MODULE V

22.3 FUZZY SETS

Fuzzy sets introduce a certain amount of vagueness to reduce complexity of comprehension. This set consists of elements that signify the degree or grade of membership to a fuzzy aspect. Membership values usually use closed intervals and denote the sense of belonging of a member of a crisp set to a fuzzy set. To make the point clear consider a crisp set A comprising of elements that signify the ages of a set of people in years.

$$A = \{2,4,10,15,21,30,35,40,45,60,70\}$$

We could classify age in terms of what are known as fuzzy linguistic variables – *infant, child, adolescent, adult, young* and *old*. A person whose age is 15 is no doubt young but how would you categorize a person who is 30. If the latter is to be considered *young* what about the person who is 40? Is he old? How do we translate all these into numbers for efficiently making the computer understand what our feelings about age are?

Inspect the Table 22.1 giving ages and their membership to a particular set.

Table 22.1 *Ages and their memberships*

Age	*Infant*	*Child*	*Adolescent*	*Young*	*Adult*	*Old*
2	1	0	0	1	0	0
4	0.1	0.5	0	1	0	0
10	0	1	0.3	1	0	0
15	0	0.8	1	1	0	0
21	0	0	0.1	1	0.8	0.1
30	0	0	0	0.6	1	0.3
35	0	0	0	0.5	1	0.35
40	0	0	0	0.4	1	0.4
45	0	0	0	0.2	1	0.6
60	0	0	0	0	1	0.8
70	0	0	0	0	1	1

The values in the table indicate memberships to the fuzzy sets – *infant, child, adolescent, young, adult* and *old*. Thus a child of age 4 belongs only 50% to the fuzzy set *child* while when he is 10 years he is a 100% member. Note that membership is different from probabilities. Memberships do not necessarily add up to 1. The entries in the table have been made after a manual evaluation of the different ages.

22.4 SOME FUZZY TERMINOLOGY

Now that you have a notion of fuzziness we could define some terms based on a Universal set U.

Universe of Discourse (U):

This is defined as the range of all possible values that comprise the input to the fuzzy system.

Fuzzy Set

Any set that empowers its members to have different grades of membership (based on a membership function) in an interval [0,1] is a fuzzy set.

Membership function

The membership function μ_A which forms the basis of a fuzzy set is given by

$$\mu_A: U \rightarrow [0,1]$$

where the closed interval is one that holds real numbers.

Support of a fuzzy set (S_f)

The support S of a fuzzy set f, in a universal crisp set U is that set which contains all elements of the set U that have a non-zero membership value in f. For instance, the support of the fuzzy set *adult* is

$$S_{adult} = \{21,30,35,40,45,60,70\}$$

Depiction of a fuzzy set

A fuzzy set f in a universal crisp set U, is written as

$$f = \mu_1 / s_1 + \mu_2 / s_2 + \mu_3 / s_3 + \ldots + \mu_n / s_n$$

where μ_i is the membership and s_i is the corresponding term in the *support* set of f i.e. S_f.

This is however only a representation and has *no algebraic implication* (the slash and + signs do not have any meaning).

Accordingly,

$$\text{Old} = 0.1/21 + 0.3/30 + 0.35/35 + 0.4/40 + 0.6/45 + 0.8/60 + 1/70$$

Fuzzy Set Operations

- **Union:** The membership function of the union of two fuzzy sets A and B is defined as the maximum of the two individual membership functions. It is equivalent to the Boolean OR operation.

$$\mu_{A \cup B} = \max(\mu_A, \mu_B)$$

- **Intersection:** The membership function of the intersection of two fuzzy sets A and B is defined as the minimum of the two individual membership functions and is equivalent to the Boolean AND operation.

$$\mu_{A \cap B} = \min(\mu_A, \mu_B)$$

- **Complement:** The membership function of the complement of a fuzzy set A is defined as the negation of the specified membership function: $\mu_{\bar{A}}$. This is equivalent to the Boolean NOT operation

$$\mu_{\bar{A}} = \mu_{A \cup B} = (1 - \mu_A)$$

It may be further noted here that the laws of Associativity, Commutativity , Distributivity and De Morgan's laws hold in fuzzy set theory too.

CONCEPT OF FUZZY NUMBERS

If a fuzzy set is convex and normalized, and its membership function is defined in R and piecewise continuous, it is called as fuzzy number. So fuzzy number (fuzzy set) represents a real number interval whose boundary is fuzzy.Fuzzy number is expressed as a fuzzy set defining a fuzzy interval in the real number R. Since the boundary of this interval is ambiguous, the interval is also a fuzzy set. Generally a fuzzy interval is represented by two end points a1 and a3 and a peak point a2 as [a1, a2, a3] (Figure1). The α -cut operation can be also applied to the fuzzy number. If we denote α -cut interval for fuzzy number A as A α , the obtained interval α A is defined as

$$A_\alpha = [a_1^{(\alpha)}, a_3^{(\alpha)}]$$

A fuzzy number is simply an ordinary number whose precise value is somewhat uncertain. Fuzzy numbers are used in statistics, computer programming, engineering, and experimental science. The arithmetic operators on fuzzy numbers are basic content in fuzzy mathematics. Operation of fuzzy number can be generalized from that of crisp interval. The operations of interval are discussed. Multiplication operation on fuzzy numbers is defined by the extension principle. Based on extension principle, nonlinear programming method, analytical method, computer drawing method and computer simulation method are used for solving multiplication operation of two fuzzy numbers

We can also know that it is an ordinary crisp interval (Figure 2).

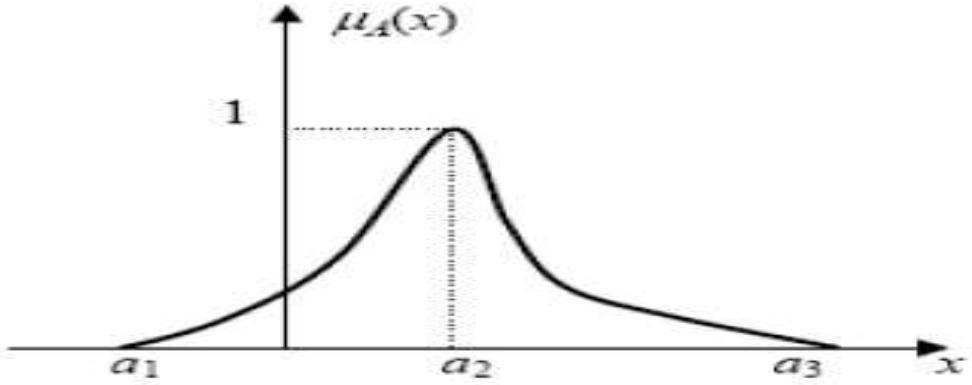

Figure 1 Fuzzy Number $A = [a_1, a_2, a_3]$

Fuzzy number should be normalized and convex. Here the condition of normalization implies that maximum membership value is 1.

$$\exists x_0 \in R, \mu_{\tilde{A}}(x_0) = 1$$

B. Operation of α -cut Interval

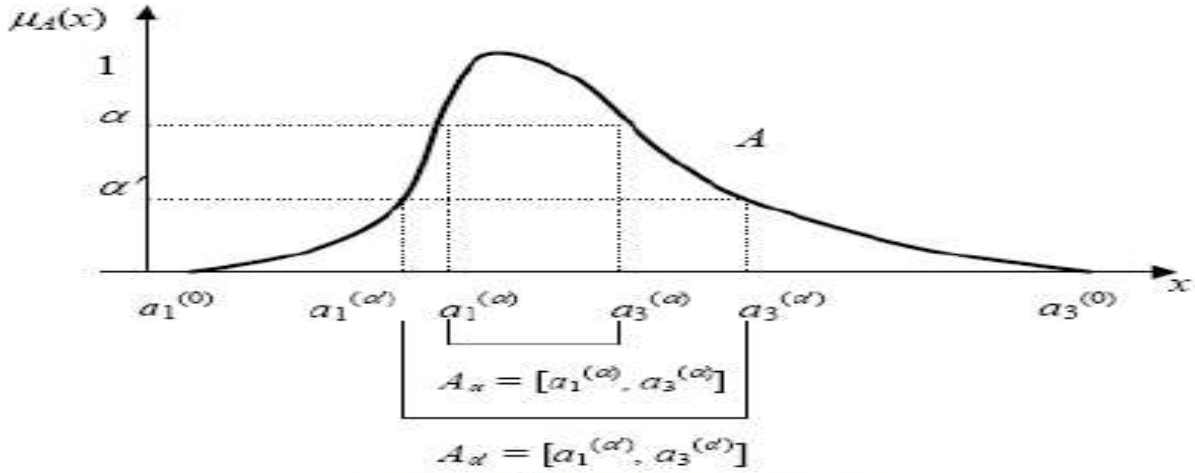

Figure 2 α -cut of fuzzy number

$$(\alpha' < \alpha) \Rightarrow (A_\alpha \subset A_{\alpha'})$$

The convex condition is that the line by α -cut is continuous and α -cut interval satisfies the following relation.

$$A_\alpha = [a_1^{(\alpha)}, a_3^{(\alpha)}]$$

$$(\alpha' < \alpha) \Rightarrow (a_1^{(\alpha')} \leq a_1^{(\alpha)}, a_3^{(\alpha')} \geq a_3^{(\alpha)})$$

The convex condition may also be written as,

$$(\alpha' < \alpha) \Rightarrow (A_\alpha \subset A_{\alpha'})$$

Operation of fuzzy number can be generalized from that of crisp interval. Let's have a look at the operations of interval.

$$\forall a_1, a_3, b_1, b_3 \in \mathbf{R}$$

$$A = [a_1, a_3], B = [b_1, b_3]$$

Assuming A and B as numbers expressed as interval, main operations of interval are

i) Addition

$$[a_1, a_3] (+) [b_1, b_3] = [a_1 + b_1, a_3 + b_3]$$

ii) Subtraction

$$[a_1, a_3] (-) [b_1, b_3] = [a_1 - b_3, a_3 - b_1]$$

iii) Multiplication

$$[a_1, a_3] (\bullet) [b_1, b_3] = [a_1 \bullet b_1 \wedge a_1 \bullet b_3 \wedge a_3 \bullet b_1 \wedge a_3 \bullet b_3,$$

$$a_1 \bullet b_1 \vee a_1 \bullet b_3 \vee a_3 \bullet b_1 \vee a_3 \bullet b_3]$$

iv) Division

$$[a_1, a_3] (/) [b_1, b_3] = [a_1 / b_1 \wedge a_1 / b_3 \wedge a_3 / b_1 \wedge a_3 / b_3,$$

$$a_1 / b_1 \vee a_1 / b_3 \vee a_3 / b_1 \vee a_3 / b_3]$$

excluding the case $b_1 = 0$ or $b_3 = 0$

v) Inverse interval

$$[a_1, a_3]^{-1} = [1 / a_1 \wedge 1 / a_3, 1 / a_1 \vee 1 / a_3]$$

excluding the case $a_1 = 0$ or $a_3 = 0$

When previous sets A and B is defined in the positive real number **R+**, the operations of multiplication, division, and inverse interval are written as,

iii') Multiplication

$$[a_1, a_3] (\bullet) [b_1, b_3] = [a_1 \bullet b_1, a_3 \bullet b_3]$$

iv') Division

$$[a_1, a_3] (/) [b_1, b_3] = [a_1 / b_3, a_3 / b_1]$$

v) Inverse Interval

$$[a_1, a_3]^{-1} = [1 / a_3, 1 / a_1]$$

vi) Minimum

$$[a_1, a_3] (\wedge) [b_1, b_3] = [a_1 \wedge b_1, a_3 \wedge b_3]$$

vii) Maximum

$$[a_1, a_3] (\vee) [b_1, b_3] = [a_1 \vee b_1, a_3 \vee b_3]$$

Example 1 There are two intervals A and B,

$$A = [3, 5], B = [-2, 7]$$

IV. ALGEBRAIC OPERATIONS ON FUZZY SETS

In addition to the operations of union and intersection, one can define a number of other ways of forming combinations of fuzzy sets and relating them to one another. Among the more important of these are the following.

Algebraic product. The *algebraic product* of A and B is denoted by AB and is defined in terms of the membership functions of A and B by the relation

$$f_{AB} = f_A f_B . \tag{14}$$

Clearly,

$$AB \subset A \cap B. \tag{15}$$

Algebraic sum.[4] The *algebraic sum* of A and B is denoted by $A + B$ and is defined by

$$f_{A+B} = f_A + f_B \tag{16}$$

provided the sum $f_A + f_B$ is less than or equal to unity. Thus, unlike the algebraic product, the algebraic sum is meaningful only when the condition $f_A(x) + f_B(x) \leqq 1$ is satisfied for all x.

Absolute difference. The *absolute difference* of A and B is denoted by $|A - B|$ and is defined by

$$f_{|A-B|} = |f_A - f_B|.$$

Note that in the case of ordinary sets $|A - B|$ reduces to the relative complement of $A \cap B$ in $A \cup B$.

[4] The dual of the algebraic product is the *sum* $A \oplus B = (A'B')' = A + B - AB$. (This was pointed out by T. Cover.) Note that for ordinary sets $\cap$ and the algebraic product are equivalent operations, as are $\cup$ and $\oplus$.

Convex combination. By a convex combination of two vectors f and g is usually meant a linear combination of f and g of the form $\lambda f + (1 - \lambda)g$, in which $0 \leqq \lambda \leqq 1$. This mode of combining f and g can be generalized to fuzzy sets in the following manner.

Let A, B, and Λ be arbitrary fuzzy sets. The *convex combination of* A, B, *and* Λ is denoted by $(A, B; \Lambda)$ and is defined by the relation

$$(A, B; \Lambda) = \Lambda A + \Lambda' B \tag{17}$$

where Λ' is the complement of Λ. Written out in terms of membership functions, (17) reads

$$f_{(A,B;\Lambda)}(x) = f_\Lambda(x) f_A(x) + [1 - f_\Lambda(x)] f_B(x), \qquad x \in X. \tag{18}$$

A basic property of the convex combination of A, B, and Λ is expressed by

$$A \cap B \subset (A, B; \Lambda) \subset A \cup B \qquad \text{for all } \Lambda. \tag{19}$$

This property is an immediate consequence of the inequalities

$$\begin{aligned} \text{Min}\,[f_A(x), f_B(x)] &\leqq \lambda f_A(x) + (1 - \lambda) f_B(x) \\ &\leqq \text{Max}\,[f_A(x), f_B(x)], \qquad x \in X \end{aligned} \tag{20}$$

which hold for all λ in $[0, 1]$. It is of interest to observe that, given any fuzzy set C satisfying $A \cap B \subset C \subset A \cup B$, one can always find a fuzzy set Λ such that $C = (A, B; \Lambda)$. The membership function of this set is given by

$$f_\Lambda(x) = \frac{f_C(x) - f_B(x)}{f_A(x) - f_B(x)}, \qquad x \in X. \tag{21}$$

Fuzzy relation. The concept of a *relation* (which is a generalization of that of a *function*) has a natural extension to fuzzy sets and plays an important role in the theory of such sets and their applications—just as it does in the case of ordinary sets. In the sequel, we shall merely define the notion of a fuzzy relation and touch upon a few related concepts.

Ordinarily, a relation is defined as a set of ordered pairs (Halmos, 1960); e.g., the set of all ordered pairs of real numbers x and y such that $x \geqq y$. In the context of fuzzy sets, a *fuzzy relation in* X is a fuzzy set in the product space $X \times X$. For example, the relation denoted by $x \gg y$, $x, y \in R^1$, may be regarded as a fuzzy set A in R^2, with the membership function of A, $f_A(x, y)$, having the following (subjective) representative values: $f_A(10, 5) = 0$; $f_A(100, 10) = 0.7$; $f_A(100, 1) = 1$; etc.

3.1 Fuzzy Sets and Membership Functions

To appreciate the nature of a fuzzy set, let us consider the following hypothetical example taken from Laviolette, Seaman, Barrett, and Woodall (1995). Let $\mathcal{X}$ denote the set of integers between 0 and 10, both inclusive; that is,

$$\mathcal{X} = \{0, 1, \ldots, 10\}.$$

Suppose that we are interested in a subset $\widetilde{A}$ of $\mathcal{X}$, where $\widetilde{A}$ contains all of the "medium" integers of $\mathcal{X}$. Thus

$$\widetilde{A} = \{x; x \in \mathcal{X} \text{ and } x \text{ is "medium"}\}.$$

For the purposes of this section, x is not to be viewed as an outcome of any experiment. Some reasons as to why one would be interested in sets like $\widetilde{A}$ are given in Section 3.4. Clearly, to be able to specify $\widetilde{A}$, we must be precise as to what we mean by a medium integer; that is, we must be able to operationalize (Deming 1986, p. 276) the term "medium integer." Whereas most would agree that 5 is a medium integer, what is the disposition of an integer like 7? Is 7 a medium integer, or is it a large integer? Our uncertainty (or vagueness) about classifying 7 as a member of the subset $\widetilde{A}$ makes $\widetilde{A}$ a fuzzy set. The uncertainty of classification arises because the boundaries of $\widetilde{A}$ are not sharp. The subset $\widetilde{A}$ rejects the law of the excluded middle, because an integer like 7 can simultaneously belong to and not belong to $\widetilde{A}$.

Membership functions were introduced as a way of dealing with the foregoing form of uncertainty of classification. Specifically, the number $m_{\widetilde{A}}(x)$ which lies between 0 and 1, reflects an assessor's view of the extent to which $x \in \widetilde{A}$. As a function of x, $m_{\widetilde{A}}(x)$ is known as the *membership function* of set $\widetilde{A}$. Clearly, the membership function is subjective, because it is specific to an individual assessor or a group of assessors. We also assume that for each $x \in \mathcal{X}$, the assessor is able to assign an $m_{\widetilde{A}}(x)$, and that this can be done for all subsets of the type $\widetilde{A}$ that are of interest.

If $m_{\widetilde{A}}(x) = 1$ (or 0) for all $x \in \mathcal{X}$, then $\widetilde{A}$ is the usual well-defined sharp (or crisp) set. Thus the notion of fuzzy sets incorporates that of crisp sets as a special case, and because it is on crisp sets that probability measures have been defined, our aim here is to develop a foundation for constructing probability measures of fuzzy sets.

EXPERT SYSTEM

Expert systems solve problems (such as the ones in Fig. 1.1) that are normally solved by human "experts." To solve expert-level problems, expert systems need access to a substantial domain knowledge base, which must be built as efficiently as possible. They also need to exploit one or more reasoning mechanisms to apply their knowledge to the problems they are given. Then they need a mechanism for explaining what they have done to the users who rely on them. One way to look at expert systems is that they represent applied AI in a very broad sense. They tend to lag several years behind research advances, but because they are tackling harder and harder problems, they will eventually be able to make use of all of the kinds of results that we have described throughout this book. So this chapter is in some ways a review of much of what we have already discussed.

The problems that expert systems deal with are highly diverse. There are some general issues that arise across these varying domains. But it also turns out that there are powerful techniques that can be defined for specific classes of problems. Recall that in Section 2.3.8 we introduced the notion of problem classification and we described some classes into which problems can be organized. Throughout this chapter we have occasion to return to this idea, and we see how some key problem characteristics play an important role in guiding the design of problem-solving systems. For example, it is now clear that tools that are developed to support one classification or diagnosis task are often useful for another, while different tools are useful for solving various kinds of design tasks.

20.1 REPRESENTING AND USING DOMAIN KNOWLEDGE

Expert systems are complex AI programs. Almost all the techniques that we described in Parts I and II have been exploited in at least one expert system. However, the most widely used way of representing domain knowledge in expert systems is as a set of production rules, which are often coupled with a frame system that defines the objects that occur in the rules. In Section 8.2, we saw one example of an expert system rule, which was taken from the MYCIN system. Let's look at a few additional examples drawn from some other

representative expert systems. All the rules we show are English versions of the actual rules that the systems use. Differences among these rules illustrate some of the important differences in the ways that expert systems operate.

R1 [McDermott, 1982; McDermott, 19841 (sometimes also called XCON) is a program that configures DEC VAX systems. Its rules look like this:

```
If: the most current active context is distributing
        massbus devices, and
    there is a single-port disk drive that has not been
        assigned to a massbus, and
    there are no unassigned dual-port disk drives, and
        the number of devices that each massbus should
        support is known, and
    there is a massbus that has been assigned at least
        one disk drive and that should support additional
        disk drives,
    and the type of cable needed to connect the disk drive
        to the previous device on the massbus is known
then: assign the disk drive to the massbus.
```

Notice that R1's rules, unlike MYCIN's, contain no numeric measures of certainty. In the task domain with which R1 deals, it is possible to state exactly the correct thing to be done in each particular set of circumstances (although it may require a relatively complex set of antecedents to do so). One reason for this is that there exists a good deal of human expertise in this area. Another is that since R1 is doing a design task (in contrast to the diagnosis task performed by MYCIN), it is not necessary to consider all possible alternatives; one good one is enough. As a result, probabilistic information is not necessary in R1.

PROSPECTOR [Duda *et al.*. 1979; Hart *et al.*, 1978] is a program that provides advice on mineral exploration. Its rules look like this:

```
If: magnetite or pyrite in disseminated or veinlet form is present
then: (2, -4) there is favorable mineralization and texture
    for the propylitic stage.
```

In PROSPECTOR, each rule contains two confidence estimates. The first indicates the extent to which the presence of the evidence described in the condition part of the rule suggests the validity of the rule's conclusion. In the PROSPECTOR rule shown above, the number 2 indicates that the presence of the evidence is mildly encouraging. The second confidence estimate measures the extent to which the evidence is necessary to the validity of the conclusion, or stated another way, the extent to which the lack of the evidence indicates that the conclusion is not valid. In the example rule shown above, the number -4 indicates that the absence of the evidence is strongly discouraging for the conclusion.

DESIGN ADVISOR [Steele *et al.*, 1989] is a system that critiques chip designs. Its rules look like:

```
If: the sequential level count of ELEMENT is greater than 2,
    UNLESS the signal of ELEMENT is resetable
then: critique for poor resetability
DEFEAT: poor resetability of ELEMENT
due to: sequential level count of ELEMENT greater than 2
by: ELEMENT is directly resetable
```

The DESIGN ADVISOR gives advice to a chip designer, who can accept or reject the advice. If the advice is rejected, the system can exploit a justification-based truth maintenance system to revise its model of the circuit. The first rule shown here says that an element should be criticized for poor resetability if its sequential level count is greater than two, unless its signal is currently believed to be resetable. Resetability is a fairly common condition, so it is mentioned explicitly in this first rule. But there is also a much less common condition, called direct resetability. The DESIGN ADVISOR does not even bother to consider that condition unless it gets in trouble with its advice. At that point, it can exploit the second of the rules shown above. Specifically, if the chip designer rejects a critique about resetability and if that critique was based on a high level count, then the system will attempt to discover (possibly by asking the designer) whether the element is directly resetable. If it is, then the original rule is defeated and the conclusion withdrawn.

Reasoning with the Knowledge

As these example rules have shown, expert systems exploit many of the representation and reasoning mechanisms that we have discussed. Because these programs are usually written primarily as rule-based systems, forward chaining, backward chaining, or some combination of the two, is usually used. For example, MYCIN used backward chaining to discover what organisms were present; then it used forward chaining to reason from the organisms to a treatment regime. RI, on the other hand, used forward chaining. As the field of expert systems matures, more systems that exploit other kinds of reasoning mechanisms are being developed. The DESIGN ADVISOR is an example of such a system; in addition to exploiting rules, it makes extensive use of a justification-based truth maintenance system.

20.2 EXPERT SYSTEM SHELLS

Initially, each expert system that was built was created from scratch, usually in LISP. But, after several systems had been built this way, it became clear that these systems often had a lot in common. In particular, since the systems were constructed as a set of declarative representations (mostly rules) combined with an interpreter for those representations, it was possible to separate the interpreter from the domain-specific knowledge and thus to create a system that could be used to construct new expert systems by adding new knowledge corresponding to the new problem domain. The resulting interpreters are called *shells*. One influential example of such a shell is EMYCIN (for Empty MYCIN) [Buchanan and Shortliffe, 1984], which was derived from MYCIN.

There are now several commercially available shells that serve as the basis for many of the expert systems currently being built. These shells provide much greater flexibility in representing knowledge and in reasoning with it than MYCIN did. They typically support rules, frames, truth maintenance systems, and a variety of other reasoning mechanisms.

Early expert system shells provided mechanisms for knowledge representation, reasoning, and explanation. Later, tools for knowledge acquisition were added, as we see in Section 20.4. But as experience with using these systems to solve real world problems grew, it became clear that expert system shells needed to do something else as well. They needed to make it easy to integrate expert systems with other kinds of programs. Expert systems cannot operate in a vacuum, any more than their human counterparts can. They need access to corporate databases, and access to them needs to be controlled just as it does for other systems. They are often embedded within larger application programs that use primarily conventional programming techniques. So one of the important features that a shell must provide is an easy-to-use interface between an expert system that is written with the shell and a larger, probably more conventional, programming environment.

20.3 EXPLANATION

In order for an expert system to be an effective tool, people must be able to interact with it easily. To facilitate this interaction, the expert system must have the following two capabilities in addition to the ability to perform its underlying task:

- Explain its reasoning. In many of the domains in which expert systems operate, people will not accept results unless they have been convinced of the accuracy of the reasoning process that produced those results. This is particularly true, for example, in medicine, where a doctor must accept ultimate responsibility for a diagnosis, even if that diagnosis was arrived at with considerable help from a program. Thus it is important that the reasoning process used in such programs proceed in understandable steps and that enough meta-knowledge (knowledge about the reasoning process) be available so the explanations of those steps can be generated.
- Acquire new knowledge and modifications of old knowledge. Since expert systems derive their power from the richness of the knowledge bases they exploit, it is extremely important that those knowledge bases be as complete and as accurate as possible. But often there exists no standard codification of that knowledge; rather it exists only inside the heads of human experts. One way to get this knowledge into a program is through interaction with the human expert. Another way is to have the program learn expert behavior from raw data.

TEIRESIAS [Davis, 1982; Davis, 1977] was the first program to support explanation and knowledge acquisition. TEIRESIAS served as a front-end for the MYCIN expert system. A fragment of a TEIRESIAS-MYCIN conversation with a user (a doctor) is shown in Fig. 20.1. The program has asked for a piece of information that it needs in order to continue its reasoning. The doctor wants to know why the program wants the information, and later asks how the program arrived at a conclusion that it claimed it had reached.

An important premise underlying TEIRESIAS's approach to explanation is that the behavior of a program can be explained simply by referring to a trace of the program's execution. There are ways in which this assumption limits the kinds of explanations that can be produced, but it does minimize the overhead involved in generating each explanation. To understand how TEIRESIAS generates explanations of MYCIN's behavior, we need to know how that behavior is structured.

MYCIN attempts to solve its goal of recommending a therapy for a particular patient by first finding the cause of the patient's illness. It uses its production rules to reason backward from goals to clinical observations. To solve the top-level diagnostic goal, it looks for rules whose right sides suggest diseases. It then uses the left sides of those rules (the preconditions) to set up subgoals whose success would enable the rules to be invoked. These subgoals are again matched against rules, and their preconditions are used to set up additional subgoals. Whenever a precondition describes a specific piece of clinical evidence, MYCIN uses that evidence if it already has access to it. Otherwise, it asks the user to provide the information. In order that MYCIN's requests for information will appear coherent to the user, the actual goals that MYCIN sets up are often more general than they need be to satisfy the preconditions of an individual rule. For example, if a precondition specifies that the identity of an organism is *X*, MYCIN will set up the goal "infer identity." This approach also means that if another rule mentions the organism's identity, no further work will be required, since the identity will be known.

We can now return to the trace of TEIRESIAS-MYCIN's behavior shown in Fig. 20.1. The first question that the user asks is a "WHY" question, which is assumed to mean "Why do you need to know that?" Particularly for clinical tests that are either expensive or dangerous, it is important for the doctor to be convinced that the information is really needed before ordering the test. (Requests for sensitive or confidential information present similar difficulties.) Because MYCIN is reasoning backward, the question can easily be answered by examining the goal tree. Doing so provides two kinds of information:

- What higher-level question might the system be able to answer if it had the requested piece of information? (In this case, it could help determine the category of ORGANISM-1.)
- What other information does the system already have that makes it think that the requested piece of knowledge would help? (In this case, facts [2.1] to [2.4].)

```
Is the patient's illness with ORGANISM-1 a hospital-acquired infection?
** WHY
[i.e., WHY is it important to determine whether or not the
infection with ORGANISM-1 was acquired while the patient
was hospitalized?]
[2.0]  This will aid in determining the category of
       ORGANISM-1. It has already been established that
[2.1]  the gram stain of ORGANISM-1 is gramneg, and
[2.2]  the morphology of ORGANISM-1 is rod, and
[2.3]  the aerobicity of ORGANISM-1 is facultative
       Therefore, if
[2.4]  the infection with ORGANISM-1 was not acquired
       while the patient was hospitalized, then
```

20.4 KNOWLEDGE ACQUISITION

How are expert systems built? Typically, a knowledge engineer interviews a domain expert to elucidate expert knowledge, which is then translated into rules. After the initial system is built, it must be iteratively refined until it approximates expert-level performance. This process is expensive and time-consuming, so it is worthwhile to look for more automatic ways of constructing expert knowledge bases. While no totally automatic knowledge acquisition systems yet exist, there are many programs that interact with domain experts to extract expert knowledge efficiently. These programs provide support for the following activities:

- Entering knowledge
- Maintaining knowledge base consistency
- Ensuring knowledge base completeness

The most useful knowledge acquisition programs are those that are restricted to a particular problem-solving paradigm, e.g., diagnosis or design. It is important to be able to enumerate the roles that knowledge can play in the problem-solving process. For example, if the paradigm is diagnosis, then the program can structure its knowledge base around symptoms, hypotheses, and causes. It can identify symptoms for which the expert has not yet provided causes. Since one symptom may have multiple causes, the program can ask for knowledge about how to decide when one hypothesis is better than another. If we move to another type of problem-solving, say designing artifacts, then these acquisition strategies no longer apply, and we must look for other ways of profitably interacting with an expert. We now examine two knowledge acquisition systems in detail.

MOLE [Eshelman, 1988] is a knowledge acquisition system for heuristic classification problems, such as diagnosing diseases. In particular, it is used in conjunction with the *cover-and-differentiate* problem-solving method. An expert system produced by MOLE accepts input data, comes up with a set of candidate explanations or classifications that cover (or explain) the data, then uses differentiating knowledge to determine which one is best. The process is iterative, since explanations must themselves be justified, until ultimate causes are ascertained.

MOLE interacts with a domain expert to produce a knowledge base that a system called MOLE-p (for MOLE-performance) uses to solve problems. The acquisition proceeds through several steps:

1. Initial knowledge base construction. MOLE asks the expert to list common symptoms or complaints that might require diagnosis. For each symptom, MOLE prompts for a list of possible explanations. MOLE then iteratively seeks out higher-level explanations until it comes up with a set of ultimate causes. During this process, MOLE builds an influence network similar to the belief networks we saw in Chapter 8.

   Whenever an event has multiple explanations, MOLE tries to determine the conditions under which one explanation is correct. The expert provides *covering* knowledge, that is, the knowledge that a hypothesized event might be the cause of a certain symptom. MOLE then tries to infer *anticipatory* knowledge, which says that if the hypothesized event does occur, then the symptom will definitely appear. This knowledge allows the system to rule out certain hypotheses on the basis that specific symptoms are absent.
2. Refinement of the knowledge base. MOLE now tries to identify the weaknesses of the knowledge base. One approach is to find holes and prompt the expert to fill them. It is difficult, in general, to know whether a knowledge base is complete, so instead MOLE lets the expert watch MOLE-p solving sample

problems. Whenever MOLE-p makes an incorrect diagnosis, the expert adds new knowledge. There are several ways in which MOLE-p can reach the wrong conclusion. It may incorrectly reject a hypothesis because it does not feel that the hypothesis is needed to explain any symptom. It may advance a hypothesis because it is needed to explain some otherwise inexplicable hypothesis. Or it may lack differentiating knowledge for choosing between alternative hypotheses.

For example, suppose we have a patient with symptoms A and B. Further suppose that symptom A could be caused by events X and Y, and that symptom B can be caused by Y and Z. MOLE-p might conclude Y, since it explains both A and B. If the expert indicates that this decision was incorrect, then MOLE will ask what evidence should be used to prefer X and/or Z over Y.

MOLE has been used to build systems that diagnose problems with car engines, problems in steel-rolling mills, and inefficiencies in coal-burning power plants. For MOLE to be applicable, however, it must be possible to preenumerate solutions or classifications. It must also be practical to encode the knowledge in terms of covering and differentiating.

But suppose our task is to design an artifact, for example, an elevator system. It is no longer possible to preenumerate all solutions. Instead, we must assign values to a large number of parameters, such as the width of the platform, the type of door, the cable weight, and the cable strength. These parameters must be consistent with each other, and they must result in a design that satisfies external constraints imposed by cost factors, the type of building involved, and expected payloads.

One problem-solving method useful for design tasks is called *propose-and-revise*. Propose-and-revise systems build up solutions incrementally. First, the system proposes an extension to the current design. Then it checks whether the extension violates any global or local constraints. Constraint violations are then fixed, and the process repeats. It turns out that domain experts are good at listing overall design constraints and at providing local constraints on individual parameters, but not so good at explaining how to arrive at global solutions. The SALT program [Marcus and McDermott, 1989] provides mechanisms for elucidating this knowledge from the expert.

Like MOLE, SALT builds a dependency network as it converses with the expert. Each node stands for a value of a parameter that must be acquired or generated. There are three kinds of links: *contributes-to*, *constrains*, and *suggests-revision-of*. Associated with the first type of link are procedures that allow SALT to generate a value for one parameter based on the value of another. The second type of link, *constrains*, rules out certain parameter values. The third link, *suggests-revision-of*, points to ways in which a constraint violation can be fixed. SALT uses the following heuristics to guide the acquisition process:

1. Every noninput node in the network needs at least one *contributes-to* link coming into it. If links are missing, the expert is prompted to fill them in.
2. No *contributes-to* loops are allowed in the network. Without a value for at least one parameter in the loop, it is impossible to compute values for any parameter in that loop. If a loop exists, SALT tries to transform one of the *contributes-to* links into a *constrains* link.
3. Constraining links should have *suggests-revision-of* links associated with them. These include *constrains* links that are created when dependency loops are broken.

Control knowledge is also important. It is critical that the system propose extensions and revisions that lead toward a design solution. SALT allows the expert to rate revisions in terms of how much trouble they tend to produce.

SALT compiles its dependency network into a set of production rules. As with MOLE, an expert can watch the production system solve problems and can override the system's decision. At that point, the knowledge base can be changed or the override can be logged for future inspection.

The process of interviewing a human expert to extract expertise presents a number of difficulties, regardless of whether the interview is conducted by a human or by a machine. Experts are surprisingly inarticulate when it comes to how they solve problems. They do not seem to have access to the low-level details of what they do and are especially inadequate suppliers of any type of statistical information. There is, therefore, a great deal of interest in building systems that automatically induce their own rules by looking at sample problems and solutions. With inductive techniques, an expert needs only to provide the conceptual framework for a problem and a set of useful examples.

For example, consider a bank's problem in deciding whether to approve a loan. One approach to automating this task is to interview loan officers in an attempt to extract their domain knowledge. Another approach is to inspect the record of loans the bank has made in the past and then try to generate automatically rules that will maximize the number of good loans and minimize the number of bad ones in the future.

META-DENDRAL [Mitchell, 1978] was the first program to use learning techniques to construct rules for an expert system automatically. It built rules to be used by DENDRAL, whose job was to determine the structure of complex chemical compounds. META-DENDRAL was able to induce its rules based on a set of mass spectrometry data; it was then able to identify molecular structures with very high accuracy. META-DENDRAL used the version space learning algorithm, which we discussed in Chapter 17. Another popular method for automatically constructing expert systems is the induction of decision trees, data structures we described in Section 17.5.3. Decision tree expert systems have been built for assessing consumer credit applications, analyzing hypothyroid conditions, and diagnosing soybean diseases, among many other applications.