MODULE IV

LEARNING

17.1 WHAT IS LEARNING?

One of the most often heard criticisms of AI is that machines cannot be called intelligent until they are able to learn to do new things and to adapt to new situations, rather than simply doing as they are told to do. There can be little question that the ability to adapt to new surroundings and to solve new problems is an important characteristic of intelligent entities. Can we expect to see such abilities in programs? Ada Augusta, one of the earliest philosophers of computing, wrote that

> The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order it* to perform. [Lovelace, 1961]

This remark has been interpreted by several AI critics as saying that computers cannot learn. In fact, it does not say that at all. Nothing prevents us from telling a computer how to interpret its inputs in such a way that its performance gradually improves.

Rather than asking in advance whether it is possible for computers to "learn," it is much more enlightening to try to describe exactly what activities we mean when we say "learning" and what mechanisms could be used to enable us to perform those activities. Simon [1983] has proposed that learning denotes

> ...changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

As thus defined, learning covers a wide range of phenomena. At one end of the spectrum is *skill refinement*. People get better at many tasks simply by practicing. The more you ride a bicycle or play tennis, the better you get. At the other end of the spectrum lies *knowledge acquisition*. As we have seen, many AI programs draw

heavily on knowledge as their source of power. Knowledge is generally acquired through experience, and such acquisition is the focus of this chapter.

Knowledge acquisition itself includes many different activities. Simple storing of computed information, or *rote learning*, is the most basic learning activity. Many computer programs, e.g., database systems, can be said to "learn" in this sense, although most people would not call such simple storage learning. However, many AI programs are able to improve their performance substantially through rote-learning techniques, and we will look at one example in depth, the checker-playing program of Samuel [1963].

Another way we learn is through taking advice from others. Advice taking is similar to rote learning, but high-level advice may not be in a form simple enough for a program to use directly in problem-solving. The advice may need to be first *operationalized*, a process explored in Section 17.3.

People also learn through their own problem-solving experience. After solving a complex problem, we remember the structure of the problem and the methods we used to solve it. The next time we see the problem, we can solve it more efficiently. Moreover, we can generalize from our experience to solve related problems more easily. In contrast to advice taking, learning from problem-solving experience does not usually involve gathering new knowledge that was previously unavailable to the learning program. That is, the program remembers its experiences and generalizes from them, but does not add to the transitive closure[1] of its knowledge, in the sense that an advice-taking program would, i.e., by receiving stimuli from the outside world. In large problem spaces, however, efficiency gains are critical. Practically speaking, learning can mean the difference between solving a problem rapidly and not solving it at all. In addition, programs that learn through problem-solving experience may be able to come up with qualitatively better solutions in the future.

Another form of learning that does involve stimuli from the outside is *learning from examples*. We often learn to classify things in the world without being given explicit rules. For example, adults can differentiate between cats and dogs, but small children often cannot. Somewhere along the line, we induce a method for telling cats from dogs based on seeing numerous examples of each. Learning from examples usually involves a teacher who helps us classify things by correcting us when we are wrong. Sometimes, however, a program can discover things without the aid of a teacher.

AI researchers have proposed many mechanisms for doing the kinds of learning described above. In this chapter, we discuss several of them. But keep in mind throughout this discussion that learning is itself a problem-solving process. In fact, it is very difficult to formulate a precise definition of learning that distinguishes it from other problem-solving tasks. Thus it should come as no surprise that, throughout this chapter, we will make extensive use of both the problem-solving mechanisms and the knowledge representation techniques that were presented in Parts I and II.

17.2 ROTE LEARNING

When a computer stores a piece of data, it is performing a rudimentary form of learning. After all, this act of storage presumably allows the program to perform better in the future (otherwise, why bother?). In the case of data caching, we store computed values so that we do not have to recompute them later. When computation is more expensive than recall, this strategy can save a significant amount of time. Caching has been used in AI programs to produce some surprising performance improvements. Such caching is known as *rote learning*.

In Chapter 12, we mentioned one of the earliest game-playing programs, Samuel's checkers program [Samuel, 1963]. This program learned to play checkers well enough to beat its creator. It exploited two kinds of learning: rote learning, which we look at now, and parameter (or coefficient) adjustment, which is described in Section 17.4.1. Samuel's program used the minimax search procedure to explore checkers game trees. As

is the case with all such programs, time constraints permitted it to search only a few levels in the tree. (The exact number varied depending on the situation.) When it could search no deeper, it applied its static evaluation function to the board position and used that score to continue its search of the game tree. When it finished searching the tree and propagating the values backward, it had a score for the position represented by the root of the tree. It could then choose the best move and make it. But it also recorded the board position at the root of the tree and the backed up score that had just been computed for it. This situation is shown in Fig. 17.1 (a).

Now suppose that in a later game, the situation shown in Fig. 17.1 (b) were to arise. Instead of using the static evaluation function to compute a score for position A, the stored value for A can be used. This creates the effect of having searched an additional several ply since the stored value for A was computed by backing up values from exactly such a search.

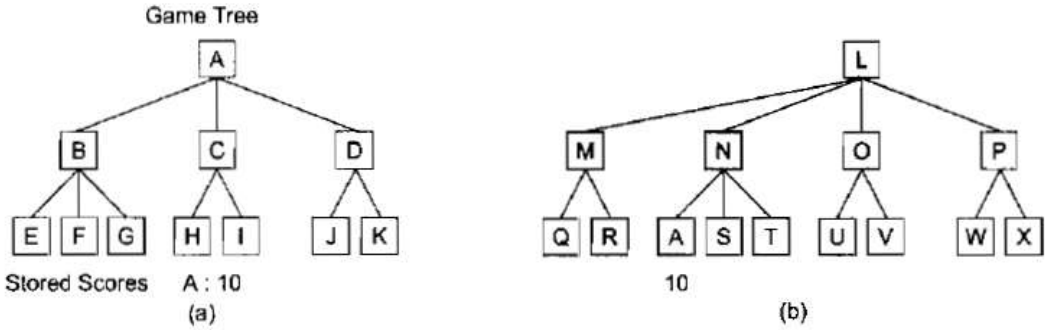

Fig.17.1 *Storing Backed-Up Values*

Rote learning of this sort is very simple. It does not appear to involve any sophisticated problem-solving capabilities. But even it shows the need for some capabilities that will become increasingly important in more complex learning systems. These capabilities include:

- *Organized Storage of Information*—In order for it to be faster to use a stored value than it would be to recompute it, there must be a way to access the appropriate stored value quickly. In Samuel's program, this was done by indexing board positions by a few important characteristics, such as the number of pieces. But as the complexity of the stored information increases, more sophisticated techniques are necessary.
- *Generalization*—The number of distinct objects that might potentially be stored can be very large. To keep the number of stored objects down to a manageable level, some kind of generalization is necessary. In Samuel's program, for example, the number of distinct objects that could be stored was equal to the number of different board positions that can arise in a game. Only a few simple forms of generalization were used in Samuel's program to cut down that number. All positions are stored as though White is to move. This cuts the number of stored positions in half. When possible, rotations along the diagonal are also combined. Again, though, as the complexity of the learning process increases, so too does the need for generalization.

At this point, we have begun to see one way in which learning is similar to other kinds of problem solving. Its success depends on a good organizational structure for its knowledge base.

17.3 LEARNING BY TAKING ADVICE

A computer can do very little without a program for it to run. When a programmer writes a series of instructions into a computer, a rudimentary kind of learning is taking place: The programmer is a sort of teacher, and the computer is a sort of student. After being programmed, the computer is now able to do something it previously could not. Executing the program may not be such a simple matter, however. Suppose the program is written

in a high-level language like LISP. Some interpreter or compiler must intervene to change the teacher's instructions into code that the machine can execute directly.

People process advice in an analogous way. In chess, the advice "fight for control of the center of the board" is useless unless the player can translate the advice into concrete moves and plans. A computer program might make use of the advice by adjusting its static evaluation function to include a factor based on the number of center squares attacked by its own pieces.

Mostow [1983] describes a program called FOO, which accepts advice for playing hearts, a card game. A human user first translates the advice from English into a representation that FOO can understand. For example, "Avoid taking points" becomes:

```
(avoid (take-points me) (trick))
```

FOO must *operationalize* this advice by turning it into an expression that contains concepts and actions FOO can use when playing the game of hearts. One strategy FOO can follow is to UNFOLD an expression by replacing some term by its definition. By UNFOLDing the definition of avoid, FOO comes up with:

```
(achieve (not (during (trick) (take-points me))))
```

FOO considers the advice to apply to the player called "me." Next, FOO UNFOLDs the definition of trick:

```
(achieve (not (during
                (scenario
                  (each pl (players) (play-card pl))
                  (take-trick (trick-winner)))
                (take-points me))))
```

In other words, the player should avoid taking points during the scenario consisting of (1) players playing cards and (2) one player taking the trick. FOO then uses *case analysis* to determine which steps could cause one to take points. It rules out step 1 on the basis that it knows of no intersection of the concepts take-points and play-card. But step 2 could affect taking points, so FOO UNFOLDs the definition of take-points:

```
(achieve (not (there-exists c1 (cards-played)
                (there-exists c2 (point-cards)
                  (during (take (trick-winner) c1)
                    (take me c2))))))
```

This advice says that the player should avoid taking point-cards during the process of the trick-winner taking the trick. The question for FOO now is: Under what conditions does (take me c2) occur during (take (trick-winner) c1)? By using a technique called *partial match*, FOO hypothesizes that points will be taken if me = trick-winner and c2 = c1. It transforms the advice into:

```
(achieve (not (and (have-points (cards-played))
                   (= (trick-winner) me))))
```

This means "Do not win a trick that has points." We have not traveled very far conceptually from "avoid taking points," but it is important to note that the current vocabulary is one that FOO can understand in terms of actually playing the game of hearts. Through a number of other transformations, FOO eventually settles on:

```
(achieve (>= (and (in-suit-led (card-of me))
                  (possible (trick-has-points)))
             (low (card-of me)))
```

In other words, when playing a card that is the same suit as the card that was played first, if the trick possibly contains points, then play a low card. At last, FOO has translated the rather vague advice "avoid taking points" into a specific, usable heuristic. FOO is able to play a better game of hearts after receiving this advice. A human can watch FOO play, detect new mistakes, and correct them through yet more advice, such as "play high cards when it is safe to do so." The ability to operationalize knowledge is critical for systems that learn from a teacher's advice. It is also an important component of explanation-based learning, another form of learning discussed in Section 17.6.

17.4 LEARNING IN PROBLEM-SOLVING

In the last section, we saw how a problem-solver could improve its performance by taking advice from a teacher. Can a program get better *without* the aid of a teacher? It can, by generalizing from its own experiences.

17.4.1 Learning by Parameter Adjustment

Many programs rely on an evaluation procedure that combines information from several sources into a single summary statistic. Game-playing programs do this in their static evaluation functions, in which a variety of factors, such as piece advantage and mobility, are combined into a single score reflecting the desirability of a particular board position. Pattern classification programs often combine several features to determine the correct category into which a given stimulus should be placed. In designing such programs, it is often difficult to know *a priori* how much weight should be attached to each feature being used. One way of finding the correct weights is to begin with some estimate of the correct settings and then to let the program modify the settings on the basis of its experience. Features that appear to be good predictors of overall success will have their weights increased, while those that do not will have their weights decreased, perhaps even to the point of being dropped entirely.

Samuel's checkers program [Samuel, 1963] exploited this kind of learning in addition to the rote learning described above, and it provides a good example of its use. As its static evaluation function, the program used a polynomial of the form

$$c_1t_1 + c_2t_2 + \ldots + c_{16}t_{16}$$

The t terms are the values of the sixteen features that contribute to the evaluation. The c terms are the coefficients (weights) that are attached to each of these values. As learning progresses, the c values will change.

The most important question in the design of a learning program based on parameter adjustment is "When should the value of a coefficient be increased and when should it be decreased?" The second question to be answered is then "By how much should the value be changed?" The simple answer to the first question is that the coefficients of terms that predicted the final outcome accurately should be increased, while the coefficients of poor predictors should be decreased. In some domains, this is easy to do. If a pattern classification program uses its evaluation function to classify an input and it gets the right answer, then all the terms that predicted that answer should have their weights increased. But in game-playing programs, the problem is more difficult. The program does not get any concrete feedback from individual moves. It does not find out for sure until the end of the game whether it has won. But many moves have contributed to that final outcome. Even if the program wins, it may have made some bad moves along the way. The problem of appropriately assigning responsibility to each of the steps that led to a single outcome is known as the *credit assignment problem.*

Samuel's program exploits one technique, albeit imperfect, for solving this problem. Assume that the initial values chosen for the coefficients are good enough that the total evaluation function produces values

that are fairly reasonable measures of the correct score even if they are not as accurate as we hope to get them. Then this evaluation function can be used to provide feedback to itself. Move sequences that lead to positions with higher values can be considered good (and the terms in the evaluation function that suggested them can be reinforced).

Because of the limitations of this approach, however, Samuel's program did two other things, one of which provided an additional test that progress was being made and the other of which generated additional nudges to keep the process out of a rut:

- When the program was in learning mode, it played against another copy of itself. Only one of the copies altered its scoring function during the game; the other remained fixed. At the end of the game, if the copy with the modified function won, then the modified function was accepted. Otherwise, the old one was retained. If, however, this happened very many times, then some drastic change was made to the function in an attempt to get the process going in a more profitable direction.
- Periodically, one term in the scoring function' was eliminated and replaced by another. This was possible because, although the program used only sixteen features at any one time, it actually knew about thirty-eight. This replacement differed from the rest of the learning procedure since it created a sudden change in the scoring function rather than a gradual shift in its weights.

This process of learning by successive modifications to the weights of terms in a scoring function has many limitations, mostly arising out of its lack of exploitation of any knowledge about the structure of the problem with which it is dealing and the logical relationships among the problem's components. In addition, because the learning procedure is a variety of hill climbing, it suffers from the same difficulties as do other hill-climbing programs. Parameter adjustment is certainly not a solution to the overall learning problem. But it is often a useful technique, either in situations where very little additional knowledge is available or in programs in which it is combined with more knowledge-intensive methods. We have more to say about this type of learning in Chapter 18.

17.4.2 Learning with Macro-Operators

We saw in Section 17.2 how rote learning was used in the context of a checker-playing program. Similar techniques can be used in more general problem-solving programs. The idea is the same: to avoid expensive recomputation. For example, suppose you are faced with the problem of getting to the downtown post office. Your solution may involve getting in your car, starting it, and driving along a certain route. Substantial planning may go into choosing the appropriate route, but you need not plan about how to go about starting your car. You are free to treat START-CAR as an atomic action, even though it really consists of several actions: sitting down, adjusting the mirror, inserting the key, and turning the key. Sequences of actions that can be treated as a whole are called *macro-operators.*

Macro-operators were used in the early problem-solving system STRIPS [Fikes and Nilsson, 1971; Fikes *et al.,* 1972]. We discussed the operator and goal structures of STRIPS in Section 13.2, but STRIPS also has a learning component. After each problem-solving episode, the learning component takes the computed plan and stores it away as a macro-operator, or MACROP. A MACROP is just like a regular operator except that it consists of a sequence of actions, not just a single one. A MACROP's preconditions are the initial conditions of the problem just solved, and its postconditions correspond to the goal just achieved. In its simplest form, the caching of previously computed plans is similar to rote learning.

Suppose we are given an initial blocks world situation in which ON(C, B) and ON(A, Table) are both true. STRIPS can achieve the goal ON(A, B) by devising a plan with the four steps UNSTACK(C, B), PUTDOWN(C), PICKUP(A), STA•K(A, B). STRIPS now builds a MACROP with preconditions ON(C, B), ON(A, Table) and postconditions ON(C, Table), ON(A, B). The body of the MACROP consists of the four

steps just mentioned. In future planning, STRIPS is free to use this complex macro-operator just as it would use any other operator.

But rarely will STRIPS see the exact same problem twice. New problems will differ from previous problems. We would still like the problem solver to make efficient use of the knowledge it gained from its previous experiences. By *generalizing* MACROPs before storing them, STRIPS is able to accomplish this. The simplest idea for generalization is to replace all of the constants in the macro-operator by variables. Instead of storing the MACROP described in the previous paragraph, STRIPS can generalize the plan to consist of the steps UNSTACK(x_1, x_2), PUTDOWN(x_1), PICKUP(x_3), STACK(x_3 x_2), where x_1, x_2, and x_3 are variables. This plan can then be stored with preconditions ON(x_1, x_2), ON(x_3, Table) and postconditionsON(x_1, Table), ON(x_2, x_3). Such a MACROP can now apply in a variety of situations.

Generalization is not so easy, however. Sometimes constants must retain their specific values. Suppose our domain included an operator called STACK-ON-B(x), with preconditions that both x and B be clear, and with postcondition ON(x, B). Consider the same problem as above:

start: ON(C, B) goal: ON(A, B)

STRIPS might come up with the plan UNSTACK(C, B), PUTDOWN(C), STACK-ON-B(A). Let's generalize this plan and store it as a MACROP. The precondition becomes ON(x_3, x_2), the postcondition becomes ON(x_1, x_2), and the plan itself becomes UNSTACK(x_3, x_2), PUTDOWN(x_3), STACK-ON-B(x_1). Now, suppose we encounter a slightly different problem:

start: ON(E, C)
ON(D, B)

goal: ON(A, C)

The generalized MACROP we just stored seems well-suited to solving this problem if we let x_1 = A, x_2 = C, and x_3 = E. Its preconditions are satisfied, so we construct the plan UNSTACK(E, C), PUTDOWN(E), STACK-ON-B(A). But this plan does not work. The problem is that the postcondition of the MACROP is overgeneralized. This operation is only useful for stacking blocks onto B, which is not what we need in this new example. In this case, this difficulty will be discovered when the last step is attempted. Although we cleared C, which is where we wanted to put A, we failed to clear B, which is were the MACROP is going to try to put it. Since B is not clear, STACK-ON-B cannot be executed. If B had happened to be clear, the MACROP would have executed to completion, hut it would not have accomplished the stated goal.

In reality, STRIPS uses a more complex generalization procedure. First, all constants are replaced by variables. Then, for each operator in the parameterized plan, STRIPS revaluates its preconditions. In our example, the preconditions of steps 1 and 2 are satisfied, but the only way to ensure that B is clear for step 3 is to assume that block x_2, which was cleared by the UNSTACK operator, is actually block B. Through "re-proving" that the generalized plan works, STRIPS locates constraints of this kind.

More recent work on macro-operators appears in Korf [1985b]. It turns out that the set of problems for which macro-operators are critical are exactly those problems with *nonserializable subgoals*. Nonserializability means that working on one subgoal will necessarily interfere with the previous solution to another subgoal. Recall that we discussed such problems in connection with nonlinear planning (Section 13.5). Macro-operators can be useful in such cases, since one macro-operator can produce a small global change in the world, even though the individual operators that make it up produce many undesirable local changes.

For example, consider the 8-puzzle. Once a program has correctly placed the first four tiles, it is difficult to place the fifth tile without disturbing the first four. Because disturbing previously solved subgoals is detected as a bad thing by heuristic scoring functions, it is strongly resisted. For many problems, including the 8-puzzle and Rubik's cube, weak methods based on heuristic scoring are therefore insufficient. Hence, we either need domain-specific knowledge, or else a new weak method. Fortunately, we can *learn* the domain-specific knowledge we need in the form of macro-operators. Thus, macro-operators can be viewed as a weak method for learning. In the 8-puzzle, for example, we might have a macro—a complex, prestored sequence of operators—for placing the fifth tile without disturbing any of the first four tiles externally (although in fact they are disturbed within the macro itself). Korf [1985b] gives an algorithm for learning a complete set of macro-operators. This approach contrasts with STRIPS, which learned its MACROPs gradually, from experience. Korf's algorithm runs in time proportional to the time it takes to solve a single problem without macro-operators.

17.4.3 Learning by Chunking

Chunking is a process similar in flavor to macro-operators. The idea of chunking comes from the psychological literature on memory and problem solving. Its computational basis is in production systems, of the type studied in Chapter 6. Recall that in that chapter we described the SOAR system and discussed its use of control knowledge. SOAR also exploits chunking [Laird *et al.*, 1986] so that its performance can increase with experience. In fact, the designers of SOAR hypothesize that chunking is a universal learning method, i.e., it can account for all types of learning in intelligent systems.

SOAR solves problems by firing productions, which are stored in long-term memory. Some of those firings turn out to be more useful than others. When SOAR detects a useful sequence of production firings, it creates a chunk, which is essentially a large production that does the work of an entire sequence of smaller ones. As in MACROPs, chunks are generalized before they are stored.

Recall from Section 6.5 that SOAR is a uniform processing architecture. Problems like choosing which subgoals to tackle and which operators to try (i.e., search control problems) are solved with the same mechanisms as problems in the original problem space. Because the problem-solving is uniform, chunking can be used to learn general search control knowledge in addition to operator sequences. For example, if SOAR tries several different operators, but only one leads to a useful path in the search space, then SOAR builds productions that help it choose operators more wisely in the future.

SOAR has used chunking to replicate the macro-operator results described in the last section. In solving the 8-puzzle, for example, SOAR learns how to place a given tile without permanently disturbing the previously placed tiles. Given the way that SOAR learns, several chunks may encode a single macro-operator, and one chunk may participate in a number of macro sequences. Chunks are generally applicable toward any goal state. This contrasts with macro tables, which are structured toward reaching a particular goal state from any initial state. Also, chunking emphasizes how learning can occur during problem-solving, while macro tables are usually built during a preprocessing stage. As a result, SOAR is able to learn within trials as well as across trials. Chunks learned during the initial stages of solving a problem are applicable in the later stages of the same problem-solving episode. After a solution is found, the chunks remain in memory, ready-for-use in the next problem.

The price that SOAR pays for this generality and flexibility is speed. At present, chunking is inadequate for duplicating the contents of large, directly-computed macro-operator tables.

17.4.4 The Utility Problem

PRODIGY [Minton *et al.*, 1989], which we described in Section 6.5, also acquires control knowledge automatically. PRODIGY employs several learning mechanisms. One mechanism uses *explanation-based learning* (EBL), a learning method we discuss in Section 17.6. PRODIGY can examine a trace of its own problem-solving behavior and try to explain why certain paths failed. The program; uses those explanations

to formulate control rules that help the problem solver avoid those paths in the future. So while SOAR learns primarily from examples of successful problem solving, PRODIGY also learns from its failures.

A major contribution of the work on EBL in PRODIGY [Minton, 1988] was the identification of the *utility problem* in learning systems. While new search control knowledge can be of great benefit in solving future problems efficiently, there are also some drawbacks. The learned control rules can take up large amounts of memory and the search program must take the time to consider each rule at each step during problem solving. Considering a control rule amounts to seeing if its postconditions are desirable and seeing if its preconditions are satisfied This is a time-consuming process. So while learned rules may reduce problem-solving time by directing the search more carefully, they may also increase problem-solving time by forcing the problem solver to consider them. If we only want to minimize the number of node expansions in the search space, then the more control rules we learn, the better. But if we want to minimize the total CPU time required to solve a problem, we must consider this trade-off.

PRODIGY maintains a utility measure for each control rule. This measure takes into account the average savings provided by the rule, the frequency of its application, and the cost of matching it. If a proposed rule has a negative utility, it is discarded (or "forgotten"). If not, it is placed in long-term memory with the other rules. It is then monitored during subsequent problem solving. If its utility falls, the rule is discarded. Empirical experiments have demonstrated the effectiveness of keeping only those control rules with high utility. Utility considerations apply to a wide range of learning systems. For example, for a discussion of how to deal with large, expensive chunks in SOAR, see Tambe and Rosenbloom [1989].

17.5 LEARNING FROM EXAMPLES: INDUCTION

Classification is the process of assigning to a particular input, the name of a class to which it belongs. The classes from which the classification procedure can choose can be described in a variety of ways. Their definition will depend on the use to which they will be put.

Classification is an important component of many problem-solving tasks. In its simplest form, it is presented as a straightforward recognition task. An example of this is the question "What letter of the alphabet is this?" But often classification is embedded inside another operation. To see how this can happen, consider a problem-solving system that contains the following production rule:

```
If:   the current goal is to get from place A to place B, and
      there is a WALL separating the two places
then: look for a DOORWAY in the WALL and go through it.
```

To use this rule successfully, the system's matching routine must be able to identify an object as a wall. Without this, the rule can never be invoked. Then, to apply the rule, the system must be able to recognize a doorway.

Before classification can be done, the classes it will use must be defined. This can be done in a variety of ways, including:

- Isolate a set of features that are relevant to the task domain. Define each class by a weighted sum of values of these features. Each class is then defined by a scoring function that looks very similar to the scoring functions often used in other situations, such as game playing. Such a function has the form:

  $$c_1t_1 + c_2t_2 + c_3t_3 + \ldots$$

  Each t corresponds to a value of a relevant parameter, and each c represents the weight to be attached to the corresponding t. Negative weights can be used to indicate features whose presence usually constitutes negative evidence for a given class.

For example, if the task is weather prediction, the parameters can be such measurements as rainfall and location of cold fronts. Different functions can be written to combine these parameters to predict sunny, cloudy, rainy, or snowy weather.

- Isolate a set of features that are relevant to the task domain. Define each class as a structure composed of those features.
  For example, if the task is to identify animals, the body of each type of animal can be stored as a structure, with various features representing such things as color, length of neck, and feathers.

There are advantages and disadvantages to each of these general approaches. The statistical approach taken by the first scheme presented here is often more efficient than the structural approach taken by the second. But the second is more flexible and more extensible.

Regardless of the way that classes are to be described, it is often difficult to construct, by hand, good class definitions. This is particularly true in domains that are not well understood or that change rapidly. Thus the idea of producing a classification program that can evolve its own class definitions is appealing. This task of constructing class definitions is called *concept learning*, or *induction*. The techniques used for this task must, of course, depend on the way that classes (concepts) are described. If classes are described by scoring functions, then concept learning can be done using the technique of coefficient adjustment described in Section 17.4.1. If, however, we want to define classes structurally, some other technique for learning class definitions is necessary. In this section, we present three such techniques.

17.5.1 Winston's Learning Program

Winston [1975] describes an early structural concept learning program. This program operated in a simple blocks world domain. Its goal was to construct representations of the definitions of concepts in the blocks domain. For example, it learned the concepts *House*, *Tent*, and *Arch* shown in Fig. 17.2. The figure also shows an example of a near miss for each concept. A *near miss* is an object that is not an instance of the concept in question but that is very similar to such instances.

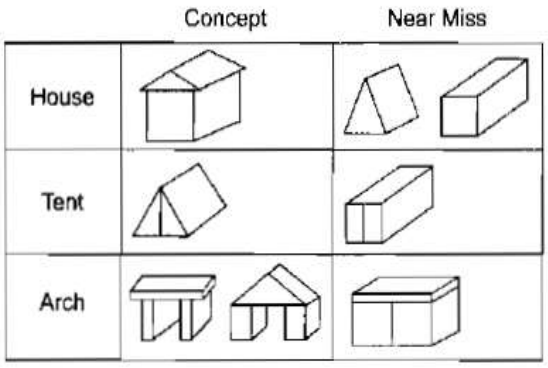

Fig. 17.2 *Some Blocks World Concepts*

The program started with a line drawing of a blocks world structure. It used procedures such as the one described in Section 14.3 to analyze the drawing and construct a semantic net representation of the structural description of the object(s) .This structural description was then provided as input to the learning program. An example of such a structural description for the *House* of Fig. 17.2 is shown in Fig. 17.3(a). Node A represents the entire structure, which is composed of two parts: node B, a *Wedge*, and node C, a *Brick*. Figures 17.3(b) and 17.3(c) show descriptions of the two *Arch* structures of Fig. 17.2. These descriptions are identical except for the types of the objects on the top; one is a *Brick* while the other is a *Wedge*. Notice that the two supporting objects are related not only by *left-of* and *right-of* links, but also by a *does-not-marry* link, which says that the two objects do not *marry*. Two objects *marry* if they have faces that touch and they have a common edge. The *marry* relation is critical in the definition of an *Arch*. It is the difference between the first arch structure and the near miss arch structure shown in Fig. 17.2.

The basic approach that Winston's program took to the problem of concept formation can be described as follows:

1. Begin with a structural description of one known instance of the concept. Call that description the concept definition.

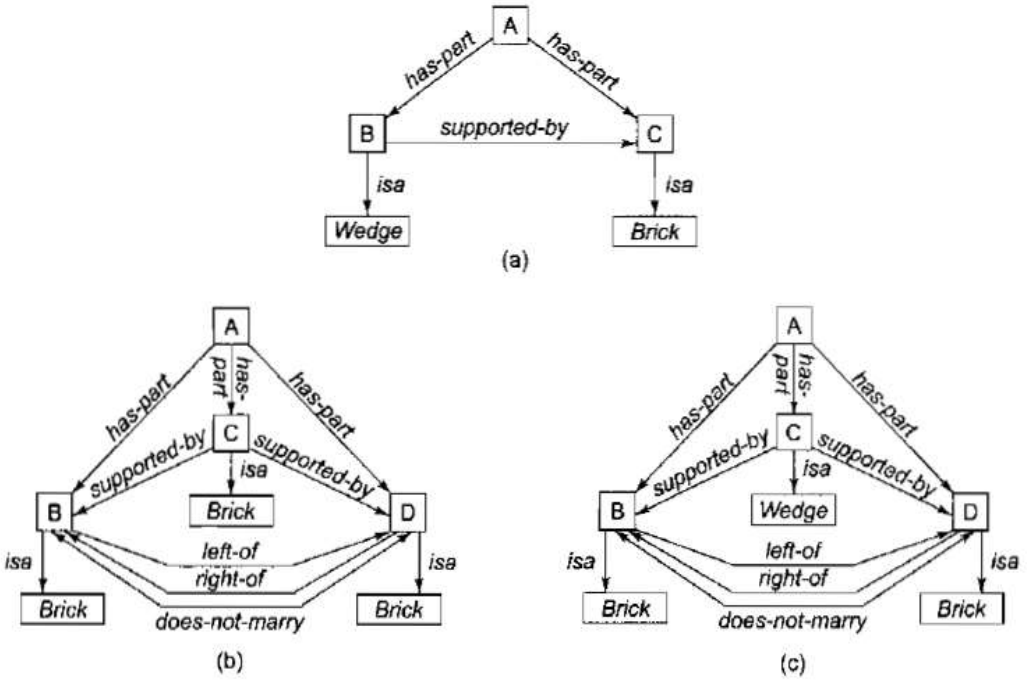

Fig. 17.3 *Structural Descriptions*

2. Examine descriptions of other known instances of the concept. Generalize the definition to include them.
3. Examine descriptions of near misses of the concept. Restrict the definition to exclude these.

Steps 2 and 3 of this procedure can be interleaved.

Steps 2 and 3 of this procedure rely heavily on a comparison process by which similarities and differences between structures can be detected. This process must function in much the same way as does any other matching process, such as one to determine whether a given production rule can be applied to a particular problem state. Because differences as well as similarities must be found, the procedure must perform not just literal but also approximate matching. The output of the comparison procedure is a skeleton structure describing the commonalities between the two input structures. It is annotated with a set of comparison notes that describe specific similarities and differences between the inputs.

To see how this approach works, we trace it through the process of learning what an arch is. Suppose that the arch description of Fig. 17.3(b) is presented first. It then becomes the definition of the concept *Arch*. Then suppose that the arch description of Fig. 17.3(c) is presented. The comparison routine will return a structure similar to the two input structures except that it will note that the objects represented by the nodes labeled C are not identical. This structure is shown as Fig. 17.4. The *c-note* link from node C describes

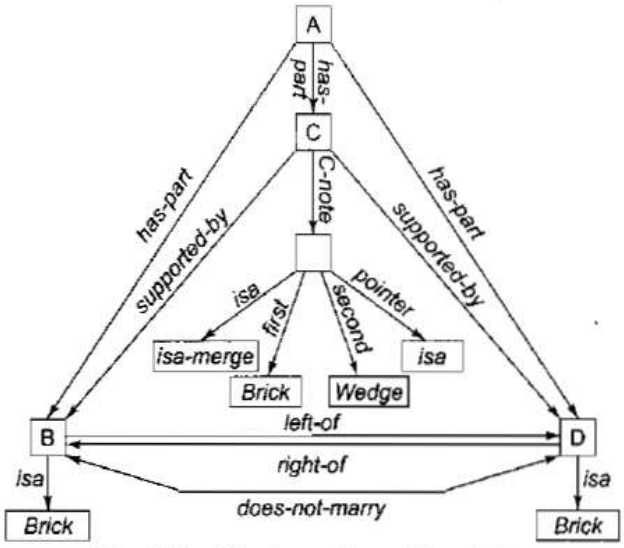

Fig. 17.4 *The Comparison of Two Arches*

the difference found by the comparison routine. It notes that the difference occurred in the *isa* link, and that in the first structure the *isa* link pointed to *Brick*, and in the second it pointed to *Wedge*. It also notes that if we were to follow *isa* links from *Brick* and *Wedge*, these links would eventually merge. At this point, a new description of the concept *Arch* can be generated. This description could say simply that node C must be either a *Brick* or a *Wedge*. But since this particular disjunction has no previously known significance, it is probably better to trace up the *isa* hierarchies of *Brick* and *Wedge* until they merge. Assuming that that happens at the node *Object*, the *Arch* definition shown in Fig. 17.5 can be built.

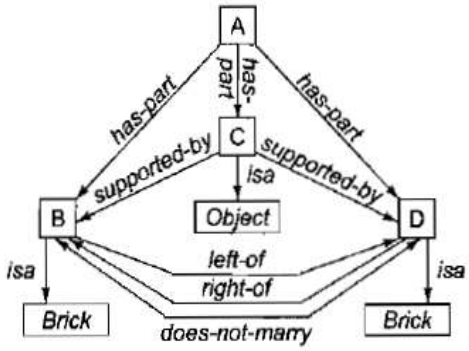

Fig. 17.5 *The Arch Description after Two Examples*

Next, suppose that the near miss arch shown in Fig. 17.2 is presented. This time, the comparison routine will note that the only difference between the current definition and the near miss is in the *does-not-marry* link between nodes *B* and *D*. But since this is a near miss, we do not want to broaden the definition to include it. Instead, we want to restrict the definition so that it is specifically excluded. To do this, we modify the link *does-not-marry*, which may simply be recording something that has happened by chance to be true of the small number of examples that have been presented. It must now say *must-not-marry*. The *Arch* description at this point is shown in Fig. 17.6. Actually, *must-not-marry* should not be a completely new link. There must be some structure among link types to reflect the relationship between *marry*, *does-not-marry*, and *must-not-marry*.

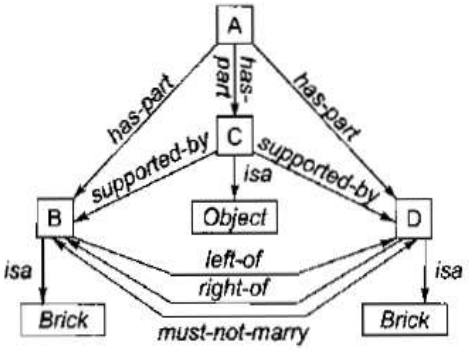

Fig. 17.6 *The Arch Description after a Near Miss*

Notice how the problem-solving and knowledge representation techniques we covered in earlier chapters are brought to bear on the problem of learning. Semantic networks were used to describe block structures, and an *isa* hierarchy was used to de-scribe relationships among already known objects. A matching process was used to detect similarities and differences between structures, and hill climbing allowed the program to evolve a more and more accurate concept definition.

This approach to structural concept learning is not without its problems. One major problem is that a teacher must guide the learning program through a carefully chosen sequence of examples. In the next section, we explore a learning technique that is insensitive to the order in which examples are presented.

17.5.2 Version Spaces

Mitchell [1977; 1978] describes another approach to concept learning called *version spaces*. The goal is the same: to produce a description that is consistent with all positive examples but no negative examples in the training set. But while Winston's system did this by evolving a single concept description, version spaces work by maintaining a *set* of possible descriptions and evolving that set as new examples and near misses are presented. As in the previous section, we need some sort of representation language for examples so that we can describe exactly what the system sees in an example. For now we assume a simple frame-based language; although version spaces can be constructed for more general representation languages. Consider Fig. 17.7, a frame representing an individual car.

Car023	
origin :	Japan
manufacturer :	Honda
color :	Blue
decade :	1970
type :	Economy

Fig. 17.7 *An Example of the Concept Car*

Now, suppose that each slot may contain only the discrete values shown in Fig. 17.8. The choice of features and values is called the *bias* of the learning system. By being embedded in a particular program and by using particular representations, every learning system is biased, because it learns some things more easily than others. In our example, the bias is fairly simple — e.g., we can learn concepts that have to do with car manufacturers, but not car owners. In more complex systems, the bias is less obvious. A clear statement of the bias of a learning system is very important to its evaluation.

origin	$\in$	{Japan, USA, Britain, Germany, Italy}
manufacturer	$\in$	{Honda, Toyota, Ford, Chrysler, Jaguar, BMW, Fiat}
color	$\in$	{Blue, Green, Red, White}
decade	$\in$	{1950, 1960, 1970, 1980, 1990, 2000}
type	$\in$	{Economy, Luxury, Sports}

Fig. 17.8 *Representation Language for Cars*

Concept descriptions, as well as training examples, can be stated in terms of these slots and values. For example, the concept "Japanese economy car" can be represented as in Fig. 17.9. The names x_1, x_2, and x_3 are variables. The presence of x_2, for example, indicates that the color of a car is not relevant to whether the car is a Japanese economy car. Now the learning problem is: Given a representation language such as in Fig. 17.8, and given positive and negative training examples such as those in Fig. 17.7, how can we produce a concept description such as that in Fig. 17.9 that is consistent with all the training examples?

origin :	Japan
manufacturer :	x_1
color :	x_2
decade :	x_3
type :	Economy

Fig. 17.9 *The Concept "Japanese economy car"*

Before we proceed to the version space algorithm, we should make some observations about the representation. Some descriptions are more general than others. For example, the description in Fig. 17.9 is more general than the one in Fig. 17.7. In fact, the representation language defines a partial ordering of descriptions. A portion of that partial ordering is shown in Fig. 17.10.

The entire partial ordering is called the *concept space*, and can be depicted as in Fig. 17.11. At the top of the concept space is the null description, consisting only of variables, and at the bottom are all the possible training instances, which contain no variables. Before we receive any training examples, we know that the target concept lies somewhere in the concept space. For example, if every possible description is an instance of the intended concept, then the null description is the concept definition since it matches everything. On the other hand, if the target concept includes only a single example, then one of the descriptions at the bottom of the concept space is the desired concept definition. Most target concepts, of course, lie somewhere in between these two extremes.

As we process training examples, we want to refine our notion of where the target concept might lie. Our current hypothesis can be represented as a subset of the concept space called the *version space*. The version space is the largest collection of descriptions that is consistent with all the training examples seen so far.

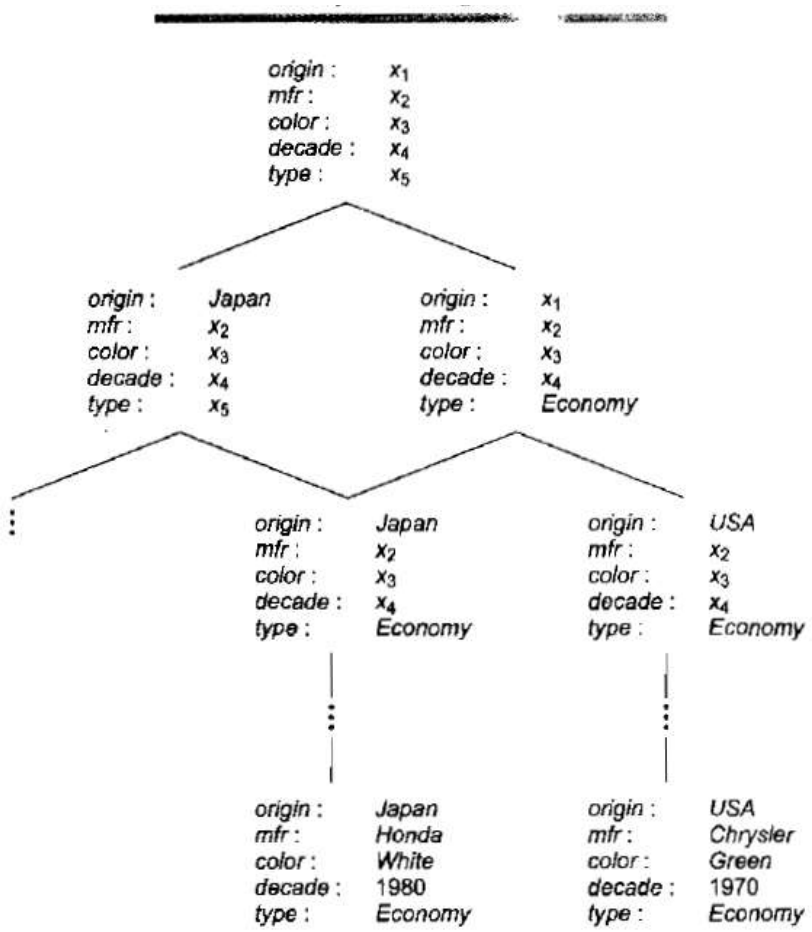

Fig. 17.10 *Partial Ordering of Concepts Specified by the Representation Language*

How can we represent the version space? The version space is simply a set of descriptions, so an initial idea is to keep an explicit list of those descriptions. Unfortunately, the number of descriptions in the concept space is exponential in the number of features and values. So enumerating them is prohibitive. However, it turns out that the version space has a concise representation. It consists of two subsets of the concept space. One subset, called G contains the most *general* descriptions consistent with the training examples seen so far; the other subset, called S, contains the most *specific* descriptions consistent with the training examples. The version space is the set of all descriptions that lie between some element of G and some element of S in the partial order of the concept space.

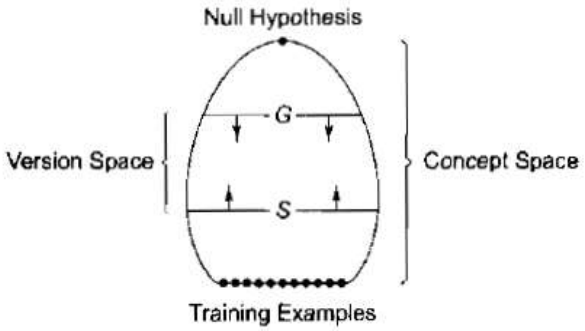

Fig. 17.11 *Concept and Version Spaces*

This representation of the version space is not only efficient for storage, but also for modification. Intuitively, each time we receive a positive training example, we want to make the S set more general. Negative training examples serve to make the G set more specific. If the S and G sets converge, our range of hypotheses will narrow to a single concept description. The algorithm for narrowing the version space is called the *candidate elimination algorithm.*