one,

```
(∀ x) (∃ y) L(x,y)
```

where L(x,y) is true if the integer x is less than the integer y. This wff means that for every integer x there is an integer y that is greater than x. Notice that the formula does not tell how to compute y given a value for x. Assume a function $f(x)$ exists which produces a y greater than x. So the above wff becomes Skolemized as:

```
(∀ x) L(x,f(x))
```

The Skolem function of an existential variable within the scope of a universal quantifier is a function of all the quantifiers on the left. For example,

```
(∃ u) (∀ v) (∀ w) (∃ x) (∀ y) (∃ z) P(u,v,w,x,y,z)
```

is Skolemized as:

```
(∀ v) (∀ w) (∀ y) P(a,v,w,f(v,w),y,g(v,w,y))
```

where a is some constant and the second Skolem function, g, must be different from the first function, f. Our example wff becomes:

```
P(a) ∧ (∀ y) (~F(y) ∨ H(a,y))
```

5. Convert the wff to **prenex form**, which is a sequence of quantifiers, Q, followed by a **matrix** M. In general, the quantifiers can be either ∀ or ∃. However, in this case step 4 has already eliminated all existential quantifiers and so the Q can only be ∀. Also, since each ∀ has its own dummy variable, all the ∀ can be moved to the left of the wff and the scope of each ∀ can be the entire wff.

   Our example becomes:

   ```
   (∀ y) [P(a) ∧ (~F(y) ∨ H(a,y)]
   ```

   where the matrix is the term in brackets.

6. Convert the matrix to conjunctive normal form, which is a conjunctive of clauses. Each clause is a disjunction. Our example is already in conjunctive normal form where one clause is P(a) and the other is (~F(y) ∨ H(a,y)). If necessary, the following distributive rule can be used as necessary to put the matrix in conjunctive normal form:

   ```
   p ∨ (q ∧ r) ≡ (p ∨ q) ∧ (p ∨ r)
   ```

7. Drop the universal quantifiers as unnecessary since all the variables in a wff at this stage must be bound. The wff is now the matrix. Our example wff becomes:

   ```
   P(a) ∧ (~F(y) ∨ H(a,y))
   ```

8. Eliminate the ∧ signs by writing the wff as a set of clauses. Our example is:

```
{ P(a), ~F(y) ∨ H(a,y) }
```

which is usually written without the braces as the clauses:

```
P(a)
~F(y) ∨ H(a,y)
```

9. Rename variables in clauses, if necessary, so that the same variable name is used in only one clause. For example, if we had the clauses:

```
P(x) ∧ Q(x) ∨ L(x,y)
~P(x) ∨ Q(y)
~Q(z) ∨ L(z,y)
```

these could be renamed as:

```
P(x1) ∧ Q(x1) ∨ L(x1,y1)
~P(x2) ∨ Q(y2)
~Q(z) ∨ L(z,y3)
```

If we carry out the procedure to convert to clausal form the second premise and the negated conclusion of our example, we finally obtain the clauses:

```
(1a) P(a)
(1b) ~F(y) ∨ H(a,y)
(2a) ~P(x) ∨ ~S(y) ∨ ~H(x,y)
(3a) F(b)
(3b) S(b)
```

where the numbers refer to the original premises and negated conclusion. Thus, premises (1) and (3) are converted to two clauses with suffixes (a) and (b), while premise (2) is converted to the single clause (2a).

Unification and Rules

Once the wffs have been converted to clausal form, it is usually necessary to find appropriate **substitution instances** for the variables. That is, clauses such as:

```
~F(y) ∨ H(a,y)
F(b)
```

cannot be resolved on the predicate *F* until the arguments of *F* match. The process of finding substitutions for variables to make arguments match is called **unification**.

Unification is one feature that distinguishes expert systems from simple decision trees. Without unification, the conditional elements of rules could match only constants. This means that a specific rule would have to be written for every possible fact. For example, suppose someone wanted to sound a fire alarm if a sensor indicated smoke. If there are N sensors, you would need a rule for each sensor as follows:

```
IF sensor 1 indicates smoke THEN sound fire alarm 1
IF sensor 2 indicates smoke THEN sound fire alarm 2
...
IF sensor N indicates smoke THEN sound fire alarm N
```

However, with unification, a variable called ?N can be used for the sensor identifier so that one rule can be written as follows:

```
IF sensor ?N indicates smoke
THEN sound fire alarm ?N
```

When two complementary literals are unified, they can be eliminated by resolution. For the two preceding clauses, the substitution of *y* by *b* gives:

```
~F(b) ∨ H(a,b)
F(b)
```

The predicate *F* has now been unified and can be resolved into:

```
H(a,b)
```

A substitution is defined as the simultaneous replacement of variables by terms that differentiate from the variables. The terms may be constants, variables, or functions. The substitution of terms for variables is indicated by the set:

$$\{ t_1/v_1,\ t_2/v_2\ \ldots\ t_N/v_N \}$$

If θ is such a set and A an argument, then Aθ is defined as the substitution instance of A. For example, if:

```
θ = { a/x, f(y)/y, x/z }
A = P(x) ∨ Q(y) ∨ R(z)
```

Then:

```
Aθ = P(a) ∨ Q(f(y)) ∨ R(x)
```

Notice that the substitution is simultaneous, so that we get R(x) and not R(a).

Suppose there are two clauses C_1 and C_2 defined as:

$C_1 = \sim P(x)$
$C_2 = P(f(x))$

One possible unification of P is:

$C_1' = C_1 \{ f(x)/x \} = \sim P(f(x))$

Another possible unification for P is the two substitutions using the constant *a:*

$C_1'' = C_1 \{ f(a)/x \} = \sim P(f(a))$
$C_2' = C_2 \{ a/x \} = P(f(a))$

Notice that P(f(x)) is more general than P(f(a)) since there are infinitely many instances of P(f(x)) that can be produced by substituting a constant for x. A clause like C_1 is called the **most general clause**.

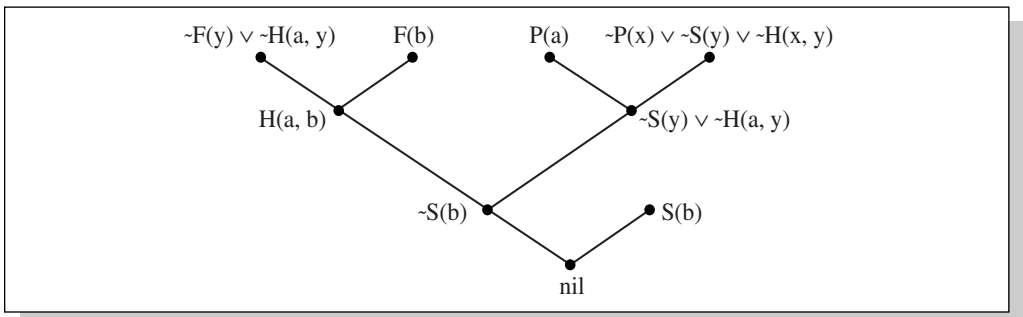
In general, a substitution θ is called a **unifier** for a set of clauses $\{A_1, A_2, \ldots A_n\}$ if and only if $A_1\theta = A_2\theta = \ldots A_n\theta$. A set that has a unifier is called **unifiable**. The **most general unifier** (**mgu**) is that from which all the other unifiers are instances. This can be expressed more formally by stating that θ is the most general unifier if and only if for each unifier, α, of a set, there is a substitution β such that:

$\alpha = \theta\beta$

where $\theta\beta$ is the compound substitution created by first applying θ and then β. The **Unification Algorithm** is a method of finding the most general unifier for a finite unifiable set of arguments.

For our example with clauses (1a) –(3b), the results of substitution and unification are shown in the resolution refutation tree of Figure 3.20. Since the root is nil, the negated conclusion is false and so the conclusion is valid that "no failure is a success."

Figure 3.20 Resolution Refutation Tree to Prove No Failure Is a Success



The examples used so far are very simple and their resolution is straightforward, although tedious for a human. However, in many other situations the resolution process may lead to a dead end and so backtracking is necessary to try alternative clauses for resolution. Although resolution is very powerful and is the basis of PROLOG, it may be inefficient for some problems. One of the problems with resolution is that it has no efficient built-in search strategy and so the programmer must supply heuristics, such as the PROLOG cut, for efficient searching.

A number of modified versions of resolution have been investigated, such as unit preference, input resolution, linear resolution, and set of support. The main advantage of resolution is that it is a single powerful technique that is adequate for many cases. This makes it easier to build a mechanical system such as PROLOG than systems that attempt to implement many different rules of inference. Another advantage of resolution is that when successful, resolution automatically provides a proof by showing the sequence of steps to a nil.

3.14 FORWARD AND BACKWARD CHAINING

A group of multiple inferences that connect a problem with its solution is called a **chain**. A chain that is searched or traversed from a problem to its solution is called a forward chain. Another way of describing forward chaining is reasoning from facts to the conclusions that follow from the facts. A chain that is traversed from a hypothesis back to the facts that support the hypothesis is a backward chain. Another way of describing a backward chain is in terms of a goal that can be accomplished by satisfying subgoals. As this shows, the terminology used to describe forward and backward chaining depends on the problem being discussed.

Chaining can be easily expressed in terms of inference. For example, suppose we have rules of the modus ponens type:

```
p → q
p
∴ q
```

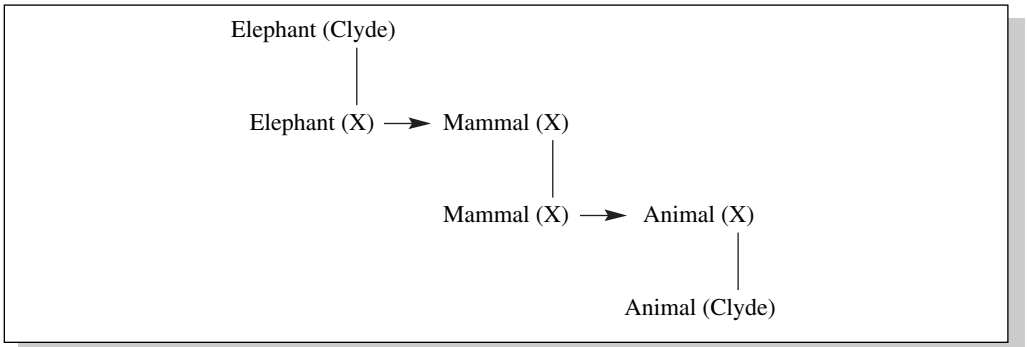
that form a chain of inference, such as the following:

```
elephant(x) → mammal(x)
mammal(x) → animal(x)
```

These rules may be used in a causal chain of forward inference, which deduces that Clyde is an animal given that Clyde is an elephant. The inference chain is illustrated in Figure 3.21. Notice that the same diagram also illustrates backward chaining.

In Figure 3.21 the causal chain is represented by the sequence of **bars**, **|,** connecting the consequent of one rule to the antecedent of the next. A bar also indicates the unification of variables to facts. For example, the variable x in the predicate elephant(x) must first be unified with the fact elephant(Clyde) before

Figure 3.21 Causal Forward Chain



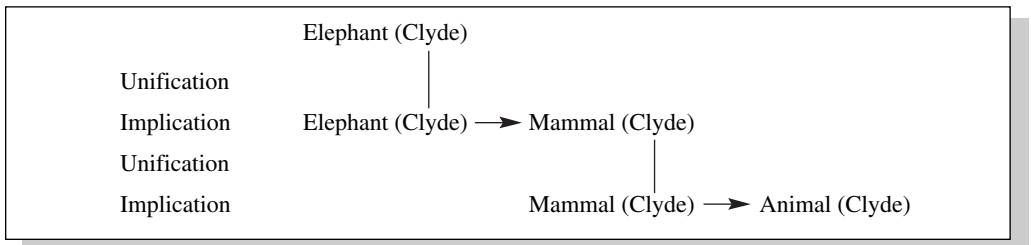
the elephant rule can be applied. The causal chain is really a sequence of implications and unifications, as shown in Figure 3.22.

Backward chaining is the reverse process. Suppose we want to prove the hypothesis animal(Clyde). The central problem of backward chaining is to find a chain linking the evidence to the hypothesis. The fact elephant(Clyde) is called the **evidence** in backward chaining to indicate it is used to support the hypothesis, like the way that evidence in a court is used to prove the guilt or innocence of the defendant.

As a simple example of forward and backward chaining, suppose you are driving and suddenly see a police car with flashing lights and siren. By forward chaining, you may infer that the police want you or someone else to stop. That is, the initial facts support two possible conclusions. If the police car pulls up right in back of you or the police wave at you, a further inference is that they want you rather than someone else. Adopting this as a working hypothesis, you can apply backward chaining to reason why.

Some possible intermediate hypotheses are littering, speeding, malfunctioning equipment, and driving a stolen vehicle. Now you examine the evidence to support these intermediate hypotheses. Was it the beer bottle that you threw out the window, going 100 in a 30 miles per hour speed zone, the broken tail-lights or the license plates identifying the stolen car you're driving? In this case, each

Figure 3.22 Explicit Causal Chain



piece of evidence supports an intermediate hypothesis and so they are all true. Any or all of these intermediate hypotheses are possible reasons to prove the working hypothesis that the police want you.

It's helpful to visualize forward and backward chaining in terms of a path through a problem space in which the intermediate states correspond to intermediate hypotheses under backward chaining or intermediate conclusions under forward chaining. Table 3.14 summarizes some of the common characteristics of forward and backward chaining. Note that the characteristics in this table are meant only as a guide. It is certainly possible to do diagnosis in a forward-chaining system and planning in a backward-chaining one. In particular, explanation is facilitated in backward chaining because the system can easily explain exactly what goal it is trying to satisfy. In forward chaining, explanation is not so easily facilitated because the subgoals are not explicitly known until discovered.

Figure 3.23 illustrates the basic concept of forward chaining in a rule-based system. Rules are triggered by the facts that satisfy their antecedent or left-hand-sides (LHS). For example, the rule R_1 must be satisfied by facts B and C for it to be activated. However, only fact C is present, and so R_1 is not activated. Rule R_2 is activated by facts C and D, which are present and so R_2 produces the intermediate fact H. Other satisfied rules are R_3, R_6, R_7, R_8, and R_9. The execution of rules R_8 and R_9 produce the conclusions of the forward-chaining process. These conclusions may be other facts, output, and so forth.

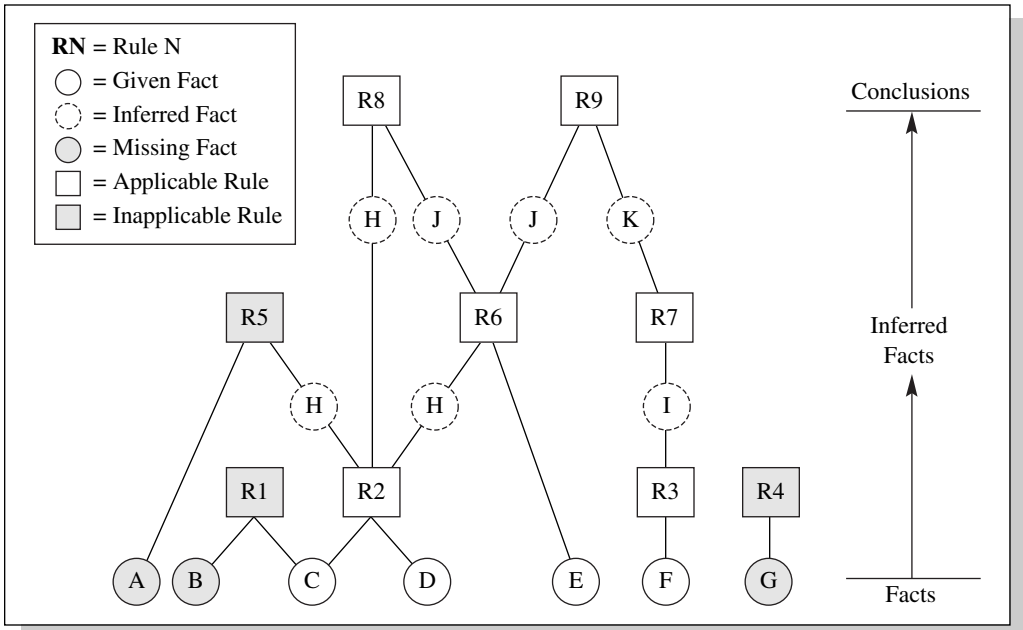
Forward chaining is called **bottom-up reasoning** because it reasons from the low-level evidence, facts, to the top-level conclusions that are based on the facts. Bottom-up reasoning in an expert system is analogous to bottom-up conventional programming, which was discussed in Chapter 1. Facts are the elementary units of the knowledge-based paradigm since they cannot be decomposed into any smaller units that have meaning. For example, the fact "duck" has definite meanings as a noun and as a verb. However, if it is decomposed any further, the result is the letters d, u, c, and k, which have no special meaning. In conventional programs the basic units of meaning are **data**.

By custom, higher-level constructs that are composed of lower-level ones are put at the top. So reasoning from the higher-level constructs such as

Table 3.14 Some Characteristics of Forward and Backward Chaining

Forward Chaining	Backward Chaining
Planning, monitoring, control	Diagnosis
Present to future	Present to past
Antecedent to consequent	Consequent to antecedent
Data driven, bottom-up reasoning	Goal driven, top-down reasoning
Work forward to find what solutions follow from the facts	Work backward to find facts that support the hypothesis
Breadth-first search facilitated	Depth-first search facilitated
Antecedents determine search	Consequents determine search
Explanation not facilitated	Explanation facilitated

Figure 3.23 Forward Chaining



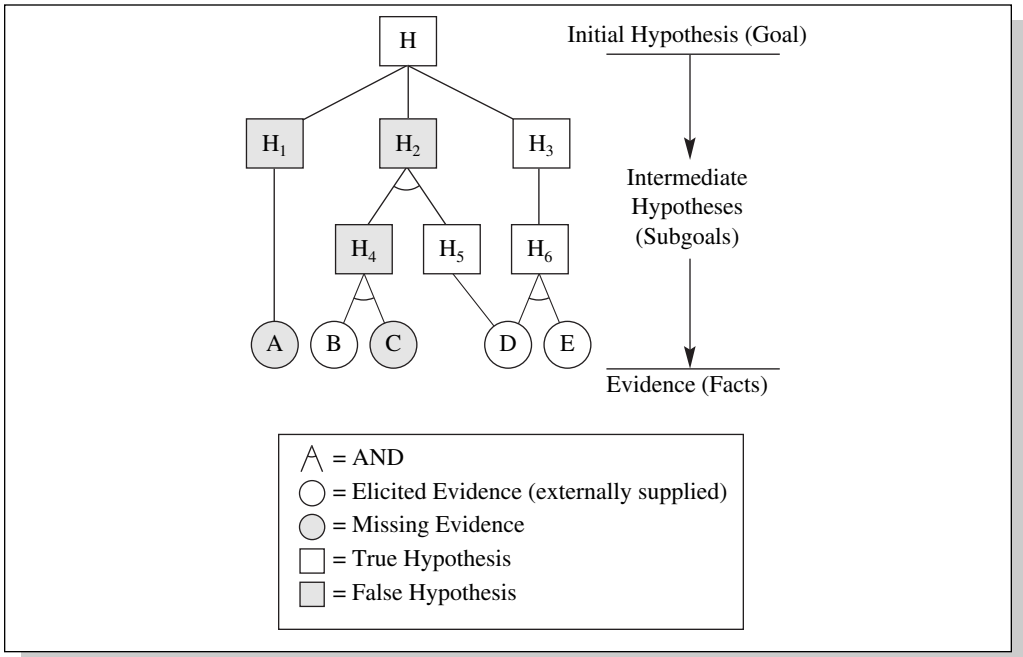
hypotheses down to the lower-level facts that may support the hypotheses is called **top-down reasoning** or backward chaining. Figure 3.24 illustrates the concept of backward chaining. In order to prove or disprove hypotheses H, at least one of the intermediate hypotheses, H_1, H_2, or H_3, must be proven. Notice that this diagram is drawn as an AND-OR tree to indicate that in some cases, such as H_2, all its lower-level hypotheses must be present to support H_2. In other cases, such as the top-level hypotheses, H, only one lower-level hypothesis is necessary. In backward chaining the system will commonly elicit evidence from the user to aid in proving or disproving hypotheses. This contrasts with a forward-chaining system, in which all the relevant facts are usually known in advance.

If you look back at Figure 3.4 in Section 3.2, you will see that the decision net is organized very well for backward chaining. The top-level hypotheses are the different types of raspberries such as flowering raspberry, black raspberry, wineberry, and red raspberry. The evidence to support these hypotheses is lower down. The rules to identify a raspberry can be easily written. For example,

```
IF the leaves are simple THEN flowering raspberry
```

One important aspect of eliciting evidence is asking the right questions. The right questions are those that improve the efficiency in determining the correct answer. One obvious requirement for this is that the expert system should ask questions that deal only with the hypotheses that it is trying to prove. While there may be hundreds or thousands of questions that the system could ask about, there is a cost in time and money to obtain the evidence to answer the

Figure 3.24 Backward Chaining



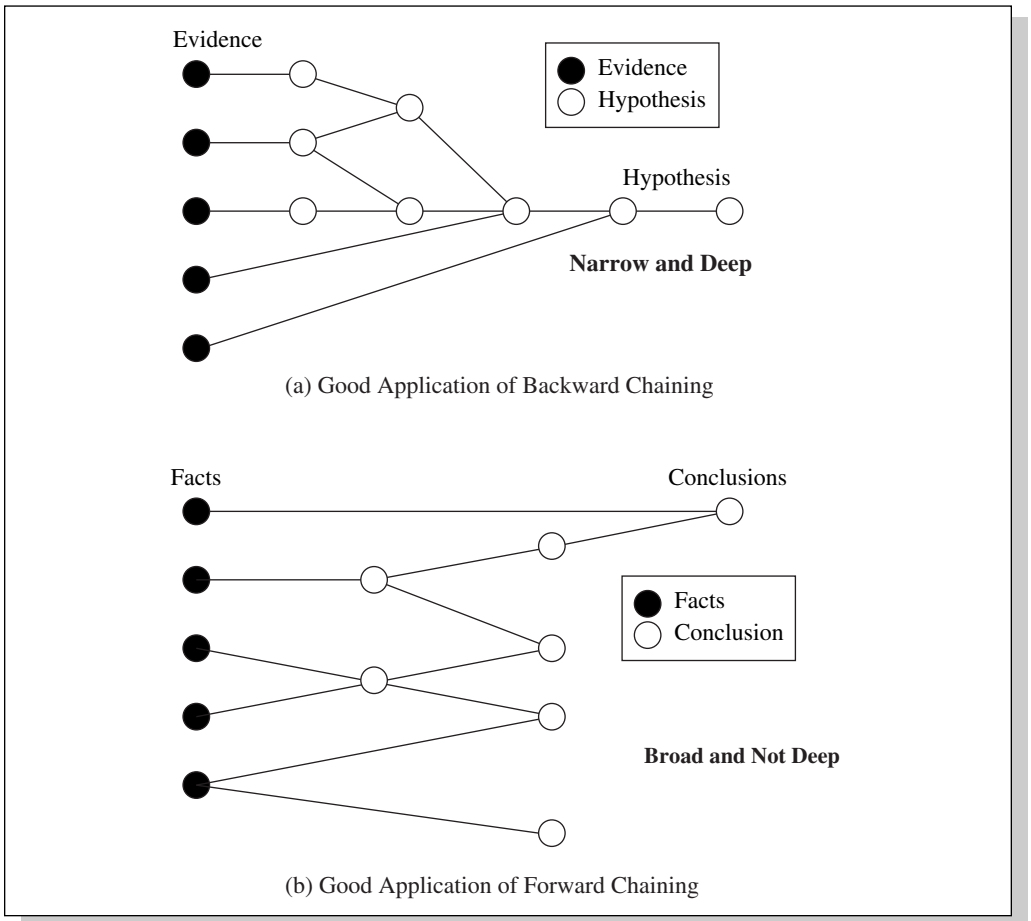
questions. Also, accumulating certain types of evidence such as medical test results may be uncomfortable and possibly hazardous to the patient (in fact, it's difficult to think of a pleasurable medical test).

Ideally the expert system should also allow the user to volunteer evidence even if the system has not asked for it. Allowing the user to volunteer evidence speeds up the backward-chaining process and makes the system more convenient for the user. The volunteered evidence may let the system skip some links in the causal chain or pursue a completely new approach. The disadvantage is that more complex programming of the expert system is involved since the system may not follow a chain link by link.

Good applications for forward and backward chaining are shown in Figure 3.25. For simplicity these diagrams are drawn as trees instead of a general network. A good application for forward chaining occurs if the tree is wide and not very deep. This is because forward chaining facilitates a breadth-first search. That is, forward chaining is good if the search for conclusions proceeds level by level. In contrast, backward chaining facilitates a depth-first search. A good tree for depth-first search is narrow and deep.

Notice that the structure of the rules determines the search for a solution. That is, the activation of a rule depends on the patterns that the rule is designed to match. The patterns on the LHS determine whether a rule can become activated by facts. The actions on the RHS determine the facts that are asserted and deleted and so affect other rules. An analogous situation exists for backward chaining except that hypotheses are used instead of rules. Of course, an intermediate

Figure 3.25 Forward and Backward Chaining



hypothesis may be simply a rule that is matched on its consequent instead of its antecedent.

As a very simple example, consider the following IF-THEN type rules:

```
IF A THEN B
IF B THEN C
IF C THEN D
```

If fact A is given and the inference engine is designed to match facts against antecedents, then the intermediate facts B and C will be asserted and conclusion D. This process corresponds to forward chaining. The inference engine is the heart of an expert system.

In contrast, if the fact (actually hypothesis) D is asserted and the inference engine matches facts against consequents, the result corresponds to backward

chaining. In systems designed for backward chaining, such as PROLOG, the backward-chaining mechanism includes a number of features, such as automatic backtracking, to facilitate backward chaining.

Backward chaining can be accomplished in a forward-chaining system and vice versa by redesign of the rules. For example, the previous rules for forward chaining can be rewritten as:

```
IF D THEN C
IF C THEN B
IF B THEN A
```

Now C and B are considered as subgoals or intermediate hypotheses that must be satisfied to satisfy the hypothesis D. The evidence A is the fact that indicates the end of the generation of subgoals. If there is a fact A, then D is supported and considered true under this chain of backward inference. If there is no A, then the hypothesis D is unsupported and considered false.

One difficulty with this approach is efficiency. A backward-chaining system facilitates depth-first search while a forward-chaining system facilitates breadth-first search. Although you can write a backward-chaining application in a forward-chaining system and vice versa, the system will not be as efficient in its search for a solution. The second difficulty is a conceptual one. The knowledge elicited from the expert will have to be altered to meet the demands of the inference engine. For example, a forward-chaining inference engine matches the antecedent of rules while a backward-chaining one matches the consequent. That is, if the expert's knowledge is naturally backward chaining, it will have to be totally restructured to cast it in a forward-chaining mode, and vice versa.

3.15 OTHER METHODS OF INFERENCE

A number of other types of inference are sometimes used with expert systems. While these methods are not as general purpose as deduction, they are very useful.

Analogy

Besides deduction and induction, another powerful inference method is **analogy**. The basic idea of reasoning by analogy is to try and relate old situations as a guide to new ones. All living creatures are very good at applying analogical reasoning in their lives, which is essential because of the tremendous number of new situations that are encountered in the real world. Rather than treating every new situation as unique, it's often helpful to try and see the similarities of the new situation to old ones that we know how to deal with. Analogical reasoning is related to induction. While induction makes inferences from the specific to the general of the same situation, analogy tries to make inferences from situations that are not the same. Analogy cannot make formal proofs like deduction. Instead, analogy is a heuristic reasoning tool that may sometimes work and is the primary tool of case-based reasoning in legal arguments and medical diagnosis.

An example of reasoning by analogy is medical diagnosis. When you see a doctor because of a medical problem, the doctor will elicit information from you and note the symptoms of the problem. If your symptoms are identical or strongly similar to those of other people with Problem X, the doctor may infer by analogy that you have Problem X. Analogy is cheap reasoning.

Notice that this diagnosis is not a deduction, because you are unique. Just because someone else with the problem shows certain symptoms doesn't mean that you will exhibit the same symptoms. Instead, the doctor assumes that your symptoms make you analogous to a person with the same symptoms and known problem. This initial diagnosis is a hypothesis that medical tests may either prove or disprove. It's important to have an initial working hypothesis because it narrows down the thousands of potential problems. It would be prohibitively expensive and time consuming to just start giving you every possible test without an initial hypothesis.

As an example of the utility of reasoning by analogy, suppose two people play a game called the 15 Game. They take turns in picking a number from 1 to 9 with the constraint that the same number cannot be used twice. The first person whose numbers add up to 15 wins. Although at first this seems like a strange game that requires some thinking, an analogy can make it very easy to play.

Consider the tic-tac-toe board below with values assigned to each cell as shown:

6	1	8
7	5	3
2	9	4

This is a **magic square** because the sum of the values in the rows, columns, and diagonals is a constant. The tic-tac-toe board with the magic square values can be considered an analogy to the 15 Game. Playing the 15 Game is now very easy if you think in terms of tic-tac-toe and then translate the winning strategy to the 15 Game.

This particular magic square is called the **standard square of order 3**. The term *order* refers to the number of rows or columns of a square. There is only one unique square of order 3. Other magic squares can be created by rotating or reflecting the standard square. Another way to construct magic squares is by adding the same constant to each cell. Knowing this information allows us to deduce the winning strategies for the 18 Game, where the numbers must be selected from the set:

```
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

or the 21 Game, using the set:

```
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

We can now use induction to infer the winning strategy for the 15 + 3N Game, where N is any natural number 1, 2, 3, ... by thinking of the game as analogous to a tic-tac-toe board, which is analogous to a magic square of values:

```
{1+N, 2+N, 3+N, 4+N, 5+N, 6+N, 7+N, 8+N, 9+N}
```

Using the analogy of the order 3 square for games of three moves, by induction we can infer that magic squares of higher order can be used for games involving more than three moves. For example, the magic square of order 4:

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

allows you to play the winning strategy of the four-move, 34 game, by thinking of it in terms of tic-tac-toe. In contrast to the one standard order 3 square, there are 880 standard squares of order 4, which allows considerably more games.

Reasoning by analogy is an important part of commonsense reasoning, which is very difficult for computers (and children). Other applications of analogy have been to learning.

Generate-and-Test

Another method of inference is the classic AI strategy of **generate-and-test**, sometimes called generation and test, which involves the generation of a likely solution and then testing it to see if the proposed solution meets all the requirements. If the solution is satisfactory, then quit, else generate a new solution, test again, and so forth. This method was used in the first expert system, DENDRAL, conceived in 1965, to aid in identifying organic molecular structures. Data from an unknown sample are supplied by a mass spectrometer and input to DENDRAL, which generates all the potential molecular structures that could produce the unknown spectrogram. DENDRAL then tests the most likely candidate molecules by simulating their mass spectrograms and comparing the results to the original unknown. Another program that uses generate-and-test is the Artificial Mathematician program to infer new mathematical concepts.

In order to reduce the enormous number of potential solutions, generate-and-test is normally used with a planning program to limit the likely potential solutions for generation. This variation is called **plan-generate-test** and is used for efficiency in many systems. For example, the medical diagnosis expert system MYCIN also has the capability of planning a therapeutic drug treatment after a patient's disease has been diagnosed. A **plan** is essentially finding chains of rules or inferences that connect a problem with a solution, or goal, with the evidence to support it. Planning is most efficiently done by simultaneously searching forward from the facts and backward from the goal.

The MYCIN planner first creates a prioritized list of therapeutic drugs to which the patient is sensitive. In order to reduce undesirable drug interactions,

it's best to limit the number of drugs that a patient receives, even if the patient is thought to be suffering from several different infections. The generator takes the prioritized list from the planner and generates sublists of one or two drugs if possible. These sublists are then tested for efficacy against the infections, patient's allergies, and other considerations before a decision is made to administer them to the patient.

Generate-and-test can also be considered the basic inference paradigm of rules. If the conditional elements of a rule are satisfied, it generates some actions such as new facts. The inference engine tests these facts against the conditional elements of rules in the knowledge base. Those rules that are satisfied are put on the agenda and the top-priority rule generates its actions, which are then tested, and so forth. Thus generate-and-test produces an inference chain that may lead to the correct solution.

Abduction

Inference by abduction is another method that is commonly used in diagnostic problem solving. The schema of abduction resembles modus ponens, but is actually very different, as shown in Table 3.15.

Table 3.15 Comparison of Abduction and Modus Ponens

Abduction	Modus ponens
$p \rightarrow q$	$p \rightarrow q$
q	p
$\therefore p$	$\therefore q$

Abduction is another name for a fallacious argument that we discussed in Section 3.6, as the Fallacy of the Converse. Although abduction is not a valid deductive argument, it is a useful method of inference and has been used in expert systems. Like analogy, which also is not a valid deductive argument but is quick and cheap (in reasoning time, not money). Abduction may be useful as a heuristic rule of inference. That is, when we have no deductive method of inference, abduction may prove useful but is not guaranteed to work. Analogy, generate-and-test, and abduction are all methods that are not deductive and are not guaranteed to work all the time. From true premises, these methods cannot prove true conclusions. However, these techniques are useful in reducing the search space by generating reasonable hypotheses, which can then be used with deduction.

Abduction is sometimes referred to as *reasoning from observed facts to the best explanation*. As an example of abduction, consider the following:

```
IF x is an elephant THEN x is an animal
IF x is an animal THEN x is a mammal
```

If we know that Clyde is a mammal, can we conclude that Clyde is an elephant?

The answer to this question depends on whether we are talking about the real world or our expert system. In the real world we could not make this conclusion with any degree of certainty. Clyde could be a dog, cat, cow, or any other kind of animal that was a mammal but not an elephant. In fact, considering how many species of animals there are, the probability of Clyde being an elephant is rather low, without knowing any more information about Clyde.

However, in an expert system with only the preceding rules, we could say by abduction with 100% certainty that if Clyde is a mammal, then Clyde is an elephant. This inference follows from a closed-world-assumption in which we have assumed that nothing else exists outside the closed world of our expert system. Any-thing that cannot be proven is assumed false in a closed world. Under the closed-world assumption, all the possibilities are known. Since the expert system consists of only these two rules and only an elephant can be a mammal, then if Clyde is a mammal, he must be an elephant.

Suppose we add a third rule, as follows:

```
IF x is a dog THEN X is an animal
```

We can still operate our expert system under the closed-world assumption. However, now we cannot conclude with 100 percent certainty that Clyde is an elephant. All we can be sure of is that Clyde is either an elephant or a dog.

In order to decide between the two, more information is needed. For example, if there is another rule,

```
IF x is a dog THEN x barks
```

and evidence that Clyde barks, the rules can be revised as follows:

```
(1) IF x is an animal THEN x is a mammal
(2) IF x barks THEN x is an animal
(3) IF x is a dog THEN x barks
(4) IF x is an elephant THEN x is an animal
```

Now a backward chain of abductive inference using rules (1), (2), and (3) can be made to show that Clyde must be a dog.

A backward chain of abduction is not the same as the customary meaning of backward chaining. The term backward chaining means that we are trying to prove a hypothesis by looking for evidence to support it. Backward chaining would be used in trying to prove that Clyde is a mammal. Of course, for our small system there are no other possibilities. However, other classifications could be added for reptiles, birds, and so forth.

If it's known that Clyde is a mammal, abduction could be used to determine if Clyde is an elephant or a dog. Forward chaining would be used if it is known that Clyde is an elephant and we wanted to know if he is a mammal. As you can see, the choice of inference method depends on what is to be determined. Since forward chaining is deductive, only its conclusions are always guaranteed valid. Table 3.16 summarizes the purpose of each of the three inference techniques.

Table 3.16 Summary of the Purpose of Forward Chaining, Backward Chaining, and Abduction

Inference	Start	Purpose
Forward chaining	Facts	Conclusions that must follow
Backward chaining	Uncertain conclusion	Facts to support the conclusion
Abduction	True conclusion	Facts which may follow

A number of AI and expert systems have used frame-based abduction for diagnostic problem solving. In these systems, the knowledge base contains **causal associations** between disorders and symptoms. Inferences are made using generate-and-test of hypotheses for the disorders.

Nonmonotonic Reasoning

Normally the addition of new axioms to a logic system means that more theorems can be proven since there are more axioms from which theorems can be derived. This property of increasing theorems with increasing axioms is known as **monotonicity** and systems such as deductive logic are called **monotonic systems**.

However, a problem can occur if a newly introduced axiom partially or completely contradicts a previous axiom. In this case, the theorems that had been proven may no longer be valid. Thus, in a **nonmonotonic system**, the theorems do not necessarily increase as the number of axioms increases.

The concept of nonmonotonicity has an important application to expert systems. As new facts are generated—which is analogous to theorems being proved—a monotonic expert system would keep building up facts. A major problem can occur if one or more facts becomes false because a monotonic system cannot deal with changes in the truth of axioms and theorems. As a very simple example, suppose there is a fact that asserts the time. As soon as time changes by one second, the old fact is no longer valid. A monotonic system would not be capable of dealing with this situation. As another example, suppose a fact was asserted by an aircraft identification system that a target was hostile. Later, new evidence proves the target is friendly. In a monotonic system, the original identification of hostile could not be changed. A nonmonotonic system allows the retraction of facts.

As another application, suppose you wanted to write an explanation facility for an expert system that would allow a user to go back to previous inferences and explore alternate inference paths of "What if" type questions. All inferences made after the desired previous one must be retracted from the system. Besides facts, rules may also have been excised from the system and so must be put back in the knowledge base for nonmonotonicity. Further complications arise in systems such as OPS5 in which rules can be created automatically on the right-hand side of rules during execution. For nonmonotonicity, any inferred rules made after the previous desired inference would have to be excised from the system. Keeping track of all inferences made can consume a lot of memory and significantly slow down the system.

In order to provide for nonmonotonicity, it is necessary to attach a justification or dependency to each fact and rule that explains the reason for belief in it. If a nonmonotonic decision is made, then the inference engine can examine the justification of each fact and rule to see if it is still believed, and also to possibly restore excised rules and retracted facts that are believed again.

The problem of justifying facts was first pointed out in the **frame problem**, which is not the same concept as the frames discussed in Chapter 2. The frame problem is a descriptive term named after the problem of identifying what has or has not changed in a movie frame. Motion pictures are photographed as a succession of still pictures, called frames. When played back at 24 frames per second or faster, the human eye cannot distinguish the individual frames and so the illusion of motion results. An AI frame problem is to recognize changes in an environment over time. As an example, consider the Monkey and Bananas problem discussed earlier. Suppose the monkey must step on a red box to reach the bananas and so the action is "push red box under bananas." Now the frame problem is—how do we know the box is still red after the action? Pushing the box should not change the environment. However, other actions such as "paint the box blue" will change the environment. In some expert-system tools an environment is called a **world** and consists of a set of related facts. An expert system may keep track of multiple worlds to do hypothetical reasoning simultaneously. The problem of maintaining the correctness or truth of a system is called **truth maintenance**. Truth maintenance or a variation called **assumption-based truth maintenance,** is essential for keeping each world pure by retracting unjustified facts.

As a simple example of nonmonotonic reasoning, let's consider the classic example of Tweety the bird. In the absence of any other information, we would therefore assume that since Tweety is a bird, then Tweety can fly. This is an example of **default reasoning**, much like the defaults used in frame slots. Default reasoning can be considered as a rule that makes inferences about rules, a **metarule** which states:

```
IF X is not known for certain, and
      there is no evidence contradicting X
THEN tentatively conclude Y
```

Metarules are discussed more extensively in the next section.

In our case the metarule has the more specific form:

```
X is the rule "All birds can fly," and
      fact "Tweety is a bird"
Y is the inference "Tweety can fly"
```

In terms of production rules, this can be expressed as a rule in our knowledge base, which says:

```
IF X is a bird THEN X can fly
```

and the fact that exists in working memory:

```
Tweety is a bird
```

Unifying the fact with the antecedent of the rule gives the inference that Tweety can fly.

Now comes the problem. Suppose an additional fact is added to working memory, which says that Tweety is a penguin. We know that penguins cannot fly and so the inference that Tweety can fly is incorrect. Of course, there must be a rule in the system that also states this knowledge or else the fact will be ignored.

In order to maintain the correctness of our system, the incorrect inference must be removed. However, this may not be sufficient if other inferences were based on the incorrect inference. That is, other rules may have used the incorrect inference as evidence to draw additional inferences, and so on. This is a problem of truth maintenance. The inference that Tweety can fly was a **plausible inference** that is based on default reasoning. The term *plausible* means not impossible and will be discussed further in Chapter 4.

One way of allowing for nonmonotonic inference is by defining a sentential operator M, which can be informally defined as "is consistent." For example,

$$(\forall\ x)\ [Bird(x) \wedge M(Can\_fly(x)) \rightarrow Can\_fly(x)]$$

can be stated as "For every x, if x is a bird and it is consistent that a bird can fly, then x can fly." A more informal way of stating this is "Most birds can fly." The term *is consistent* means that there are no contradictions with other knowledge. However, this interpretation has been criticized as really stating that the only birds that cannot fly are those which have been inferred not capable of flying. This is an example of **autoepistemic reasoning**, which literally means reasoning about your own knowledge. Both default and autoepistemic reasoning are used in **commonsense reasoning**, which humans generally do quite well but is very difficult for computers.

Autoepistemic reasoning is reasoning about your own knowledge as distinct from knowledge in general. Generally, you can do this very well because you know the limits of your knowledge. For example, suppose a total stranger came up and stated you were their spouse. You would immediately know (unless you had amnesia) that you were not their spouse because you had no knowledge of that stranger. The general metarule of autoepistemic reasoning is:

```
IF I have no knowledge of X
THEN X is false
```

Notice how autoepistemic reasoning relies on the closed-world assumption. Any fact that is not known is assumed false. In autoepistemic reasoning, the closed world is your self-knowledge.

Autoepistemic and default reasoning are both nonmonotonic. However, the reasons are different. Default reasoning is nonmonotonic because it is **defeasible**. The term defeasible means that any inferences are tentative and may have to be withdrawn as new information becomes available. However, pure autoepistemic reasoning is not defeasible because of the closed-world assumption that declares all true knowledge is already known to you. For example, since married people know who is their spouse (unless they want to forget), a married

person would not accept that a total stranger was their spouse if they were told that. Because most people recognize that they have imperfect memories, they do not adhere to pure autoepistemic reasoning. Of course, computers do not have this problem until their hard disk crashes.

Autoepistemic reasoning is nonmonotonic because the meaning of an autoepistemic statement is **context-sensitive**. The term *context-sensitive* means that the meaning changes with the context. As a simple example of context-sensitivity, consider how the word "read" is pronounced in the two sentences:

```
I have read the book
I will read the book
```

The pronunciation of "read" is context sensitive.

Now consider a system consisting of the following two axioms:

```
(∀ x) [Bird(x) ∧ M(Can_fly(x))→ Can_fly(x)]
Bird(Tweety)
```

In this logic system, Can_fly(Tweety) is a theorem derived by unification of Tweety with the variable x and implication.

Now suppose a new axiom is added that states that Tweety cannot fly and thus contradicts the previously derived theorem:

```
~Can_fly(Tweety)
```

The M-operator must change its operation on its argument because now M(Can_fly(Tweety)) is not consistent with the new axiom. In this new context of three axioms, the M-operator would not give a TRUE result for Can_fly(Tweety) because it conflicts with the new axiom. The returned value by the M-operator must be FALSE in this new context, and so the conjunction is FALSE. Thus, there is no implication to produce the theorem Can_fly(Tweety) and no conflict.

One way of implementing this by rules is as follows:

```
IF x is a bird AND x is typical
THEN x can fly
IF x is a bird AND x is nontypical
THEN x cannot fly

Tweety is a bird
Tweety is nontypical
```

Notice that this system does not invalidate the conclusion Can_fly(Tweety), but rather prevents the incorrect rule from firing at all. This is a much more efficient method of truth maintenance than if we had one rule and the special axiom ~Can_fly(Tweety). Now we have a more general system that can easily handle other nonflying birds such as ostriches without our having to continually add new inferences, which is what the system is supposed to do.

3.16 METAKNOWLEDGE

The Classic Meta-DENDRAL program used induction to infer new rules of chemical structure. Meta-DENDRAL was an attempt to overcome the knowledge bottleneck of trying to extract molecular structural rules from human experts. Some expert system tools come with progress to induce rules.

For example, the following classic metarule is taken from the TEIRESIAS knowledge acquisition program for MYCIN, the expert system to diagnose blood infections and meningitis:

```
METARULE 2
IF
The patient is a compromised host, and
There are rules which mention in their premise
   pseudomonas, and
There are rules which mention in their premise
   klebsiellas
THEN
There is suggestive evidence (0.4) that the former
   should be done before the latter
```

The number 0.4 in the action of the rule is a degree of certainty and will be discussed in a later chapter.

TEIRESIAS acquires knowledge interactively from an expert. If a wrong diagnosis has been made by MYCIN, then TEIRESIAS will lead the expert back through the chain of incorrect reasoning until the expert states where the incorrect reasoning started. While going back through the reasoning chain, TEIRESIAS will also interact with the expert to modify incorrect rules or acquire new rules.

Knowledge about new rules is not immediately put into MYCIN. Instead, TEIRESIAS checks to see if the new rule is compatible with similar rules. For example, if the new rule describes how an infection enters the body and other accepted rules have a conditional element stating the portal of entry to the body, then the new rule should also. If the new rule does not state the portal of entry, then TEIRESIAS will query the user about this discrepancy. TEIRESIAS has a **rule model pattern** of similar rules that it knows about, and tries to fit the new rule into its rule model. In other words, the rule model is the knowledge that TEIRESIAS has about its knowledge. An analogous situation for a human would occur if you went to a car dealer to buy a new car and the dealer tried to sell you a car with three wheels.

The metaknowledge of TEIRESIAS is of two types. The METARULE 2, described previously, is a control strategy that tells how rules are to be applied. In contrast, the rule model type of metaknowledge determines if the new rule is in the appropriate form to be entered in the knowledge base. In a rule-based expert system, determining if the new rule is in the correct form is called **verification** of the rule. Determining that a chain of correct inferences leads to the correct answer is called **validation**. Validation and verification are so interdependent

that the acronym **V&V** is commonly used to refer to both. A more colloquial definition of the terms from software engineering:

```
Verification: "Am I building the product right?"
Validation: "Am I building the right product?"
```

Basically, verification has to do with internal correctness while validation has to do with external correctness. V&V will be discussed in more detail in Chapter 6.

3.17 HIDDEN MARKOV MODELS

As a modern example of metaknowledge, consider path planning for a robot. If the robot does not have a global positioning system (GPS), or it is not precise enough, which is usually the case for commercial systems that can only determine a location to within 3 meters, other means must be used. A good application to path planning uses a **Markov decision process** (**MDP**) (Kaelbling 98). Other methods use the classic A* algorithm, Kalman Filters, and other techniques (Lakemeyer 03).

In the real world there are always uncertainties and pure logic is not a good guide when there is uncertainty. A MDP is more realistic in the case of partial or hidden information about the state and parameters, and the need for planning. Such processes are not restricted to physical path planning but cover any type of planning in a partially known environment such as oil exploration, transportation logistics, and factory process control where sensors may malfunction and things go wrong. Such process control may be especially important when the process being controlled is a nuclear power plant.

The ultimate example of partial information occurs when a robot is exploring the surface of another planet, such as Mars. If the goal is to get to a certain rock but the path is partially obscured, the robot must try to make optimum decisions on how to reach the goal while subject to constraints such as its limited energy supply and perhaps an inability to retrace the correct path once it has gone into a steep crater.

A MDP can be defined as the tuple {States, Actions, Transitions, Rewards}. More formally we can write MDP≡ {S, A, T, R} where

```
S is a set of states of the environment;
A is a set of actions;
```

T: S x A ≡∏(S); where ∏ is called the **state-transition function** which determines what state a particular action will determine. The "x" is the Cartesian product operator. ∏ is the set of all possible states and actions. Naturally some may not be feasible or desirable but the Cartesian operation does not filter out which are optimum. In order to determine which actions lead to better states, e.g., the robot not stuck in a dead end, a reward is necessary.

R: S x A→R(S) where **R** is the **reward function** which provides the immediate reward gained by the **agent** for taking a certain action. The term agent is used for any entity that acts on behalf of another. Thus the robot is a human agent on another planet. Different states will give different expected rewards. In

search for the optimum path to the goal, one way of defining the optimum is by maximizing the expected sum of the rewards. The immediate reward is given right away but the sum of rewards determines whether the agent will succeed or fail in reaching the goal. For example, if the robot takes a right turn it may find a clear path. Unfortunately this clear path may not lead to the goal of getting to the desired rock. This is like the verification discussed earlier while validation in this case means the robot has gone to the correct rock. In the next chapter we will see more examples of rewards in terms of the utility functions and Bayes Theorem for dealing with uncertainty.

In (1) the case of the robot not sure of its location, an unknown state, or (2) which path to take, an unknown parameter, the Hidden Markov Model (HMM) is very useful. Software for HMM is available such as the HTK toolkit (http://htk.eng.cam.ac.uk/). This has been successfully used for speech recognition, speech synthesis, character recognition, and DNA sequencing. Once the biological structure is determined by finding the sequence of its genetic material, it can be synthesized or modified to identify and treat disease. Other software is HMMER, a freely distributable implementation of HMM for protein sequence analysis (http://hmmer.wustl.edu/).

The popular Matlab has a toolkit available for HMM which supports various types of scientific inference especially suited for signal analysis (http://www.ai.mit.edu/~murphyk/Software/HMM/hmm.html). The classic case is to determine the phonemes and thus the words from speech when the user speaks into a microphone to generate an acoustic signal for computer analysis. While broken speech or that enunciated slowly has been solved by many techniques such as ANS, the hard problem is continuous speech recognition by any speaker without training the system (independent speaker recognition.)

Path planning is also essential to many video games where a character has to get from one location to another with walls and obstacles in the way. The popular A* algorithm is often used.

3.18 SUMMARY

In this chapter the commonly used methods of inference for expert systems have been discussed. Inference is particularly important in expert systems because it is the technique by which expert systems solve problems. The application of trees, graphs, and lattices to the representation of knowledge was discussed. The advantages of these structures to inferences were illustrated.

Deductive logic was covered starting with simple syllogistic logic. Next, the propositional and first-order predicate logic was discussed. Truth tables and rules of inference were described as ways of proving theorems and statements. The characteristics of logical systems such as completeness, soundness, and decidability were mentioned.

The resolution method of proving theorems was discussed for propositional logic and first-order predicate logic. The nine steps involved in converting a well-formed formula to clausal form were illustrated by an example. Skolemization, the prenex-normal form, and unification were all discussed in the context of converting a wff to clausal form.

Another powerful method of inferences, analogy, was discussed. Although not widely used in expert systems because of the difficulty in implementing it, analogy is the normal method of doctors and lawyers, and should be considered in the design of expert systems. Generate-and-test was also discussed with an example of its use in MYCIN. The application of metaknowledge in TEIRESAS was described and its relationship to verification and validation of expert systems.

PROBLEMS

3.1 Write a decision tree program that is self-learning. Teach it the animal knowledge of Figure 3.3.

3.2 Write a program that will automatically translate the knowledge stored in a binary decision tree into IF-THEN type rules. Test it with the animal decision tree of Problem 3.1.

3.3 Draw a semantic net containing the raspberries knowledge of Figure 3.4.

3.4 Draw the state diagram showing a solution to the classic problem of the farmer, fox, goat, and cabbage. In this problem a farmer, fox, goat, and cabbage are on one bank of a river. A boat must be used to transport them to the other side. However, the boat can be used to hold only two at a time (and only the farmer can row). If the fox is left with the goat and the farmer is not present, then the fox will eat the goat. If the goat is left alone with the cabbage, the goat will eat the cabbage.

3.5 Draw a state diagram for a well-structured travel problem having:

(a) three methods of payment: cash, check, or charge
(b) traveler's interests: sun, snow
(c) four possible destinations depending on the traveler's interests and money
(d) three types of transportation

Write IF-THEN rules to advise a traveler where to go depending on money and interests. Pick real destinations and find out the costs to get there from your location.

3.6 Determine whether the following are valid or invalid arguments:

(a) $p \rightarrow q, \sim q \rightarrow r, r; \therefore p$
(b) $\sim p \vee q, p \rightarrow (r \wedge s), s \rightarrow q; \therefore q \vee r$
(c) $p \rightarrow (q \rightarrow r), q; \therefore p \rightarrow r$

3.7 Using the Venn diagram decision procedure, determine if the following are valid or invalid syllogisms:

(a) AEE-4
(b) AOO-1
(c) OAO-3
(d) AAI-1
(e) OAI-2

3.8 Prove whether the following are fallacies or rules of inference. Give an example of each.

(a) Complex Constructive Dilemma

$p \rightarrow q$
$r \rightarrow s$
$p \vee r$
$\therefore q \vee s$

(b) Complex Destructive Dilemma

$p \rightarrow q$
$r \rightarrow s$
$\sim q \vee \sim s$
$\therefore \sim p \vee \sim r$

(c) Simple Destructive Dilemma

$p \rightarrow q$
$p \rightarrow r$
$\sim q \vee \sim r$
$\therefore \sim p$

(d) Inverse

$p \rightarrow q$
$\sim p$
$\therefore \sim q$

3.9 Draw the conclusion from the following premises, taken from Lewis Carroll, author of the famous *Alice's Adventures in Wonderland*:

(a) All the dated letters in this room are written on blue paper.
(b) None of them are in black ink, except those that are written in the third person.
(c) I have not filed any of them that I can read.
(d) None of them, that are written on one sheet, are undated.
(e) All of them, that are not crossed, are in black ink.
(f) All of them, written by Brown, begin with "Dear Sir,".
(g) All of them, written on blue paper, are filed.
(h) None of them, written on more than one sheet, are crossed.
(i) None of them, that begin with "Dear Sir," are written in the third person.

Hint: Use the law of contrapositives to define the following:

A = beginning with "Dear Sir"	B = crossed	C = dated
D = filed	E = in black ink	F = in third person
G = letters that I can read	H = on blue paper	I = on one sheet
J = written by Brown		

3.10 Give a formal proof using predicate logic for the syllogism:
No software is guaranteed
All programs are software
∴ No programs are guaranteed

3.11 Use the following to write quantified first-order predicate logic formulas:

P(x) = x is a programmer

S(x) = x is smart

L(x,y) = x loves y

(a) All programmers are smart.

(b) Some programmers are smart.

(c) No programmers are smart.

(d) Someone is not a programmer.

(e) Not everyone is a programmer.

(f) Everyone is not a programmer.

(g) Everyone is a programmer.

(h) Some programmers are not smart.

(i) There are programmers.

(j) Everyone loves someone.

Hint: Use Appendix B.

3.12 Consider the predicate logic argument:

A horse is an animal

Therefore, the head of a horse is the head of an animal. Define the following:

H(x,y) = x is the head of y

A(x) = x is an animal

S(x) = x is a horse

The premise and conclusion are as follows:

$(\forall x)\ (S(x) \rightarrow A(x))$

$$(\forall x)\ \{[(\exists y)\ (S(y) \wedge H(x,y))] \rightarrow [(\exists z)\ (A(z) \wedge H(x,z))]\}$$

Prove the conclusion using resolution refutation. Show all nine steps in clausal conversion for the conclusion.

3.13 (a) Obtain information from your bank or savings & loan on the criteria for obtaining a car loan. Write a backward-chaining system of rules to determine if an applicant should get a car loan. Be as specific as possible.

(b) Obtain information on qualifying for a home mortgage. Give the modifications of your car loan program to handle home loans as well as car loans.

(c) Obtain information on qualifying for a business loan. Give the modifications to the car-mortgage program to handle business loans as well.

3.14 Write a production system of causal rules to simulate the operation of the fuel system of your car. You may obtain this information from the repair guide for your car. Be as detailed as possible.

3.15 Using the repair guide for your car, write a production system that can diagnose and remedy electrical problems for your car, given symptoms of problems.

3.16 If the conclusion is false and you wanted to determine the facts that followed, would you use abduction or another method of inference? Explain.

3.17 Write IF-THEN rules to identify raspberries using the portion of the decision tree shown in Figure 3.4.

3.18 A parent and two children are on the left bank of a river and wish to cross to the right side. The parent weighs 200 lb and each child weighs 100 lb. One boat is available with a capacity of 200 lb. At least one person is required to row the boat. Draw the decision tree showing all the possibilities for them crossing the river and indicate the successful path through the tree so that they all cross. Assume the following operators in labeling your tree:

SL–Single child crosses from left to right
SR–Single child crosses from right to left
BL–Both children cross from left to right
BR–Both children cross from right to left
PL–Parent crosses from left to right
PR–Parent crosses from right to left

3.19 (a) Draw a logic diagram version of the AND-exclusive-OR car decision of Figure 3.11. Note that you will have to implement the exclusive-OR using standard AND, OR, and NOT elements.

(b) Write forward chaining IF-THEN rules to determine which hypothesis to follow, i.e., sell or repair.

3.20 What type of inference (if any) do the following statements represent?

(a) "The reason I keep insisting that there was a relationship between Iraq and Saddam and al Qaeda is because there was a relationship between Iraq and al Qaeda."

(b) "If it is reasonable to think that a Supreme Court justice can be bought so cheap, the nation is in deeper trouble than I had imagined.''

BIBLIOGRAPHY

(Giarratano 93). Joseph Giarratano, *CLIPS User's Guide*, NASA, Version 6.2 of CLIPS, 1993.

(Kaelbling 98). Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra, "Planning and Acting in Partially Observable Stochastic Domains," *Artificial Intelligence* 101 pp. 99-134, 1998.

(Klir 98). George J. Klir and Mark J. Weirman, *Uncertainty-Based Information, Physica-Verlag*, 1998.

(Lakemeyer 03). ed. by Gerhard Lakemeyer and Bernhard Nebel, *Exploring Artificial Intelligence in the New Millennium,* Morgan Kaufmann Publishers, 2003.

(Lakoff 00). George Lakoff and Rafael E. Nunez, *Where Mathematics Comes From*, Basic Books, 2000.

(Merritt 04). Dennis Merrit, "Building a Custom Rule Engine with PROLOG," *Dr. Dobb's Journal*, pp. 40-45, March 2004.

(Russell 03). Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach,* 2nd Edition, 2002.