

Part III

Advanced Topics

Chapter 12

Game Playing

12.1 Overview

Games hold an inexplicable fascination for many people, and the notion that computers might play games has existed at least as long as computers. Charles Babbage, the nineteenth-century computer architect, thought about programming his Analytical Engine to play chess and later of building a machine to play tic-tac-toe [Bowden, 1953]. Two of the pioneers of the science of information and computing contributed to the fledgling computer game-playing literature. Claude Shannon [1950] wrote a paper in which he described mechanisms that could be used in a program to play chess. A few years later, Alan Turing described a chess-playing program, although he never built it. (For a description, see Bowden [1953].) By the early 1960s, Arthur Samuel had succeeded in building the first significant, operational game-playing program. His program played checkers and, in addition to simply playing the game, could learn from its mistakes and improve its performance [Samuel, 1963].

There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine. Unfortunately, the second is not true for any but the simplest games. For example, consider chess.

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.

Thus it is clear that a program that simply does a straightforward search of the game tree will not be able to select even its first move during the lifetime of its opponent. Some kind of heuristic search procedure is necessary.

One way of looking at all the search procedures we have discussed is that they are essentially generate-and-test procedures in which the testing is done after varying amounts of work by the generator. At one extreme, the generator generates entire proposed solutions, which the tester then evaluates. At the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the tester and the most promising one is chosen. Looked at this way, it is clear that to improve the effectiveness of a search-based problem-solving program two things can be done:

- Improve the generate procedure so that only good moves (or paths) are generated.
- Improve the test procedure so that the best moves (or paths) will be recognized and explored first.

In game-playing programs, it is particularly important that both these things be done. Consider again the problem of playing chess. On the average, there are about 35 legal moves available at each turn. If we use a simple legal-move generator, then the test procedure (which probably uses some combination of search and a heuristic evaluation function) will have to look at each of them. Because the test procedure must look at so many possibilities, it must be fast. So it probably cannot do a very accurate job. Suppose, on the other hand, that instead of a legal-move generator, we use a *plausible-move generator* in which only some small number of promising moves are generated. As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise. (So, for example, it is extremely important in programs that play the game of go [Benson *et al.*, 1979].) With a more selective move generator, the test procedure can afford to spend more time evaluating each of the moves it is given so it can produce a more reliable result. Thus by incorporating heuristic knowledge into both the generator and the tester, the performance of the overall system can be improved.

Of course, in game playing, as in other problem domains, search is not the only available technique. In some games, there are at least some times when more direct techniques are appropriate. For example, in chess, both openings and endgames are often highly stylized, so they are best played by table lookup into a database of stored patterns. To play an entire game then, we need to combine search-oriented and nonsearch-oriented techniques.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached. In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for interesting games such as chess, it is not usually possible, even with a good plausible-move generator, to search until a goal state is found. The depth of the resulting tree (or graph) and its branching factor are too great. In the amount of time available, it is usually possible to search a tree only ten or twenty moves (called *ply* in the game-playing literature) deep. Then, in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous. This is done using a *static evaluation function*, which uses whatever information it has to evaluate

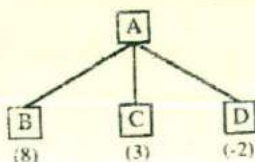


Figure 12.1: One-Ply Search

so that the best next move can be chosen. But because of their adversarial nature, this procedure is inadequate for two-person games such as chess. As values are passed back up, different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses. There are several ways that this can be done. The most commonly used method is the *minimax* procedure, which is described in the next section. An alternative approach is the *B** algorithm [Berliner, 1979a], which works on both standard problem-solving trees and on game trees.

12.2 The Minimax Search Procedure

The *minimax search procedure* is a depth-first, depth-limited search procedure. It was described briefly in Section 1.3.1. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evaluation of it. The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to *maximize* the value of the static evaluation function of the next board position.

An example of this operation is shown in Figure 12.1. It assumes a static evaluation function that returns values ranging from -10 to 10 , with 10 indicating a win for us, -10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply. This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed. So we would like to look ahead to see what will happen to each of the new game positions at the next move which will be made by the opponent. Instead of applying the static evaluation function to each of the positions that we just generated, we apply the plausible-move generator, generating a set of successor positions for each position. If we wanted to stop here, at two-ply lookahead, we could apply the static evaluation function to each of these positions, as shown in Figure 12.2.

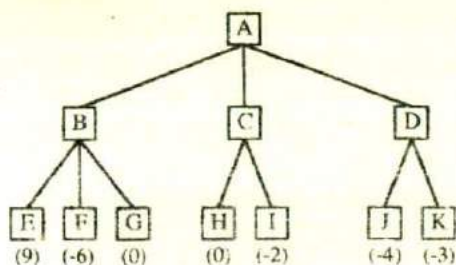


Figure 12.2: Two-Ply Search

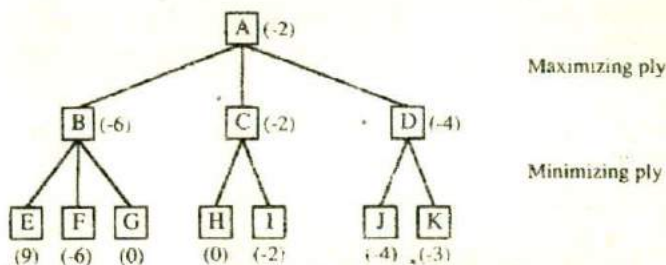


Figure 12.3: Backing Up the Values of a Two-Ply Search

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level. Suppose we made move B. Then the opponent must choose among moves E, F, and G. The opponent's goal is to *minimize* the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 12.3 shows the result of propagating the new values up the tree. At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.

Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information we have available, is C, since there is nothing the opponent can do from there to produce a value worse than -2 . This process can be repeated for as many ply as time allows, and the more accurate evaluations that are produced can be used to choose the correct move at the top level. The alternation of maximizing and minimizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and gives this method the name minimax.

Having described informally the operation of the minimax procedure, we now

describe it precisely. It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played:

1. *MOVEGEN(Position, Player)*—The plausible-move generator, which returns a list of nodes representing the moves that can be made by *Player* in *Position*. We call the two players *PLAYER-ONE* and *PLAYER-TWO*; in a chess program, we might use the names *BLACK* and *WHITE* instead.
2. *STATIC(Position, Player)*—The static evaluation function, which returns a number representing the goodness of *Position* from the standpoint of *Player*.²

As with any recursive program, a critical issue in the design of the *MINIMAX* procedure is when to stop the recursion and simply call the static evaluation function. There are a variety of factors that may influence this decision. They include:

- Has one side won?
- How many ply have we already explored?
- How promising is this path?
- How much time is left?
- How stable is the configuration?

For the general *MINIMAX* procedure discussed here, we appeal to a function, *DEEP-ENOUGH*, which is assumed to evaluate all of these factors and to return *TRUE* if the search should be stopped at the current level and *FALSE* otherwise. Our simple implementation of *DEEP-ENOUGH* will take two parameters, *Position* and *Depth*. It will ignore its *Position* parameter and simply return *TRUE* if its *Depth* parameter exceeds a constant cutoff value.

One problem that arises in defining *MINIMAX* as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

We assume that *MINIMAX* returns a structure containing both results and that we have two functions, *VALUE* and *PATH*, that extract the separate components.

Since we define the *MINIMAX* procedure as a recursive function, we must also specify how it is to be called initially. It takes three parameters, a board position, the current depth of the search, and the player to move. So the initial call to compute the best move from the position *CURRENT* should be

²This may be a bit confusing, but it need not be. In all the examples in this chapter so far (including Figures 12.2 and 12.3), we have assumed that all values of *STATIC* are from the point of view of the initial (maximizing) player. It turns out to be easier when defining the algorithm, though, to let *STATIC* alternate perspectives so that we do not need to write separate procedures for the two levels. It is easy to modify *STATIC* for this purpose; we merely compute the value of *Position* from *PLAYER-ONE*'s perspective, then invert the value if *STATIC*'s parameter is *PLAYER-TWO*.

MINIMAX (CURRENT, 0, PLAYER-ONE)

if PLAYER-ONE is to move, or

MINIMAX (CURRENT, 0, PLAYER-TWO)

if PLAYER-TWO is to move.

Algorithm: MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(*Position*, *Depth*) then return the structure

VALUE = STATIC(*Position*, *Player*);

PATH = nil

This indicates that there is no path from this node and that its value is that determined by the static evaluation function.

2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(*Position* *Player*) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.

Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS.

For each element SUCC of SUCCESSORS, do the following:

- (a) Set RESULT-SUCC to

MINIMAX(SUCC, *Depth* + 1, OPPOSITE(*Player*))³

This recursive call to MINIMAX will actually carry out the exploration of SUCC.

- (b) Set NEW-VALUE to -VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
- (c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
 - i. Set BEST-SCORE to NEW-VALUE.
 - ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure

```
VALUE = BEST-SCORE
PATH = BEST-PATH
```

When the initial call to MINIMAX returns, the best move from CURRENT is the first element on PATH. To see how this procedure works, you should trace its execution for the game tree shown in Figure 12.2.

The MINIMAX procedure just described is very simple. But its performance can be improved significantly with a few refinements. Some of these are described in the next few sections.

12.3 Adding Alpha-Beta Cutoffs

Recall that the minimax procedure is a depth-first process. One path is explored as far as time allows, the static evaluation function is applied to the game positions at the last step of the path, and the value can then be passed up the path one level at a time. One of the good things about depth-first procedures is that their efficiency can often be improved by using branch-and-bound techniques in which partial solutions that are clearly worse than known solutions can be abandoned early. We described a straightforward application of this technique to the traveling salesman problem in Section 2.2.1. For that problem, all that was required was storage of the length of the best path found so far. If a later partial path outgrew that bound, it was abandoned. But just as it was necessary to modify our search procedure slightly to handle both maximizing and minimizing players, it is also necessary to modify the branch-and-bound strategy to include two bounds, one for each of the players. This modified strategy is called *alpha-beta pruning*. It requires the maintenance of two threshold values, one representing a lower bound on the value that a maximizing node may ultimately be assigned (we call this *alpha*) and another representing an upper bound on the value that a minimizing node may be assigned (this we call *beta*).

To see how the alpha-beta procedure works, consider the example shown in Figure 12.4.⁴ After examining node F, we know that the opponent is guaranteed a score of -5 or less at C (since the opponent is the minimizing player). But we also know that we are guaranteed a score of 3 or greater at node A, which we can achieve if we move to B. Any other move that produces a score of less than 3 is worse than the move to B, and we can ignore it. After examining only F, we are sure that a move to C is worse (it will be less than or equal to -5) regardless of the score of node G. Thus we need not bother to explore node G at all. Of course, cutting out one node may not appear to justify the expense of keeping track of the limits and checking them, but if we were exploring this tree to six ply, then we would have eliminated not a single node but an entire tree three ply deep.

To see how the two thresholds, alpha and beta, can both be used, consider the example shown in Figure 12.5. In searching this tree, the entire subtree headed by B is searched, and we discover that at A we can expect a score of at least 3. When this

⁴In this figure, we return to the use of a single STATIC function from the point of view of the maximizing player.

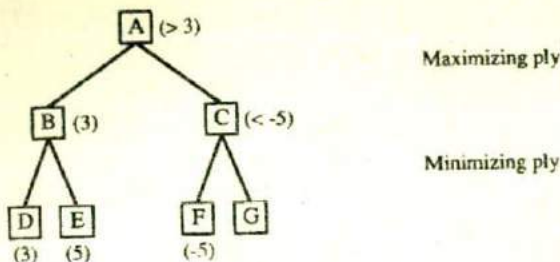


Figure 12.4: An Alpha Cutoff

alpha value is passed down to F, it will enable us to skip the exploration of L. Let's see why. After K is examined, we see that I is guaranteed a maximum score of 0, which means that F is guaranteed a minimum of 0. But this is less than alpha's value of 3, so no more branches of I need be considered. The maximizing player already knows not to choose to move to C and then to I since, if that move is made, the resulting score will be no better than 0 and a score of 3 can be achieved by moving to B instead. Now let's see how the value of beta can be used. After cutting off further exploration of I, J is examined, yielding a value of 5, which is assigned as the value of F (since it is the maximum of 5 and 0). This value becomes the value of beta at node C. It indicates that C is guaranteed to get a 5 or less. Now we must expand G. First M is examined and it has a value of 7, which is passed back to G as its tentative value. But now 7 is compared to beta (5). It is greater, and the player whose turn it is at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least 7, since there is an alternative move to F, which will lead to a score of 5. Thus it is not necessary to explore any of the other branches of G.

From this example, we see that at maximizing levels, we can rule out a move early if it becomes clear that its value will be less than the current threshold, while at minimizing levels, search will be terminated if values that are greater than the current threshold are discovered. But ruling out a possible move by a maximizing player actually means cutting off the search at a minimizing level. Look again at the example in Figure 12.4. Once we determine that C is a bad move from A, we cannot bother to explore G, or any other paths, at the minimizing level below C. So the way alpha and beta are actually used is that search at a minimizing level can be terminated when a value less than alpha is discovered, while a search at a maximizing level can be terminated when a value greater than beta has been found. Cutting off search at a maximizing level when a high value is found may seem counterintuitive at first, but if you keep in mind that we only get to a particular node at a maximizing level if the minimizing player at the level above chooses it, then it makes sense.

Having illustrated the operation of alpha-beta pruning with examples, we can now explore how the MINIMAX procedure described in Section 12.2 can be modified to exploit this technique. Notice that at maximizing levels, only beta is used to determine whether to cut off the search, and at minimizing levels only alpha is used. But at maximizing levels alpha must also be known since when a recursive call is made

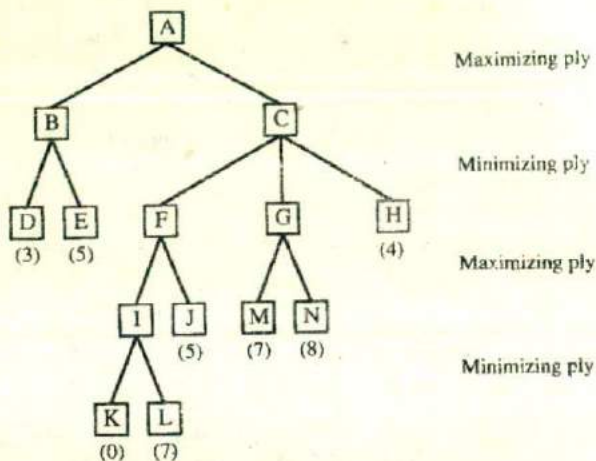


Figure 12.5: Alpha and Beta Cutoffs

to MINIMAX, a minimizing level is created, which needs access to alpha. So at maximizing levels alpha must be known not so that it can be used but so that it can be passed down the tree. The same is true of minimizing levels with respect to beta. Each level must receive both values, one to use and one to pass down for the next level to use.

The MINIMAX procedure as it stands does not need to treat maximizing and minimizing levels differently since it simply negates evaluations each time it changes levels. It would be nice if a comparable technique for handling alpha and beta could be found so that it would still not be necessary to write separate procedures for the two players. This turns out to be easy to do. Instead of referring to alpha and beta, MINIMAX uses two values, USE-THRESH and PASS-THRESH. USE-THRESH is used to compute cutoffs. PASS-THRESH is merely passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth. Just as values had to be negated each time they were passed across levels, so too must these thresholds be negated. This is necessary so that, regardless of the level of the search, a test for greater than will determine whether a threshold has been crossed. Now there need still be no difference between the code required at maximizing levels and that required at minimizing ones.

We have now described how alpha and beta values are passed down the tree. In addition, we must decide how they are to be set. To see how to do this, let's return first to the simple example of Figure 12.4. At a maximizing level, such as that of node A, alpha is set to be the value of the best successor that has yet been found. (Notice that although at maximizing levels it is beta that is used to determine cutoffs, it is alpha whose new value can be computed. Thus at any level, USE-THRESH will be checked for cutoffs and PASS-THRESH will be updated to be used later.) But if the maximizing

node is not at the top of the tree, we must also consider the alpha value that was passed down from a higher node. To see how this works, look again at Figure 12.5 and consider what happens at node F. We assign the value 0 to node I on the basis of examining node K. This is so far the best successor of F. But from an earlier exploration of the subtree headed by B, alpha was set to 3 and passed down from A to F. Alpha should not be reset to 0 on the basis of node I. It should stay as 3 to reflect the best move found so far in the entire tree. Thus we see that at a maximizing level, alpha should be set to either the value it had at the next-highest maximizing level or the best value found at this level, whichever is greater. The corresponding statement can be made about beta at minimizing levels. In fact, what we want to say is that at any level, PASS-THRESH should always be the maximum of the value it inherits from above and the best move found at its level. If PASS-THRESH is updated, the new value should be propagated both down to lower levels and back up to higher ones so that it always reflects the best move found anywhere in the tree.

At this point, we notice that we are doing the same thing in computing PASS-THRESH that we did in MINIMAX to compute BEST-SCORE. We might as well eliminate BEST-SCORE and let PASS-THRESH serve in its place.

With these observations, we are in a position to describe the operation of the function MINIMAX-A-B, which requires four arguments, *Position*, *Depth*, *Use-Thresh*, and *Pass-Thresh*. The initial call, to choose a move for PLAYER-ONE from the position CURRENT, should be

```
MINIMAX-A-B(CURRENT
             0,
             PLAYER-ONE,
             maximum value STATIC can compute,
             minimum value STATIC can compute)
```

These initial values for *Use-Thresh* and *Pass-Thresh* represent the worst values that each side could achieve.

Algorithm: MINIMAX-A-B(Position, Depth, Player, Use-Thresh, Pass-Thresh)

1. If DEEP-ENOUGH(*Position*, *Depth*), then return the structure
 VALUE = STATIC(*Position*, *Player*);
 PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(*Position*, *Player*) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have been returned if DEEP-ENOUGH had returned TRUE.
4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.
 For each element SUCC of SUCCESSORS:

- (a) Set RESULT-SUCC to
 $\text{MINIMAX-A-B}(\text{SUCC}, \text{Depth} + 1, \text{OPPOSITE}(\text{Player}),$
 $-\text{Pass-Thresh}, -\text{Use-Thresh}).$
- (b) Set NEW-VALUE to $-\text{VALUE}(\text{RESULT-SUCC}).$
- (c) If $\text{NEW-VALUE} > \text{Pass-Thresh}$, then we have found a successor that is better than any that have been examined so far. Record this by doing the following.
- i. Set Pass-Thresh to NEW-VALUE.
 - ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to the result of attaching SUCC to the front of $\text{PATH}(\text{RESULT-SUCC}).$
- (d) If Pass-Thresh (reflecting the current best value) is not better than Use-Thresh , then we should stop examining this branch. But both thresholds and values have been inverted. So if $\text{Pass-Thresh} \geq \text{Use-Thresh}$, then return immediately with the value
 $\text{VALUE} = \text{Pass-Thresh}$
 $\text{PATH} = \text{BEST-PATH}$

5. Return the structure

$\text{VALUE} = \text{Pass-Thresh}$
 $\text{PATH} = \text{BEST-PATH}$

The effectiveness of the alpha-beta procedure depends greatly on the order in which paths are examined. If the worst paths are examined first, then no cutoffs at all will occur. But, of course, if the best path were known in advance so that it could be guaranteed to be examined first, we would not need to bother with the search process. If, however, we knew how effective the pruning technique is in the perfect case, we would have an upper bound on its performance in other situations. It is possible to prove that if the nodes are perfectly ordered, then the number of terminal nodes considered by a search to depth d using alpha-beta pruning is approximately equal to twice the number of terminal nodes generated by a search to depth $d/2$ without alpha-beta [Knuth and Moore, 1975]. A doubling of the depth to which the search can be pursued is a significant gain. Even though all of this improvement cannot typically be realized, the alpha-beta technique is a significant improvement to the minimax search procedure. For a more detailed study of the average branching factor of the alpha-beta procedure, see Baudet [1978] and Pearl [1982].

The idea behind the alpha-beta procedure can be extended to cut off additional paths that appear to be at best only slight improvements over paths that have already been explored. In step 4(d), we cut off the search if the path we were exploring was not better than other paths already found. But consider the situation shown in Figure 12.6. After examining node G, we see that the best we can hope for if we make move C is a score of 3.2. We know that if we make move B we are guaranteed a score of 3. Since 3.2 is only very slightly better than 3, we should perhaps terminate our exploration of C now. We could then devote more time to exploring other parts of the tree where there may be more to gain. Terminating the exploration of a subtree that offers little possibility for improvement over other known paths is called a *futility cutoff*.

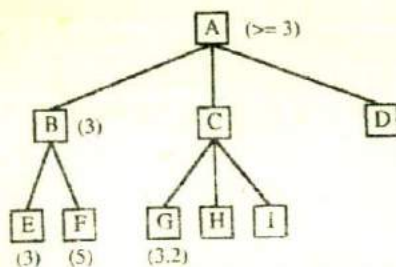


Figure 12.6: A Futility Cutoff

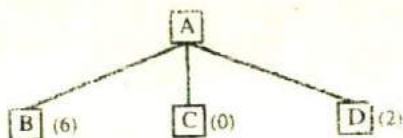


Figure 12.7: The Beginning of a Search.

12.4 Additional Refinements

In addition to alpha-beta pruning, there are a variety of other modifications to the minimax procedure that can also improve its performance. Four of them are discussed briefly in this section, and we discuss one other important modification in the next section.

12.4.1 Waiting for Quiescence

As we suggested above, one of the factors that should sometimes be considered in determining when to stop going deeper in the search tree is whether the situation is relatively stable. Consider the tree shown in Figure 12.7. Suppose that when node B is expanded one more level, the result is that shown in Figure 12.8. When we looked one move ahead, our estimate of the worth of B changed drastically. This might happen, for example, in the middle of a piece exchange. The opponent has significantly improved the immediate appearance of his or her position by initiating a piece exchange. If we stop exploring the tree at this level, we assign the value -4 to B and therefore decide that B is not a good move.

To make sure that such short-term measures do not unduly influence our choice of move, we should continue the search until no such drastic change occurs from one level to the next. This is called waiting for *quiescence*. If we do that, we might get the situation shown in Figure 12.9, in which the move to B again looks like a reasonable move for us to make since the other half of the piece exchange has occurred. A very general algorithm for quiescence can be found in Beal [1990].

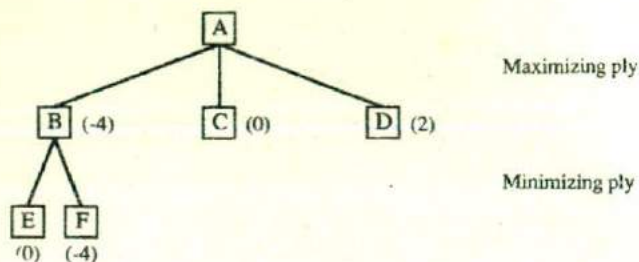


Figure 12.8: The Beginning of an Exchange

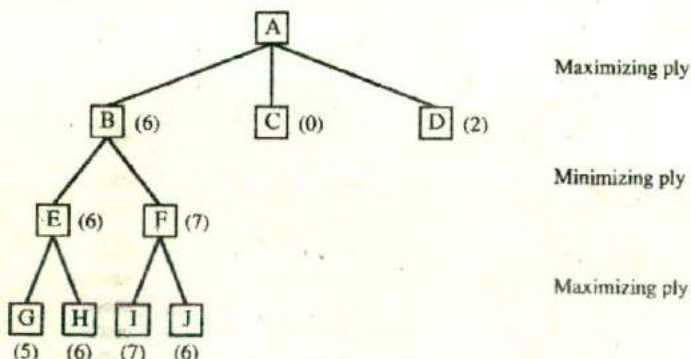


Figure 12.9: The Situation Calms Down

Waiting for quiescence helps in avoiding the *horizon effect*, in which an inevitable bad event can be delayed by various tactics until it does not appear in the portion of the game tree that minimax explores. The horizon effect can also influence a program's perception of good moves. The effect may make a move look good despite the fact that the move might be better if delayed past the horizon. Even with quiescence, all fixed-depth search programs are subject to subtle horizon effects.

12.4.2 Secondary Search

One good way of combating the horizon effect is to double-check a chosen move to make sure that a hidden pitfall does not exist a few moves farther away than the original search explored. Suppose we explore a game tree to an average depth of six ply and, on the basis of that search, choose a particular move. Although it would have been too expensive to have searched the entire tree to a depth of eight, it is not very expensive to search the single chosen branch an additional two levels to make sure that it still looks good. This technique is called *secondary search*.

One particularly successful form of secondary search is called *singular extensions*. The idea behind singular extensions is that if a leaf node is judged to be far superior to its siblings and if the value of the entire search depends critically on the correctness of that node's value, then the node is expanded one extra ply. This technique allows the search program to concentrate on tactical, forcing combinations. It employs a purely syntactic criterion, choosing interesting lines of play without recourse to any additional domain knowledge. The DEEP THOUGHT chess computer [Anantharaman *et al.*, 1990] has used singular extensions to great advantage, finding midgame mating combinations as long as thirty-seven moves, an impossible feat for fixed-depth minimax.

12.4.3 Using Book Moves

For complicated games taken as wholes, it is, of course, not feasible to select a move by simply looking up the current game configuration in a catalogue and extracting the correct move. The catalogue would be immense and no one knows how to construct it. But for some segments of some games, this approach is reasonable. In chess, for example, both opening sequences and endgame sequences are highly stylized. In these situations, the performance of a program can often be considerably enhanced if it is provided with a list of moves (called *book moves*) that should be made. The use of book moves in the opening sequences and endgames, combined with the use of the minimax search procedure for the midgame, provides a good example of the way that knowledge and search can be combined in a single program to produce more effective results than could either technique on its own.

12.4.4 Alternatives to Minimax

Even with the refinements above, minimax still has some problematic aspects. For instance, it relies heavily on the assumption that the opponent will always choose the optimal move. This assumption is acceptable in winning situations where a move that is guaranteed to be good for us can be found. But, as suggested in Berliner [1977], in a losing situation it might be better to take the risk that the opponent will make a mistake. Suppose we must choose between two moves, both of which, if the opponent plays perfectly, lead to situations that are very bad for us, but one is slightly less bad than the other. But further suppose that the less promising move could lead to a very good situation for us if the opponent makes a single mistake. Although the minimax procedure would choose the guaranteed bad move, we ought instead to choose the other one, which is probably slightly worse but possibly a lot better. A similar situation arises when one move appears to be only slightly more advantageous than another, assuming that the opponent plays perfectly. It might be better to choose the less advantageous move if it could lead to a significantly superior situation if the opponent makes a mistake. To make these decisions well, we must have access to a model of the individual opponent's playing style so that the likelihood of various mistakes can be estimated. But this is very hard to provide.

As a mechanism for propagating estimates of position strengths up the game tree, minimax stands on shaky theoretical grounds. Nau [1980] and Pearl [1983] have demonstrated that for certain classes of game trees, e.g., uniform trees with random terminal values, the deeper the search, the *poorer* the result obtained by minimaxing.

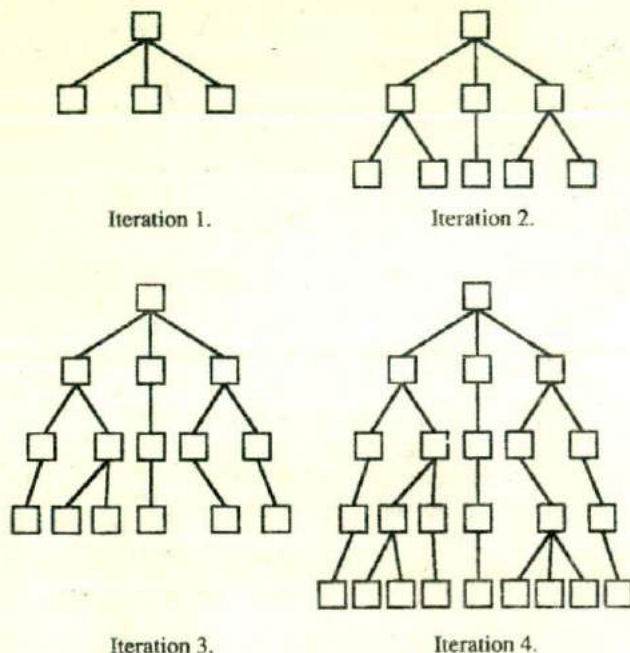


Figure 12.10: Iterative Deepening

This “pathological” behavior of amplifying error-prone heuristic estimates has not been observed in actual game-playing programs, however. It seems that game trees containing won positions and nonrandom distributions of heuristic estimates provide environments that are conducive to minimaxing.

12.5 Iterative Deepening

A number of ideas for searching two-player game trees have led to new algorithms for single-agent heuristic search, of the type described in Chapter 3. One such idea is *iterative deepening*, originally used in a program called CHESS 4.5 [Slate and Atkin, 1977]. Rather than searching to a fixed depth in the game tree, CHESS 4.5 first searched only a single ply, applying its static evaluation function to the result of each of its possible moves. It then initiated a new minimax search, this time to a depth of two ply. This was followed by a three-ply search, then a four-ply search, etc. The name “iterative deepening” derives from the fact that on each iteration, the tree is searched one level deeper. Figure 12.10 depicts this process.

On the face of it, this process seems wasteful. Why should we be interested in any iteration except the final one? There are several reasons. First, game-playing programs are subject to time constraints. For example, a chess program may be required to

complete all its moves within two hours. Since it is impossible to know in advance how long a fixed-depth tree search will take (because of variations in pruning efficiency and the need for selective search), a program may find itself running out of time. With iterative deepening, the current search can be aborted at any time and the best move found by the previous iteration can be played. Perhaps more importantly, previous iterations can provide invaluable move-ordering constraints. If one move was judged to be superior to its siblings in a previous iteration, it can be searched first in the next iteration. With effective ordering, the alpha-beta procedure can prune many more branches, and total search time can be decreased drastically. This allows more time for deeper iterations.

Years after CHESS 4.5's success with iterative deepening, it was noticed [Korf, 1985a] that the technique could also be applied effectively to single-agent search to solve problems like the 8-puzzle. In Section 2.2.1, we compared two types of uninformed search, depth-first search and breadth-first search. Depth-first search was efficient in terms of space but required some cutoff depth in order to force backtracking when a solution was not found. Breadth-first search was guaranteed to find the shortest solution path but required inordinate amounts of space because all leaf nodes had to be kept in memory. An algorithm called depth-first iterative deepening (DFID) combines the best aspects of depth-first and breadth-first search.

Algorithm: Depth-First Iterative Deepening

1. Set SEARCH-DEPTH = 1.
2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution path is found, then return it.
3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2.

Clearly, DFID will find the shortest solution path to the goal state. Moreover, the maximum amount of memory used by DFID is proportional to the number of nodes in that solution path. The only disturbing fact is that all iterations but the final one are essentially wasted. However, this is not a serious problem. The reason is that most of the activity during any given iteration occurs at the leaf-node level. Assuming a complete tree, we see that there are as many leaf nodes at level n as there are total nodes in levels 1 through n . Thus, the work expended during the n th iteration is roughly equal to the work expended during all previous iterations. This means that DFID is only slower than depth-first search by a constant factor. The problem with depth-first search is that there is no way to know in advance how deep the solution lies in the search space. DFID avoids the problem of choosing cutoffs without sacrificing efficiency, and, in fact, DFID is the optimal algorithm (in terms of space and time) for uninformed search.

But what about informed, heuristic search? Iterative deepening can also be used to improve the performance of the A* search algorithm [Korf, 1985a]. Since the major practical difficulty with A* is the large amount of memory it requires to maintain the search node lists, iterative deepening can be of considerable service.

Algorithm: Iterative-Deepening-A*

1. Set THRESHOLD = the heuristic evaluation of the start state.

2. Conduct a depth-first search, pruning any branch when its total cost function ($g + h'$) exceeds THRESHOLD.⁵ If a solution path is found during the search, return it.
3. Otherwise, increment THRESHOLD by the minimum amount it was exceeded during the previous step, and then go to Step 2.

Like A*, Iterative-Deepening-A* (IDA*) is guaranteed to find an optimal solution, provided that h' is an admissible heuristic. Because of its depth-first search technique, IDA* is very efficient with respect to space. IDA* was the first heuristic search algorithm to find optimal solution paths for the 15-puzzle (a 4x4 version of the 8-puzzle) within reasonable time and space constraints.

12.6 References on Specific Games

In this chapter we have discussed search-based techniques for game playing. We discussed the basic minimax algorithm and then introduced a series of refinements to it. But even with these refinements, it is still difficult to build good programs to play difficult games. Every game, like every AI task, requires a careful combination of search and knowledge.

Chess

Research on computer chess actually predates the field we call artificial intelligence. Shannon [1950] was the first to propose a method for automating the game, and two early chess programs were written by Greenblatt *et al.* [1967] and Newell and Simon [1972].

Chess provides a well-defined laboratory for studying the trade-off between knowledge and search. The more knowledge a program has, the less searching it needs to do. On the other hand, the deeper the search, the less knowledge is required. Human chess players use a great deal of knowledge and very little search—they typically investigate only 100 branches or so in deciding a move. A computer, on the other hand, is capable of evaluating millions of branches. Its chess knowledge is usually limited to a static evaluation function. Deep-searching chess programs have been calibrated on exercise problems in the chess literature and have even discovered errors in the official human analyses of the problems.

A chess player, whether human or machine, carries a numerical rating that tells how well it has performed in competition with other players. This rating lets us evaluate in an absolute sense the relative trade-offs between search and knowledge in this domain. The recent trend in chess-playing programs is clearly away from knowledge and toward faster brute force search. It turns out that deep, full-width search (with pruning) is sufficient for competing at very high levels of chess. Two examples of highly rated chess machines are HITECH [Berliner and Ebeling, 1989] and DEEP THOUGHT [Anantharaman *et al.*, 1990], both of which have beaten human grandmasters and both

⁵Recall that g stands for the cost so far in reaching the current node, and h' stands for the heuristic estimate of the distance from the node to the goal.

of which use custom-built parallel hardware to speed up legal move generation and heuristic evaluation.

Checkers

Work on computer checkers began with Samuel [1963]. Samuel's program had an interesting learning component which allowed its performance to improve with experience. Ultimately, the program was able to beat its author. We look more closely at the learning mechanisms used by Samuel in Chapter 17.

Go

Go is a very difficult game to play by machine since the average branching factor of the game tree is very high. Brute force search, therefore, is not as effective as it is in chess. Human go players make up for their inability to search deeply by using a great deal of knowledge about the game. It is probable that go-playing programs must also be knowledge-based, since today's brute-force programs cannot compete with humans. For a discussion of some of the issues involved, see Wilcox [1988].

Backgammon

Unlike chess, checkers, and go, a backgammon program must choose its moves with incomplete information about what may happen. If all the possible dice rolls are considered, the number of alternatives at each level of the search is huge. With current computational power, it is impossible to search more than a few ply ahead. Such a search will not expose the strengths and weaknesses of complex blocking positions, so knowledge-intensive methods must be used. One program that uses such methods is BKG Berliner [1980]. BKG actually does no searching at all but relies instead on positional understanding and understanding of how its goals should change for various phases of play. Like its chess-playing cousins, BKG has reached high levels of play even beating a human world champion in a short match.

NEUROGAMMON [Tesauro and Sejnowski, 1989] is another interesting backgammon program. It is based on a neural network model that learns from experience. Neurogammon is one of the few competitive game-playing programs that relies heavily on automatic learning.

Othello

Othello is a popular board game that is played on an 8x8 grid with bi-colored pieces. Although computer programs have already achieved world-championship level play [Rosenbloom, 1982; Lee and Mahajan, 1990], humans continue to study the game and international tournaments are held regularly. Computers are not permitted to compete in these tournaments, but it is believed that the best programs are stronger than the best humans. High-performance Othello programs rely on fast brute-force search and table lookup.

The Othello experience may shed some light on the future of computer chess. Will top human players in the future study chess games between World Champion computers in the same way that they study classic human grandmaster matches today? Perhaps it

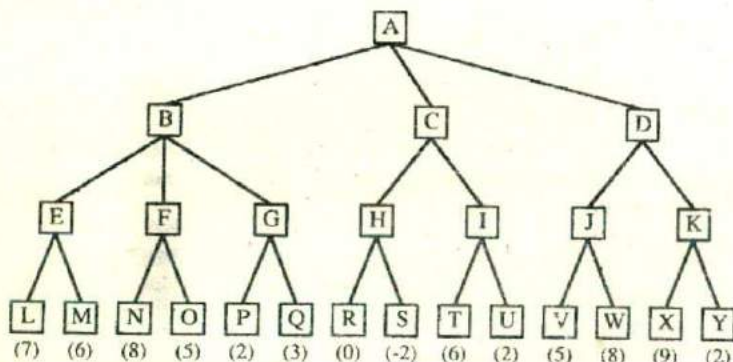
will turn out that the different search versus knowledge trade-offs made by humans and computers will make it impossible for either of them to benefit from the experiences of the other.

Others

Levy [1988] contains a number of classic papers on computer game playing. The papers cover the games listed above as well as bridge, scrabble, dominoes, go-moku, hearts, and poker.

12.7 Exercises

1. Consider the following game tree in which static scores are all from the first player's point of view:



Suppose the first player is the maximizing player. What move should be chosen?

2. In the game tree shown in the previous problem, what nodes would not need to be examined using the alpha-beta pruning procedure?
3. Why does the search in game-playing programs always proceed forward from the current position rather than backward from a goal state?
4. Is the minimax procedure a depth-first or breadth-first search procedure?
5. The minimax algorithm we have described searches a game tree. But for some games, it might be better to search a graph and to check, each time a position is generated, if it has been generated and evaluated before. Under what circumstances would this be a good idea? Modify the minimax procedure to do this.
6. How would the minimax procedure have to be modified to be used by a program playing a three- or four-person game rather than a two-person one?

7. In the context of the search procedure described in Section 12.3, does the ordering of the list of successor positions created by MOVEGEN matter? Why or why not? If it does matter, how much does it matter (i.e., how much effort is reasonable for ordering it)?
8. Implement the alpha beta search procedure. Use it to play a simple game such as tic-tac-toe.
9. Apply DFID to the water jug problem of Section 2.1.

Chapter 13

Planning

In order to solve most nontrivial problems, it is necessary to combine some of the basic problem-solving strategies discussed in Chapter 3 with one or more of the knowledge representation mechanisms that have just been presented. It is often also useful to divide the problem that must be solved into smaller pieces and to solve those pieces separately, to the extent that that is possible. In this chapter, we describe several techniques for doing this in order to construct plans for solving hard problems.

13.1 Overview

In Chapter 2, we described the process of problem solving as a search through a state space in which each point corresponded to a situation that might arise. The search started with an initial situation and performed a sequence of allowable operations until a situation corresponding to a goal was reached. Then, in Chapter 3, we described a variety of ways of moving through such a search space in an attempt to find a solution to a particular problem. For example, the A* algorithm provides a way of conducting a best-first search through a graph representing a problem space. Each node that is examined in the A* algorithm represents a description of a complete problem state, and each operator describes a way of changing the total state description. For simple problems, such as, say, the 8-puzzle, manipulating the complete state description at one time is easy and reasonable.

However, for more complicated problem domains, it becomes important to be able to work on small pieces of a problem separately and then to combine the partial solutions at the end into a complete problem solution. Unless we can do this, the number of combinations of the states of the components of a problem becomes too large to handle in the amount of time available. There are two ways in which it is important to be able to perform this decomposition.

First of all, we must avoid having to recompute the entire problem state when we move from one state to the next. Instead, we want to consider only that part of the state that may have changed. For example, if we move from one room to another, this does not affect the locations of the doors and the windows in the two rooms. The *frame problem*, which is the problem of how to determine which things change and which do

not, becomes increasingly important as the complexity of the problem state increases. It is not difficult to figure out how the state of the 8-puzzle should change after every move, nor is it a lot of work to record explicitly a new copy of the state with the appropriate changes made. Our rules for moving from one state to another can simply describe how one entire board position should be transformed into another.

But if we are considering the problem of guiding a robot around an ordinary house, the situation is much more complex. The description of a single state is very large since it must describe the location of each object in the house as well as that of the robot. A given action on the part of the robot will change only a small part of the total state. If the robot pushes a table across the room, then the locations of the table and all of the objects that were on it will change. But the locations of the other objects in the house will not. Instead of writing rules that describe transformations of one entire state into another, we would like to write rules that describe only the affected parts of the state description. The rest of the description can then be assumed to stay constant.

The second important way in which decomposition can make the solution of hard problems easier is the division of a single difficult problem into several, hopefully easier, subproblems. The AO* algorithm provides a way of doing this when it is possible to decompose the original problem into completely separate subproblems. Although this is sometimes possible, it often is not. Instead, many problems can be viewed as *nearly decomposable* [Simon, 1981], by which we mean that they can be divided into subproblems that have only a small amount of interaction. For example, suppose that we want to move all the furniture out of a room. This problem can be decomposed into a set of smaller problems, each involving moving one piece of furniture out of the room. Within each of these subproblems, considerations such as removing drawers can be addressed separately for each piece of furniture. But if there is a bookcase behind a couch, then we must move the couch before we can move the bookcase. To solve such nearly decomposable problems, we would like a method that enables us to work on each subproblem separately, using techniques such as the ones we have already studied, and then to record potential interactions among subproblems and to handle them appropriately.

Several methods for doing these two kinds of decomposition have been proposed and we investigate them in this chapter. These methods focus on ways of decomposing the original problem into appropriate subparts and on ways of recording and handling interactions among the subparts as they are detected during the problem-solving process. The use of these methods is often called *planning*.

In everyday usage, the word *planning* refers to the process of computing several steps of a problem-solving procedure before executing any of them. When we describe computer problem-solving behavior, the distinction between planning and doing fades a bit since rarely can the computer actually do much of anything besides plan. In solving the 8-puzzle, for example, it cannot actually push any tiles around. So when we discussed the computer solution of the 8-puzzle problem, what we were really doing was outlining the way the computer might generate a plan for solving it. For problems such as the 8-puzzle, the distinction between planning and doing is unimportant.

But in other situations, the distinction may be critical. Recall that in Chapter 2 one of the problem characteristics we discussed was whether solution steps could be ignored or undone if they prove unwise. If they can, then the process of planning a complete solution can proceed just as would an attempt to find a solution by actually

trying particular actions. If a dead-end path is detected, then a new one can be explored by backtracking to the last choice point. So, for example, in solving the 8-puzzle, a computer could look for a solution plan in the same way as a person who was actually trying to solve the problem by moving tiles on a board. If solution steps in the real world cannot be ignored or undone, though, planning becomes extremely important. Although real world steps may be irrevocable, computer simulation of those steps is not. So we can circumvent the constraints of the real world by looking for a complete solution in a simulated world in which backtracking is allowed. After we find a solution, we can execute it in the real world.

The success of this approach, however, hinges on another characteristic of a problem's domain: Is its universe predictable? If we look for a solution to a problem by actually carrying out sequences of operations, then at any step of the process we can be sure of the outcome of that step; it is whatever happened. But in an unpredictable universe, we cannot know the outcome of a solution step if we are only simulating it by computer. At best, we can consider the *set* of possible outcomes, possibly in some order according to the likelihood of the outcomes occurring. But then when we produce a plan and attempt to execute it, we must be prepared in case the actual outcome is not what we expected. If the plan included paths for all possible outcomes of each step, then we can simply traverse the paths that turn out to be appropriate. But often there are a great many possible outcomes, most of which are highly unlikely. In such situations, it would be a great waste of effort to formulate plans for all contingencies.

Instead, we have two choices. We can just take things one step at a time and not really try to plan ahead. This is the approach that is taken in *reactive systems*, which we will describe in Section 13.7. Our other choice is to produce a plan that is *likely* to succeed. But then what should we do if it fails? One possibility is simply to throw away the rest of the plan and start the planning process over, using the current situation as the new initial state. Sometimes, this is a reasonable thing to do.

But often the unexpected consequence does not invalidate the entire rest of the plan. Perhaps a small change, such as an additional step, is all that is necessary to make it possible for the rest of the plan to be useful. Suppose, for example, that we have a plan for baking an angel food cake. It involves separating some eggs. While carrying out the plan, we turn out to be slightly clumsy and one of the egg yolks falls into the dish of whites. We do not need to create a completely new plan (unless we decide to settle for some other kind of cake). Instead, we simply redo the egg-separating step until we get it right and then continue with the rest of the plan. This is particularly true for decomposable or nearly decomposable problems. If the final plan is really a composite of many smaller plans for solving a set of subproblems, then if one step of the plan fails, the only part of the remaining plan that can be affected is the rest of the plan for solving that subproblem. The rest of the plan is unrelated to that step. If the problem was only partially decomposable, then any subplans that interact with the affected one may also be affected. So, just as it was important during the planning process to keep track of interactions as they arise, it is important to record information about interactions along with the final plan so that if unexpected events occur at execution time, the interactions can be considered during replanning.

Hardly any aspect of the real world is completely predictable. So we must always be prepared to have plans fail. But, as we have just seen, if we have built our plan by decomposing our problem into as many separate (or nearly separate) subproblems as

possible, then the impact on our plan of the failure of one particular step may be quite local. Thus we have an additional argument in favor of the problem-decomposition approach to problem solving. In addition to reducing the combinatorial complexity of the problem-solving process, it also reduces the complexity of the dynamic plan revision process that may be required during the execution of a plan in an unpredictable world (such as the one in which we live).

In order to make it easy to patch up plans if they go awry at execution time, we will find that it is useful during the planning process not only to record the steps that are to be performed but also to associate with each step the reasons why it must be performed. Then, if a step fails, it is easy, using techniques for dependency-directed backtracking, to determine which of the remaining parts of the plan were dependent on it and so may need to be changed. If the plan-generation process proceeds backward from the desired goal state, then it is easy to record this dependency information. If, on the other hand, it proceeded forward from the start state, determining the necessary dependencies may be difficult. For this reason and because, for most problems, the branching factor is smaller going backward, most planning systems work primarily in a *goal-directed* mode in which they search backward from a goal state to an achievable initial state.

In the next several sections, a variety of planning techniques are presented. All of them, except the last, are problem-solving methods that rely heavily on problem decomposition. They deal (to varying degrees of success) with the inevitable interactions among the components that they generate.

13.2 An Example Domain: The Blocks World

The techniques we are about to discuss can be applied in a wide variety of task domains, and they have been. But to make it easy to compare the variety of methods we consider, we should find it useful to look at all of them in a single domain that is complex enough that the need for each of the mechanisms is apparent yet simple enough that easy-to-follow examples can be found. The blocks world is such a domain. There is a flat surface on which blocks can be placed. There are a number of square blocks, all the same size. They can be stacked one upon another. There is a robot arm that can manipulate the blocks. The actions it can perform include:

- UNSTACK(A, B)—Pick up block A from its current position on block B. The arm must be empty and block A must have no blocks on top of it.
- STACK(A, B)—Place block A on block B. The arm must already be holding A and the surface of B must be clear.
- PICKUP(A)—Pick up block A from the table and hold it. The arm must be empty and there must be nothing on top of block A.
- PUTDOWN(A)—Put block A down on the table. The arm must have been holding block A.

Notice that in the world we have described, the robot arm can hold only one block at a time. Also, since all blocks are the same size, each block can have at most one other

block directly on top of it.¹

In order to specify both the conditions under which an operation may be performed and the results of performing it, we need to use the following predicates:

- ON(A, B)—Block A is on block B.
- ONTABLE(A)—Block A is on the table.
- CLEAR(A)—There is nothing on top of block A.
- HOLDING(A)—The arm is holding block A.
- ARMEMPTY—The arm is holding nothing.

Various logical statements are true in this blocks world. For example,

$$\begin{aligned} & [\exists x : \text{HOLDING}(x)] \rightarrow \neg \text{ARMEMPTY} \\ & \forall x : \text{ONTABLE}(x) \rightarrow \neg \exists y : \text{ON}(x, y) \\ & \forall x : [\neg \exists y : \text{ON}(y, x)] \rightarrow \text{CLEAR}(x) \end{aligned}$$

The first of these statements says simply that if the arm is holding anything, then it is not empty. The second says that if a block is on the table, then it is not also on another block. The third says that any block with no blocks on it is clear.

13.3 Components of a Planning System

In problem-solving systems based on the elementary techniques discussed in Chapter 3, it was necessary to perform each of the following functions:

- Choose the best rule to apply next based on the best available heuristic information.
- Apply the chosen rule to compute the new problem state that arises from its application.
- Detect when a solution has been found.
- Detect dead ends so that they can be abandoned and the system's effort directed in more fruitful directions.

In the more complex systems we are about to explore, techniques for doing each of these tasks are also required. In addition, a fifth operation is often important:

- Detect when an almost correct solution has been found and employ special techniques to make it totally correct.

Before we discuss specific planning methods, we need to look briefly at the ways in which each of these five things can be done.

¹Actually, by careful alignment, two blocks could be placed on top of one, but we ignore that possibility.

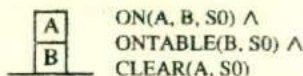


Figure 13.1: A Simple Blocks World Description

Choosing Rules to Apply

The most widely used technique for selecting appropriate rules to apply is first to isolate a set of differences between the desired goal state and the current state and then to identify those rules that are relevant to reducing those differences. If several rules are found, a variety of other heuristic information can be exploited to choose among them. This technique is based on the means-ends analysis method (recall Chapter 3). For example, if our goal is to have a white fence around our yard and we currently have a brown fence, we would select operators whose result involves a change of color of an object. If, on the other hand, we currently have no fence, we must first consider operators that involve constructing wooden objects.

Applying Rules

In the simple systems we have previously discussed, applying rules was easy. Each rule simply specified the problem state that would result from its application. Now, however, we must be able to deal with rules that specify only a small part of the complete problem state. There are many ways of doing this.

One way is to describe, for each action, each of the changes it makes to the state description. In addition, some statement that everything else remains unchanged is also necessary. An example of this approach is described in Green [1969]. In this system, a given state was described by a set of predicates representing the facts that were true in that state. Each distinct state was represented explicitly as part of the predicate. For example, Figure 13.1 shows how a state, called S_0 , of a simple blocks world problem could be represented.

The manipulation of these state descriptions was done using a resolution theorem prover. So, for example, the effect of the operator $UNSTACK(x, y)$ could be described by the following axiom. (In all the axioms given in this section, all variables are universally quantified unless otherwise indicated.)

$$\begin{aligned}
 & [CLEAR(x, s) \wedge ON(x, y, s)] \rightarrow \\
 & \quad [HOLDING(x, DO(UNSTACK(x, y), s)) \wedge \\
 & \quad \quad CLEAR(y, DO(UNSTACK(x, y), s))]
 \end{aligned}$$

Here, DO is a function that specifies, for a given state and a given action, the new state that results from the execution of the action. The axiom states that if $CLEAR(x)$ and $ON(x, y)$ both hold in state s , then $HOLDING(x)$ and $CLEAR(y)$ will hold in the state that results from DO ing an $UNSTACK(x, y)$, starting in state s .

If we execute $UNSTACK(A, B)$ in state S_0 as defined above, then we can prove, using our assertions about S_0 and our axiom about $UNSTACK$, that in the state that results from the unstacking operation (we call this state S_1).

$$\text{HOLDING}(A, S1) \wedge \text{CLEAR}(B, S1)$$

But what else do we know about the situation in state $S1$? Intuitively, we know that B is still on the table. But with what we have so far, we cannot derive it. To enable us to do so, we need also to provide a set of rules, called *frame axioms*, that describe components of the state that are not affected by each operator. So, for example, we need to say that

$$\text{ONTABLE}(z, s) \rightarrow \text{ONTABLE}(z, \text{DO}(\text{UNSTACK}(x, y), s))$$

This axiom says that the ONTABLE relation is never affected by the UNSTACK operator. We also need to say that the ON relation is only affected by the UNSTACK operator if the blocks involved in the ON relation are the same ones involved in the UNSTACK operation. This can be said as

$$[\text{ON}(m, n, s) \wedge \neg \text{EQUAL}(m, x)] \rightarrow \text{ON}(m, n, \text{DO}(\text{UNSTACK}(x, y), s))$$

The advantage of this approach is that a single mechanism, resolution, can perform all the operations that are required on state descriptions. The price we pay for this, however, is that the number of axioms that are required becomes very large if the problem-state descriptions are complex. For example, suppose that we are interested not only in the positions of our blocks but also in their color. Then, for every operation (except possibly PAINT), we would need an axiom such as the following:

$$\text{COLOR}(x, c, s) \rightarrow \text{COLOR}(x, c, \text{DO}(\text{UNSTACK}(y, z), s))$$

To handle complex problem domains, we need a mechanism that does not require a large number of explicit frame axioms. One such mechanism is that used by the early robot problem-solving system STRIPS [Fikes and Nilsson, 1971] and its descendants. In this approach, each operation is described by a list of new predicates that the operator causes to become true and a list of old predicates that it causes to become false. These two lists are called the ADD and DELETE lists, respectively. A third list must also be specified for each operator. This PRECONDITION list contains those predicates that must be true for the operator to be applied. The frame axioms of Green's system are specified implicitly in STRIPS . Any predicate not included on either the ADD or DELETE list of an operator is assumed to be unaffected by it. This means that, in specifying each operator, we need not consider aspects of the domain that are unrelated to it. Thus we need say nothing about the relationship of UNSTACK to COLOR . Of course, this means that some mechanism other than simple theorem proving must be used to compute complete state descriptions after operations have been performed.

STRIPS -style operators that correspond to the blocks world operations we have been discussing are shown in Figure 13.2. Notice that for simple rules such as these the PRECONDITION list is often identical to the DELETE list. In order to pick up a block, the robot arm must be empty; as soon as it picks up a block, it is no longer empty. But preconditions are not always deleted. For example, in order for the arm to pick up a block, the block must have no other blocks on top of it. After it is picked up, it still

STACK(x, y)	<p>P: CLEAR(y) \wedge HOLDING(x) D: CLEAR(y) \wedge HOLDING(x) A: ARMEMPTY \wedge ON(x, y)</p>
UNSTACK(x, y)	<p>P: ON(x, y) \wedge CLEAR(x) \wedge ARMEMPTY D: ON(x, y) \wedge ARMEMPTY A: HOLDING(x) \wedge CLEAR(y)</p>
PICKUP(x)	<p>P: CLEAR(x) \wedge ONTABLE(x) \wedge ARMEMPTY D: ONTABLE(x) \wedge ARMEMPTY A: HOLDING(x)</p>
PUTDOWN(x)	<p>P: HOLDING(x) D: HOLDING(x) A: ONTABLE(x) \wedge ARMEMPTY</p>

Figure 13.2: STRIPS-Style Operators for the Blocks World

has no blocks on top of it. This is the reason that the PRECONDITION and DELETE lists must be specified separately.

By making the frame axioms implicit, we have greatly reduced the amount of information that must be provided for each operator. This means, among other things, that when a new attribute that objects might possess is introduced into the system, it is not necessary to go back and add a new axiom for each of the existing operators. But how can we actually achieve the effect of the use of the frame axioms in computing complete state descriptions? The first thing we notice is that for complex state descriptions, most of the state remains unchanged after each operation. But if we represent the state as an explicit part of each predicate, as was done in Green's system, then all that information must be deduced all over again for each state. To avoid that, we can drop the explicit state indicator from the individual predicates and instead simply update a single database of predicates so that it always describes the current state of the world. For example, if we start with the situation shown in Figure 13.1, we would describe it as

$$\text{ON}(A, B) \wedge \text{ONTABLE}(B) \wedge \text{CLEAR}(A)$$

After applying the operator UNSTACK(A, B), our description of the world would be

$$\text{ONTABLE}(B) \wedge \text{CLEAR}(A) \wedge \text{CLEAR}(B) \wedge \text{HOLDING}(A)$$

This is derived using the ADD and DELETE lists specified as part of the UNSTACK operator.

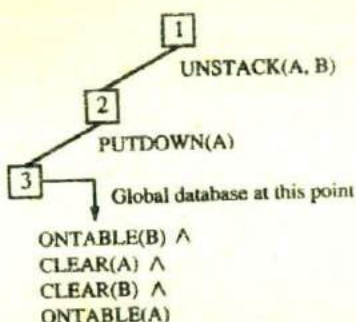


Figure 13.3: A Simple Search Tree

Simply updating a single state description works well as a way of keeping track of the effects of a given sequence of operators. But what happens during the process of searching for the correct operator sequence? If one incorrect sequence is explored, it must be possible to return to the original state so that a different one can be tried. But this is possible even if the global database describes the problem state at the current node of the search graph. All we need to do is record at each node the changes that were made to the global database as we passed through the node. Then, if we backtrack through that node, we can undo the changes. But the changes are described exactly in the ADD and DELETE lists of the operators that have been applied to move from one node to another. So we need only record, along each arc of the search graph, the operator that was applied. Figure 13.3 shows a small example of such a search tree and the corresponding global database. The initial state is the one shown in Figure 13.1 and described in STRIPS form above. Notice that we must specify not just the operator (e.g., UNSTACK) but also its arguments in order to be able to undo the changes later.

Now suppose that we want to explore a path different from the one we have just shown. First we backtrack through node 3 by *adding each of the predicates* in PUTDOWN's DELETE list to the global database and *deleting each of the elements* of PUTDOWN's ADD list. After doing that, the database contains

$$\text{ONTABLE(B)} \wedge \text{CLEAR(A)} \wedge \text{CLEAR(B)} \wedge \text{HOLDING(A)}$$

As we expected, this description is identical to the one we previously computed as the result of applying UNSTACK to the initial situation. If we repeat this process using the ADD and DELETE lists of UNSTACK, we derive a description identical to the one with which we started.

Because an implicit statement of the frame axioms is so important in complex problem domains, all the techniques we look at exploit STRIPS-style descriptions of the available operators.

Detecting a Solution

A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transforms the initial problem state into the goal state. How will it know when this has been done? In simple problem-solving systems, this question is easily answered by a straightforward match of the state descriptions. But if entire states are not represented explicitly but rather are described by a set of relevant properties, then this problem becomes more complex. The way it can be solved depends on the way that state descriptions are represented. For any representational scheme that is used, it must be possible to reason with representations to discover whether one matches another. Recall that in Part II we discussed a variety of ways that complex objects could be represented as well as reasoning mechanisms for each representation. Any of those representations (or some combination of them) could be used to describe problem states. Then the corresponding reasoning mechanisms could be used to discover when a solution had been found.

One representational technique has served as the basis for many of the planning systems that have been built. It is predicate logic, which is appealing because of the deductive mechanisms that it provides. Suppose that, as part of our goal, we have the predicate $P(x)$. To see whether $P(x)$ is satisfied in some state, we ask whether we can prove $P(x)$ given the assertions that describe that state and the axioms that define the world model (such as the fact that if the arm is holding something, then it is not empty). If we can construct such a proof, then the problem-solving process terminates. If we cannot, then a sequence of operators that might solve the problem must be proposed. This sequence can then be tested in the same way as the initial state was by asking whether $P(x)$ can be proved from the axioms and the state description that was derived by applying the operators.

Detecting Dead Ends

As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution (or at least appears unlikely to lead to one). The same reasoning mechanisms that can be used to detect a solution can often be used for detecting a dead end.

If the search process is reasoning forward from the initial state, it can prune any path that leads to a state from which the goal state cannot be reached. For example, suppose we have a fixed supply of paint: some white, some pink, and some red. We want to paint a room so that it has light red walls and a white ceiling. We could produce light red paint by adding some white paint to the red. But then we could not paint the ceiling white. So this approach should be abandoned in favor of mixing the pink and red paints together. We can also prune paths that, although they do not preclude a solution, appear to be leading no closer to a solution than the place from which they started.

If the search process is reasoning backward from the goal state, it can also terminate a path either because it is sure that the initial state cannot be reached or because little progress is being made. In reasoning backward, each goal is decomposed into subgoals. Each of them, in turn, may lead to a set of additional subgoals. Sometimes it is easy to detect that there is no way that all the subgoals in a given set can be satisfied at once. For example, the robot arm cannot be both empty and holding a block. Any path that is

attempting to make both of those goals true simultaneously can be pruned immediately. Other paths can be pruned because they lead nowhere. For example, if, in trying to satisfy goal A, the program eventually reduces its problem to the satisfaction of goal A as well as goals B and C, it has made little progress. It has produced a problem even harder than its original one, and the path leading to this problem should be abandoned.

Repairing an Almost Correct Solution

The kinds of techniques we are discussing are often useful in solving *nearly* decomposable problems. One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the subproblems separately, and then check that when the subsolutions are combined, they do in fact yield a solution to the original problem. Of course, if they do, then nothing more need be done. If they do not, however, there are a variety of things that we can do. The simplest is just to throw out the solution, look for another one, and hope that it is better. Although this is simple, it may lead to a great deal of wasted effort.

A slightly better approach is to look at the situation that results when the sequence of operations corresponding to the proposed solution is executed and to compare that situation to the desired goal. In most cases, the difference between the two will be smaller than the difference between the initial state and the goal (assuming that the solution we found did some useful things). Now the problem-solving system can be called again and asked to find a way of eliminating this new difference. The first solution can then be combined with this second one to form a solution to the original problem.

An even better way to patch up an almost correct solution is to appeal to specific knowledge about what went wrong and then to apply a direct patch. For example, suppose that the reason that the proposed solution is inadequate is that one of its operators cannot be applied because at the point it should have been invoked, its preconditions were not satisfied. This might occur if the operator had two preconditions and the sequence of operations that makes the second one true undid the first one. But perhaps, if an attempt were made to satisfy the preconditions in the opposite order, this problem would not arise.

A still better way to patch up incomplete solutions is not really to patch them up at all but rather to leave them incompletely specified until the last possible moment. Then when as much information as possible is available, complete the specification in such a way that no conflicts arise. This approach can be thought of as a *least-commitment* strategy. It can be applied in a variety of ways. One is to defer deciding on the order in which operations will be performed. So, in our previous example, instead of arbitrarily choosing one order in which to satisfy a set of preconditions, we could leave the order unspecified until the very end. Then we would look at the effects of each of the subsolutions to determine the dependencies that exist among them. At that point, an ordering can be chosen.

13.4 Goal Stack Planning

One of the earliest techniques to be developed for solving compound goals that may interact was the use of a goal stack. This was the approach used by STRIPS. In this

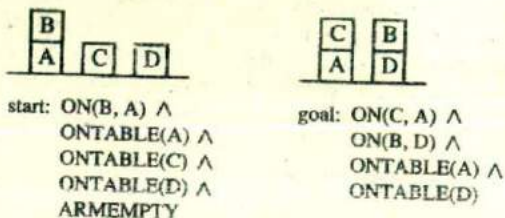


Figure 13.4: A Very Simple Blocks World Problem

method, the problem solver makes use of a single stack that contains both goals and operators that have been proposed to satisfy those goals. The problem solver also relies on a database that describes the current situation and a set of operators described as *PRECONDITION*, *ADD*, and *DELETE* lists. To see how this method works, let us carry it through for the simple example shown in Figure 13.4.

When we begin solving this problem, the goal stack is simply

$$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$$

But we want to separate this problem into four subproblems, one for each component of the original goal. Two of the subproblems, *ONTABLE(A)* and *ONTABLE(D)*, are already true in the initial state. So we will work on only the remaining two. Depending on the order in which we want to tackle the subproblems, there are two goal stacks that could be created as our first step, where each line represents one goal on the stack and *OTAD* is an abbreviation for *ONTABLE(A) \wedge ONTABLE(D)*:

$$\begin{array}{ll} ON(C, A) & ON(B, D) \\ ON(B, D) & ON(C, A) \\ ON(C, A) \wedge ON(B, D) \wedge OTAD & ON(C, A) \wedge ON(B, D) \wedge OTAD \end{array}$$

[1]

[2]

At each succeeding step of the problem-solving process, the top goal on the stack will be pursued. When a sequence of operators that satisfies it is found, that sequence is applied to the state description, yielding a new description. Next, the goal that is then at the top of the stack is explored and an attempt is made to satisfy it, starting from the situation that was produced as a result of satisfying the first goal. This process continues until the goal stack is empty. Then, as one last check, the original goal is compared to the final state derived from the application of the chosen operators. If any components of the goal are not satisfied in that state (which they might not be if they were achieved at one point and then undone later), then those unsolved parts of the goal are reinserted onto the stack and the process resumed.

To continue with the example we started above, let us assume that we choose first to explore alternative 1. Alternative 2 will also lead to a solution. In fact, it finds one so trivially that it is not very interesting. Exploring alternative 1, we first check to see

whether $ON(C, A)$ is true in the current state. Since it is not, we check for operators that could cause it to be true. Of the four operators we are considering, there is only one, $STACK$, and it would have to be called with C and A . So we place $STACK(C, A)$ on the stack in place of $ON(C, A)$, yielding

$STACK(C, A)$
 $ON(B, D)$
 $ON(C, A) \wedge ON(B, D) \wedge OTAD$

$STACK(C, A)$ replaced $ON(C, A)$ because after performing the $STACK$ we are guaranteed that $ON(C, A)$ will hold. But in order to apply $STACK(C, A)$, its preconditions must hold, so we must establish them as subgoals. Again we must separate a compound goal

$CLEAR(A) \wedge HOLDING(C)$

into its components and choose an order in which to work on them. At this point, it is useful to exploit some heuristic knowledge. $HOLDING(x)$ is very easy to achieve. At most, it is necessary to put down something else and then to pick up the desired object. But $HOLDING$ is also very easy to undo. In order to do almost anything else, the robot will need to use the arm. So if we achieve $HOLDING$ first and then try to do something else, we will most likely end up with $HOLDING$ no longer true. So we exploit the heuristic that if $HOLDING$ is one of several goals to be achieved at once, it should be tackled last. This produces the new goal stack

$CLEAR(A)$
 $HOLDING(C)$
 $CLEAR(A) \wedge HOLDING(C)$
 $STACK(C, A)$
 $ON(B, D)$
 $ON(C, A) \wedge ON(B, D) \wedge OTAD$

This kind of heuristic information could be contained in the precondition list itself by stating the predicates in the order in which they should be achieved.

Next we check to see if $CLEAR(A)$ is true. It is not. The only operator that could make it true is $UNSTACK(B, A)$. So we will attempt to apply it. This produces the goal stack

$ON(B, A)$
 $CLEAR(B)$
 $ARMEMPTY$
 $ON(B, A) \wedge CLEAR(B) \wedge ARMEMPTY$
 $UNSTACK(B, A)$
 $HOLDING(C)$
 $CLEAR(A) \wedge HOLDING(C)$
 $STACK(C, A)$
 $ON(B, D)$
 $ON(C, A) \wedge ON(B, D) \wedge OTAD$

This time, when we compare the top element of the goal stack, $ON(B, A)$, to the world model, we see that it is satisfied. So we pop it off and consider the next goal, $CLEAR(B)$. It, too, is already true in the world model, although it was not stated explicitly as one of the initial predicates. But from the initial predicates and the blocks world axiom that says that any block with no blocks on it is clear, a theorem prover could derive $CLEAR(B)$. So that goal, too, can be popped from the stack. The third precondition for $UNSTACK(B, A)$ remains. It is $ARMEMPTY$, and it is also true in the current world model, so it can be popped off the stack. The next element on the stack is the combined goal representing all of the preconditions for $UNSTACK(B, A)$. We check to make sure it is satisfied in the world model. It will be unless we undid one of its components in attempting to satisfy another. In this case, there is no problem and the combined goal can be popped from the stack.

Now the top element of the stack is the operator $UNSTACK(B, A)$. We are now guaranteed that its preconditions are satisfied, so it can be applied to produce a new world model from which the rest of the problem-solving process can continue. This is done using the ADD and $DELETE$ lists specified for $UNSTACK$. Meanwhile we record that $UNSTACK(B, A)$ is the first operator of the proposed solution sequence. At this point, the database corresponding to the world model is

$ONTABLE(A) \wedge ONTABLE(C) \wedge ONTABLE(D) \wedge$
 $HOLDING(B) \wedge CLEAR(A)$

The goal stack now is

$HOLDING(C)$
 $CLEAR(A) \wedge HOLDING(C)$
 $STACK(C, A)$
 $ON(B, D)$
 $ON(C, A) \wedge ON(B, D) \wedge OTAD$

We now attempt to satisfy the goal $HOLDING(C)$. There are two operators that might make $HOLDING(C)$ true: $PICKUP(C)$ and $UNSTACK(C, x)$, where x could be any block from which C could be unstacked. Without looking ahead, we cannot tell which of these operators is appropriate, so we create two branches of the search tree, corresponding to the following goal stacks:

$ONTABLE(C)$	$ON(C, x)$
$CLEAR(C)$	$CLEAR(C)$
$ARMEMPTY$	$ARMEMPTY$
$ONTABLE(C) \wedge CLEAR(C) \wedge$	$ON(C, x) \wedge CLEAR(C) \wedge$
$ARMEMPTY$	$ARMEMPTY$
$PICKUP(C)$	$UNSTACK(C, x)$
$CLEAR(A) \wedge HOLDING(C)$	$CLEAR(A) \wedge HOLDING(C)$
$STACK(C, A)$	$STACK(C, A)$
$ON(B, D)$	$ON(B, D)$
$ON(C, A) \wedge ON(B, D) \wedge OTAD$	$ON(C, A) \wedge ON(B, D) \wedge OTAD$

[1]

[2]

Notice that for alternative 2, the goal stack now contains a variable x , which appears in three places. Although any block could be substituted for x , it is important that the same one be matched to each of the x 's. Thus it is important that each time a variable is introduced into the goal stack, it be given a name distinct from any other variables already in the stack. And whenever a candidate object is chosen to match a variable, the binding must be recorded so that other occurrences of the same variable will be bound to the same object.

How should our program choose now between alternative 1 and alternative 2? We can tell that picking up C (alternative 1) is better than unstacking it because it is not currently on anything. So to unstack it, we would first have to stack it. Although this could be done, it would be a waste of effort. But how could a program know that? Suppose we decided to pursue alternative 2 first. To satisfy $ON(C, x)$, we would have to STACK C onto some block x . The goal stack would then be

```
CLEAR(x)
HOLDING(C)
CLEAR(x)  $\wedge$  HOLDING(C)
STACK(C, x)
CLEAR(C)
ARMEMPTY
ON(C, x)  $\wedge$  CLEAR(C)  $\wedge$  ARMEMPTY
UNSTACK(C, x)
CLEAR(A)  $\wedge$  HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A)  $\wedge$  ON(B, D)  $\wedge$  OTAD
```

But now notice that one of the preconditions of STACK is HOLDING(C). This is what we were trying to achieve by applying UNSTACK, which required us to apply STACK so that the precondition $ON(C, x)$ would be satisfied. So we are back to our original goal. In fact, we now have additional goals as well, since other predicates have also been added to the stack. At this point, this path can be terminated as unproductive. If, however, block C had been on another block in the current state, $ON(C, x)$ would have been satisfied immediately with no need to do a STACK and this path would have led to a good solution.

Now we must return to alternative 1, which used PICKUP to get the arm holding C. The top element on the goal stack is $ONTABLE(C)$, which is already satisfied, so we pop it off. The next element is CLEAR(C), which is also satisfied, so we pop it off. The remaining precondition of PICKUP(C) is ARMEMPTY, which is not satisfied since HOLDING(B) is true. There are two operators that could be applied to make ARMEMPTY true: STACK(B, x) and PUTDOWN(B). In other words, we can either put B on the table or we can put it on another block. Which should we choose? If we look ahead a bit, we see that we ultimately want to get B onto D. It would be most efficient simply to put it there now. Our program could figure this out by comparing the elements of the ADD lists of the competing operators to the rest of the goal stack. If one of the operators has the fortuitous effect of making any of those goals true, it should be chosen. So we choose to apply STACK(B, D) by binding D to x in the STACK operator. This makes the goal stack

CLEAR(D)
 HOLDING(B)
 CLEAR(D) \wedge HOLDING(B)
STACK(B, D)
 ONTABLE(C) \wedge CLEAR(C) \wedge ARMEMPTY
PICKUP(C)
 CLEAR(A) \wedge HOLDING(C)
STACK(C, A)
 ON(B, D)
 ON(C, A) \wedge ON(B, D) \wedge OTAD

CLEAR(D) and HOLDING(B) are both true. Now the operation STACK(B, D) can be performed, producing the world model

ONTABLE(A) \wedge ONTABLE(C) \wedge ONTABLE(D) \wedge
 ON(B, D) \wedge ARMEMPTY

All of the preconditions for PICKUP(C) are now satisfied so it, too, can be executed. Then all of the preconditions of STACK(C, A) are true, so it can be executed.

Now we can begin work on the second part of our original goal, ON(B, D). But it has already been satisfied by the operations that were used to satisfy the first subgoal. This happened because when we had a choice of ways to get rid of the arm holding B, we scanned back down the goal stack to see if one of the operators would have other useful side effects and we found that one did. So we now pop ON(B, D) off the goal stack. We then do one last check of the combined goal ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D) to make sure that all four parts still hold, which, of course, they do here. The problem solver can now halt and return as its answer the plan

1. \neg UNSTACK(B, A)
2. STACK(B, D)
3. PICKUP(C)
4. STACK(C, A)

In this simple example, we saw a way in which heuristic information can be applied to guide the search process, a way in which an unprofitable path could be detected, and a way in which considering some interaction among goals could help produce a good overall solution. But for problems more difficult than this one, these methods are not adequate.

To see why this method may fail to find a good solution, we attempt to solve the problem shown in Figure 13.5.² There are two ways that we could begin solving this problem, corresponding to the goal stacks

ON(A, B)	ON(B, C)
ON(B, C)	ON(A, B)
ON(A, B) \wedge ON(B, C)	ON(A, B) \wedge ON(B, C)

[1]

[2]

²This problem is often called the *Sussman Anomaly*, because it was carefully studied in Sussman [1975].

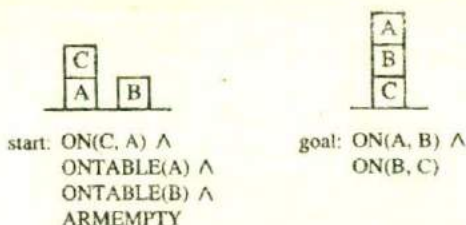


Figure 13.5: A Slightly Harder Blocks Problem

```

ON(C, A)
CLEAR(C)
ARMEMPTY
ON(C, A)  $\wedge$  CLEAR(C)  $\wedge$  ARMEMPTY
UNSTACK(C, A)
ARMEMPTY
CLEAR(A)  $\wedge$  ARMEMPTY
PICKUP(A)
CLEAR(B)  $\wedge$  HOLDING(A)
STACK(A, B)
ON(B, C)
ON(A, B)  $\wedge$  ON(B, C)

```

Figure 13.6: A Goal Stack

Suppose that we choose alternative 1 and begin trying to get A on B. We will eventually produce the goal stack shown in Figure 13.6.

We can then pop off the stack goals that have already been satisfied, until we reach the ARMEMPTY precondition of PICKUP(A). To satisfy it, we need to PUTDOWN(C). Then we can continue popping until the goal stack is

```

ON(B, C)
ON(A, B)  $\wedge$  ON(B, C)

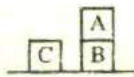
```

Then the current state is

```

ONTABLE(B)  $\wedge$ 
ON(A, B)  $\wedge$ 
ONTABLE(C)  $\wedge$ 
ARMEMPTY

```



The sequence of operators applied so far is

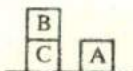
1. UNSTACK(C, A).
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A, B)

Now we can begin to work on satisfying $ON(B, C)$. Without going through all the detail, we can see that our algorithm will attempt to achieve this goal by stacking B on C. But to do that, it has to unstack A from B. By the time we have achieved the goal $ON(B, C)$ and popped it off the stack, we will have executed the following additional sequence of operators:

5. UNSTACK(A, B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B, C)

The problem state will be

$ON(B, C) \wedge$
 $ONTABLE(A) \wedge$
 $ONTABLE(C) \wedge$
 $ARMEMPTY$



But now when we check the remaining goal on the stack,

$ON(A, B) \wedge ON(B, C)$

we discover that it is not satisfied. We have undone $ON(A, B)$ in the process of achieving $ON(B, C)$. The difference between the goal and the current state is $ON(A, B)$, which is now added to the stack so that it can be achieved again. This time, the sequence of operators

9. PICKUP(A)
10. STACK(A, B)

is found. Now the combined goal is again checked, and this time it is satisfied. The complete plan that has been discovered is

- | | |
|------------------|-----------------|
| 1. UNSTACK(C, A) | 6. PUTDOWN(A) |
| 2. PUTDOWN(C) | 7. PICKUP(B) |
| 3. PICKUP(A) | 8. STACK(B, C) |
| 4. STACK(A, B) | 9. PICKUP(A) |
| 5. UNSTACK(A, B) | 10. STACK(A, B) |

Although this plan will achieve the desired goal, it does not do so very efficiently. A similar situation would have occurred if we had examined the two major subgoals in the opposite order. The method we are using is not capable of finding an efficient way of solving this problem.

There are two approaches we can take to the question of how a good plan can be found. One is to look at ways to repair the plan we already have to make it more

efficient. In this case, that is fairly easy to do. We can look for places in the plan where we perform an operation and then immediately undo it. If we find any such places, we can eliminate both the doing and the undoing steps from the plan. Applying this rule to our plan, we eliminate steps 4 and 5. Once we do that, we can also eliminate steps 3 and 6. The resulting plan

- | | |
|------------------|----------------|
| 1. UNSTACK(C, A) | 4. STACK(B, C) |
| 2. PUTDOWN(C) | 5. PICKUP(A) |
| 3. PICKUP(B) | 6. STACK(A, B) |

contains, in fact, the minimum number of operators needed to solve this problem. But for more complex tasks, the interfering operations may be farther apart in the plan and thus much more difficult to detect. In addition, we wasted a good deal of problem-solving effort producing all the steps that were later eliminated. It would be better if there were a plan-finding procedure that could construct efficient plans directly. In the next section, we present a technique for doing this.

13.5 Nonlinear Planning Using Constraint Posting

The goal-stack planning method attacks problems involving conjoined goals by solving the goals one at a time, in order. A plan generated by this method contains a sequence of operators for attaining the first goal, followed by a complete sequence for the second goal, etc. But as we have seen, difficult problems cause goal interactions. The operators used to solve one subproblem may interfere with the solution to a previous subproblem. Most problems require an intertwined plan in which multiple subproblems are worked on simultaneously. Such a plan is called a *nonlinear plan* because it is not composed of a linear sequence of complete subplans.

As an example of the need for a nonlinear plan, let us return to the Sussman anomaly described in Figure 13.5. A good plan for the solution of this problem is the following:

1. Begin work on the goal ON(A, B) by clearing A, thus putting C on the table.
2. Achieve the goal ON(B, C) by stacking B on C.
3. Complete the goal ON(A, B) by stacking A on B.

This section explores some heuristics and algorithms for tackling nonlinear problems such as this one.

Many ideas about nonlinear planning were present in HACKER [Sussman, 1975], an automatic programming system. The first true nonlinear planner, though, was NOAH [Sacerdoti, 1975]. NOAH was further improved upon by the NONLIN program [Tate, 1977]. The goal stack algorithm of STRIPS was transformed into a goal *set* algorithm by Nilsson [1980]. Subsequent planning systems, such as MOLGEN [Stefik, 1981b] and TWEAK [Chapman, 1987], used *constraint posting* as a central technique.

The idea of constraint posting is to build up a plan by incrementally hypothesizing operators, partial orderings between operators, and bindings of variables within operators. At any given time in the problem-solving process, we may have a set of useful operators but perhaps no clear idea of how those operators should be ordered with respect

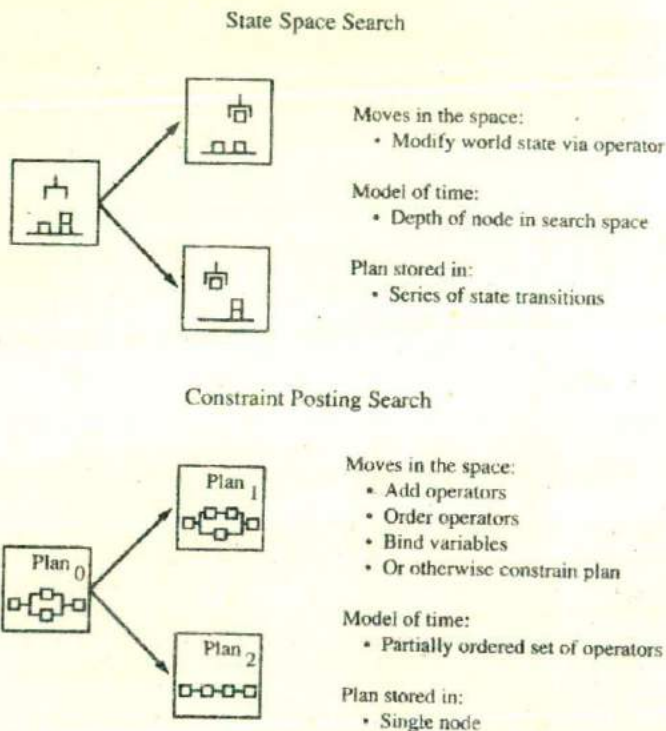


Figure 13.7: Constraint Posting versus State Space Search

to each other. A solution is a partially ordered, partially instantiated set of operators; to generate an actual plan, we convert the partial order into any of a number of total orders. Figure 13.7 shows the difference between the constraint posting method and the planning methods discussed in earlier sections.

We now examine several operations for nonlinear planning in a constraint-posting environment, although many of the operations themselves predate the use of the technique in planning.

Let's incrementally generate a nonlinear plan to solve the Sussman anomaly problem. We begin with the null plan, i.e., a plan with no steps. Next we look at the goal state and posit steps for achieving that goal. Means-ends analysis tells us to choose two steps with respective postconditions ON(A, B) and ON(B, C):

1. Step Addition—Creating new steps for a plan.
2. Promotion—Constraining one step to come before another in a final plan.
3. Declobbering—Placing one (possibly new) step s_2 between two old steps s_1 and s_3 , such that s_2 reasserts some precondition of s_3 that was negated (or “clobbered”) by s_1 .
4. Simple Establishment—Assigning a value to a variable, in order to ensure the preconditions of some step.
5. Separation—Preventing the assignment of certain values to a variable.

Figure 13.8: Heuristics for Planning Using Constraint Posting (TWEAK)

CLEAR(B)	CLEAR(C)
*HOLDING(A)	*HOLDING(B)
<hr/>	
STACK(A, B)	STACK(B, C)
<hr/>	
ARMEMPTY	ARMEMPTY
ON(A, B)	ON(B, C)
¬CLEAR(B)	¬CLEAR(C)
¬HOLDING(A)	¬HOLDING(B)

Each step is written with its preconditions above it and its postconditions below it. Delete postconditions are marked with a negation symbol (\neg). Notice that, at this point, the steps are not ordered with respect to each other. All we know is that we want to execute both of them eventually. Neither can be executed right away because some of their preconditions are not satisfied. An unachieved precondition is marked with a star (*). Both of the *HOLDING preconditions are unachieved because the arm holds nothing in the initial problem state.

Introducing new steps to achieve goals or preconditions is called *step addition*, and it is one of the heuristics we will use in generating nonlinear plans. Step addition is a very basic method dating back to GPS [Newell and Simon, 1963], where means-ends analysis was used to pick operators with postconditions corresponding to desired states. Figure 13.8 lists step addition along with other heuristics we use throughout this example.

To achieve the preconditions of the two steps above, we can use step addition again:

* CLEAR(A)	* CLEAR(B)
ONTABLE(A)	ONTABLE(B)
* ARMEMPTY	* ARMEMPTY
PICKUP(A)	PICKUP(B)
¬ONTABLE(A)	¬ONTABLE(B)
¬ARMEMPTY	¬ARMEMPTY
HOLDING(A)	HOLDING(B)

Adding these PICKUP steps is not enough to satisfy the *HOLDING preconditions of the STACK steps. This is because there are no ordering constraints present among the steps. If, in the eventual plan, the PICKUP steps were to follow the STACK steps, then the *HOLDING preconditions would need to be satisfied by some other set of steps. We solve this problem by introducing ordering constraints whenever we employ step addition. In this case, we want to say that each PICKUP step should precede its corresponding STACK step³:

PICKUP(A) ← STACK(A, B)
 PICKUP(B) ← STACK(B, C)

We now have four (partially ordered) steps in our plan and four unachieved preconditions. *CLEAR(A) is unachieved because block A is not clear in the initial state. *CLEAR(B) is unachieved because although B is clear in the initial state, there exists a step STACK(A, B) with postcondition ¬CLEAR(B), and that step might precede the step with *CLEAR(B) as a precondition. To achieve precondition CLEAR(B), we use a second heuristic known as *promotion*. Promotion, first used by Sussman in his HACKER program [Sussman, 1975], amounts to posting a constraint that one step must precede another in the eventual plan. We can achieve CLEAR(B) by stating that the PICKUP(B) step must come before the STACK(A, B) step:

PICKUP(B) ← STACK(A, B)

Let's now turn to the two unachieved *ARMEMPTY preconditions [we deal with *CLEAR(A) a little later]. While the initial state has an empty arm, each of the two pickup operators contain ¬ARMEMPTY postconditions. Either operator could prevent the other from executing. We can use promotion to achieve at least one of the two preconditions:

PICKUP(B) ← PICKUP(A)

Since the initial situation contains an empty arm, and no step preceding PICKUP(B) could make it unempty, the preconditions of PICKUP(B) are all satisfied.

A third heuristic, called *declobbering*, can help achieve the *ARMEMPTY precondition in the PICKUP(A) step. PICKUP(B) asserts ¬ARMEMPTY, but if we can insert

³S₁ ← S₂ means that step S₁ must precede step S₂ in the eventual plan.

another step between PICKUP(B) and PICKUP(A) to reassert ARMEMPTY, then the precondition will be achieved. The STACK(B, C) does the trick, so we post another constraint:

$$\text{PICKUP(B)} \leftarrow \text{STACK(B, C)} \leftarrow \text{PICKUP(A)}$$

The step PICKUP(B) is said to "clobber" PICKUP(A)'s precondition. STACK(B, C) is said to "declobber" it. Declobbering was first used in the NOAH planner [Sacerdoti, 1975], and then in NONLIN. NOAH was the first nonlinear planner to make use of the heuristics we are discussing here. NOAH also used many other heuristics and was able to solve a number of difficult nonlinear planning problems. Still, there were some natural problems that NOAH could not solve. In particular, NOAH's inability to backtrack prevented it from finding many solutions. The NONLIN program included backtracking, but it also failed to solve many hard problems.

Back in our example, the only unachieved precondition left is *CLEAR(A), from the PICKUP(A) step. We can use step addition to achieve it:

$$\begin{array}{l} * \text{ON}(x, A) \\ * \text{CLEAR}(x) \\ * \text{ARMEMPTY} \\ \hline \text{UNSTACK}(x, A) \\ \hline \neg \text{ARMEMPTY} \\ \text{CLEAR}(A) \\ \text{HOLDING}(A) \\ \neg \text{ON}(x, A) \end{array}$$

We introduce the variable x because the only postcondition we are interested in is CLEAR(A). Whatever block is on top of A is irrelevant. Constraint posting allows us to create plans that are incomplete with respect to the order of the steps. Variables allow us to avoid committing to particular instantiations of operators.

Unfortunately, we now have three new unachieved preconditions. We can achieve ON(x , A) easily by constraining the value of x to be block C. This works because block C is on block A in the initial state. This heuristic is called *simple establishment*, and in its most general form, it allows us to state that two different propositions must be ultimately instantiated to the same proposition. In our case:

$$x = C \text{ in step } \text{UNSTACK}(x, A)$$

There are still steps that deny the preconditions CLEAR(C) and ARMEMPTY, but we can use promotion to take care of them:

$$\begin{array}{l} \text{UNSTACK}(x, A) \leftarrow \text{STACK(B, C)} \\ \text{UNSTACK}(x, A) \leftarrow \text{PICKUP(A)} \\ \text{UNSTACK}(x, A) \leftarrow \text{PICKUP(B)} \end{array}$$

Among the heuristics we have looked at so far, adding a new step is the most problematic because we must always check if the new step clobbers some precondition of a later, already existing step. This has actually happened in our example. The step PICKUP(B) requires ARMEMPTY, but this is denied by the new UNSTACK(x, A) step. One way to solve this problem is to add a new declobbering step to the plan:

```

HOLDING(C)
-----
PUTDOWN(C)
-----
-HOLDING(C)
ONTABLE(x)
ARMEMPTY
  
```

ordered as:

$$\text{UNSTACK}(x, A) \leftarrow \text{PUTDOWN}(C) \leftarrow \text{PICKUP}(B)$$

Notice that we have seen two types of declobbering, one in which an existing step is used to declobber another, and one in which a new declobbering step is introduced. Fortunately, the precondition of our newest PUTDOWN step is satisfied. In fact, all preconditions of all steps are satisfied, so we are done. All that remains is to use the plan ordering and variable binding constraints to build a concrete plan:

1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(B)
4. STACK(B, C)
5. PICKUP(A)
6. STACK(A, B)

This is the same plan we found at the end of Section 13.4. We used four different heuristics to synthesize it: step addition, promotion, declobbering, and simple establishment. (These are sometimes called *plan modification operations*.) Are these four operations, applied in the correct order, enough to solve any nonlinear planning problem? Almost. We require one more, called *separation*. Separation is like simple establishment, in that it concerns variable bindings, but it is used in a declobbering fashion. Suppose step C_1 possibly precedes step C_2 and C_1 possibly denies a precondition of C_2 . We say "possibly" because the propositions may contain variables. Separation allows us to state a constraint that the two propositions must *not* be instantiated in the same way in the eventual plan.

Work on the TWEAK planner presented formal definitions of the five plan modification operations and proved that they were sufficient for solving *any* solvable nonlinear planning problem. In this manner, TWEAK cleaned up the somewhat *ad hoc*, heuristic results in nonlinear planning research. The algorithm to exploit the plan modification operations is quite simple.

Algorithm: Nonlinear Planning (TWEAK)

1. Initialize S to be the set of propositions in the goal state.
2. Remove some unachieved proposition P from S .
3. Achieve P by using step addition, promotion, declobbering, simple establishment, or separation.
4. Review all the steps in the plan, including any new steps introduced by step addition, to see if any of their preconditions are unachieved. Add to S the new set of unachieved preconditions.
5. If S is empty, complete the plan by converting the partial order of steps into a total order, and instantiate any variables as necessary.
6. Otherwise, go to step 2.

Of course, not every sequence of plan modification operations leads to a solution. For instance, we could use step addition *ad infinitum* without ever converging to a useful plan. The nondeterminism of steps 2 and 3 must be implemented as some sort of search procedure. This search can be guided by heuristics; for example, if promotion and step addition will both do the job, it is probably better to try promotion first. TWEAK uses breadth-first dependency-directed backtracking, as well as ordering heuristics.

The example above used most of the plan modification operations, but not in their full generality. We will now be more specific about these operations and how they relate to finding correct plans. The core notion is one of making a proposition *necessarily* true in some state. The *modal truth criterion* tells us exactly when a proposition is true.

The Modal Truth Criterion. A proposition P is necessarily true in a state S if and only if two conditions hold: there is a state T equal or necessarily previous to S in which P is necessarily asserted; and for every step C possibly before S and every proposition Q possibly codesignating⁴ with P which C denies, there is a step W necessarily between C and S which asserts R , a proposition such that R and P codesignate whenever P and Q codesignate.

Roughly, this means that P has to be asserted in the initial state or by some previous step and that there can be no clobbering steps without corresponding declobbering steps to save the day. The relationship between the modal truth criterion and the five plan modification operations is shown in Figure 13.9. The figure is simply a logical parse tree of the criterion, from which we can see how the plan modification operations are used to enforce the truth of various parts of the criterion. In the figure, the expression $C_1 \approx C_2$ means step (or state) C_1 necessarily precedes step (or state) C_2 . The expression $P \approx Q$ means P and Q codesignate.

The development of a provably correct planner was a noteworthy achievement in the formal (or "neat") style of AI. It cleaned up the complicated, ill-defined planning notions that preceded it and made available a reliable (if not efficient) planner. Now, however, a new round of more informal (or "scruffy") research must follow, concentrating on

⁴Two propositions *codesignate* if they can be unified, given the current constraints on variables.

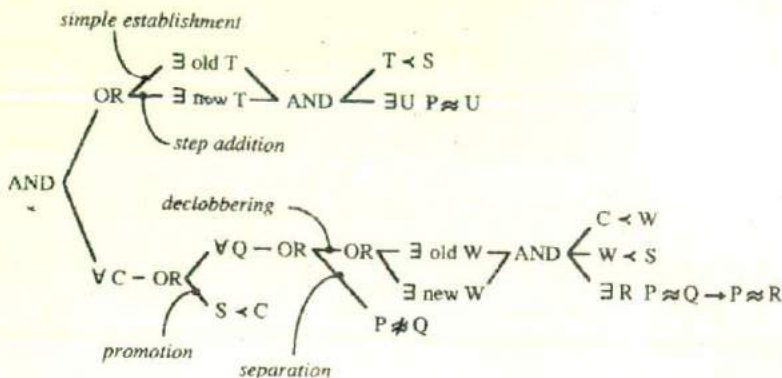


Figure 13.9: The Modal Truth Criterion for Telling whether Proposition P Necessarily Holds in State S

the weaknesses of such planners. Efficiency is of critical concern in large systems—assured correctness is nice, but a slow planner can be less useful than an incorrect one. Typically, search-based programs can be made faster through the use of heuristic knowledge. Another efficiency issue has to do with the *length* of the plans produced by a planner. Current planners can, unfortunately, generate highly inefficient plans.

Representational issues are just as important as efficiency issues, and the two are closely intertwined. The representation of operators and plans used by TWEAK is at the same time too powerful and too weak. Chapman [1987] proved that even with simple STRIPS-style operators, planning in general is not even decidable, although it is semidecidable: If there is a plan that solves a problem, a planner can find it, but if there is no such plan, the planner may never halt. NP-completeness results suggest that planning is exponentially hard. But it is of no use to look for a simpler representation that might allow for more efficient plan construction—if anything, most domains seem to require operators that are much more complex than the operators used by TWEAK. For example, it is natural to express many preconditions using quantifiers and embedded negation and also to have postconditions with different effects depending on the state of the world. Figure 13.10 depicts a more complex operator structure, of the type used in the PRODIGY planning system [Minton *et al.*, 1989]. As our representation becomes more expressive, the idea of a provably correct, efficient, domain-independent planner becomes more unlikely, and we must again turn to knowledge-intensive heuristic methods.

13.6 Hierarchical Planning

In order to solve hard problems, a problem solver may have to generate long plans. In order to do that efficiently, it is important to be able to eliminate some of the details of


```

(OPERATOR
  (PRECONDITIONS
    (and (...))
    (forall (w ...) ...)
    (not
      (exists ...)
      (or (...))))
  (POSTCONDITIONS
    (ADD (...))
    (DELETE (...))
    (if (and (...)) (...))
      (ADD (...)) (...))
    (DELETE (...)) (...))))

```

Figure 13.10. A Complex Operator

the problem until a solution that addresses the main issues is found. Then an attempt can be made to fill in the appropriate details. Early attempts to do this involved the use of macro-operators, in which larger operators were built from smaller ones [Fikes and Nilsson, 1971]. But in this approach, no details were eliminated from the actual descriptions of the operators. A better approach was developed in the ABSTRIPS system [Sacerdoti, 1974], which actually planned in a hierarchy of *abstraction spaces*, in each of which preconditions at a lower level of abstraction were ignored.

As an example, suppose you want to visit a friend in Europe, but you have a limited amount of cash to spend. It makes sense to check air fares first, since finding an affordable flight will be the most difficult part of the task. You should not worry about getting out of your driveway, planning a route to the airport, or parking your car until you are sure you have a flight.

The ABSTRIPS approach to problem solving is as follows: First solve the problem completely, considering only preconditions whose *criticality value* is the highest possible. These values reflect the expected difficulty of satisfying the precondition. To do this, do exactly what STRIPS did, but simply ignore preconditions of lower than peak criticality. Once this is done, use the constructed plan as the outline of a complete plan and consider preconditions at the next-lowest criticality level. Augment the plan with operators that satisfy those preconditions. Again, in choosing operators, ignore all preconditions whose criticality is less than the level now being considered. Continue this process of considering less and less critical preconditions until all of the preconditions of the original rules have been considered. Because this process explores entire plans at one level of detail before it looks at the lower-level details of any one of them, it has been called *length-first search*.

Clearly, the assignment of appropriate criticality values is crucial to the success of this hierarchical planning method. Those preconditions that no operators can satisfy are clearly the most critical. For example, if we are trying to solve a problem involving a robot moving around in a house and we are considering the operator PUSH-THROUGH-DOOR, the precondition that there exist a door big enough for the robot to get through

is of high criticality since there is (in the normal situation) nothing we can do about it if it is not true. But the precondition that the door be open is of lower criticality if we have the operator OPEN-DOOR. In order for a hierarchical planning system to work with STRIPS-like rules, it must be told, in addition to the rules themselves, the appropriate criticality value for each term that may occur in a precondition. Given these values, the basic process can function in very much the same way that nonhierarchical planning does. But effort will not be wasted filling in the details of plans that do not even come close to solving the problem.

13.7 Reactive Systems

So far, we have described a deliberative planning process, in which a plan for completing an entire task is constructed prior to action. There is a very different way, though, that we could approach the problem of deciding what to do. The idea of *reactive systems* [Brooks, 1986; Agre and Chapman, 1987; Kaelbling, 1987] is to avoid planning altogether, and instead use the observable situation as a clue to which one can simply react.

A reactive system must have access to a knowledge base of some sort that describes what actions should be taken under what circumstances. A reactive system is very different from the other kinds of planning systems we have discussed because it chooses actions one at a time; it does not anticipate and select an entire action sequence before it does the first thing.

One of the very simplest reactive systems is a thermostat. The job of a thermostat is to keep the temperature constant inside a room. One might imagine a solution to this problem that requires significant amounts of planning, taking into account how the external temperature rises and falls during the day, how heat flows from room to room, and so forth. But a real thermostat uses the simple pair of situation-action rules:

1. If the temperature in the room is k degrees above the desired temperature, then turn the air conditioner on
2. If the temperature in the room is k degrees below the desired temperature, then turn the air conditioner off

It turns out that reactive systems are capable of surprisingly complex behaviors, especially in real world tasks such as robot navigation. We discuss robot tasks in more detail in Chapter 21. The main advantage reactive systems have over traditional planners is that they operate robustly in domains that are difficult to model completely and accurately. Reactive systems dispense with modeling altogether and base their actions directly on their perception of the world. In complex and unpredictable domains, the ability to plan an exact sequence of steps ahead of time is of questionable value. Another advantage of reactive systems is that they are extremely responsive, since they avoid the combinatorial explosion involved in deliberative planning. This makes them attractive for real time tasks like driving and walking.

Of course, many AI tasks do require significant deliberation, which is usually implemented as internal search. Since reactive systems maintain no model of the world and no explicit goal structures, their performance in these tasks is limited. For example, it seems unlikely that a purely reactive system could ever play expert chess. It is possible

to provide a reactive system with rudimentary planning capability, but only by explicitly storing whole plans along with the situations that should trigger them. Deliberative planners need not rely on pre-stored plans; they can construct a new plan for each new problem.

Nevertheless, inquiry into reactive systems has served to illustrate many of the shortcomings of traditional planners. For one thing, it is vital to interleave planning and plan execution. Planning is important, but so is action. An intelligent system with limited resources must decide when to start thinking, when to stop thinking, and when to act. Also, goals arise naturally when the system interacts with the environment. Some mechanism for suspending plan execution is needed so that the system can turn its attention to high priority goals. Finally, some situations require immediate attention and rapid action. For this reason, some deliberative planners [Mitchell, 1990] compile out reactive subsystems (i.e., sets of situation-action rules) based on their problem-solving experiences. Such systems learn to be reactive over time.

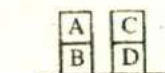
13.8 Other Planning Techniques

Other planning techniques that we have not discussed include the following.

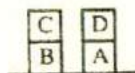
- Triangle Tables [Fikes *et al.*, 1972; Nilsson, 1980]—Provide a way of recording the goals that each operator is expected to satisfy as well as the goals that must be true for it to execute correctly. If something unexpected happens during the execution of a plan, the table provides the information required to patch the plan.
- Metaplanning [Stefik, 1981a]—A technique for reasoning not just about the problem being solved but also about the planning process itself.
- Macro-operators [Fikes and Nilsson, 1971]—Allow a planner to build new operators that represent commonly used sequences of operators. See Chapter 17 for more details.
- Case-Based Planning [Hammond, 1986]—Re-uses old plans to make new ones. We return to case-based planning in Chapter 19.

13.9 Exercises

1. Consider the following blocks world problem:



start: $ON(C, B) \wedge$
 $ON(D, A) \wedge$
 $ONTABLE(B) \wedge$
 $ONTABLE(A) \wedge$
 $ARMEMPTY$



goal: $ON(C, B) \wedge$
 $ON(D, A) \wedge$
 $ONTABLE(B) \wedge$
 $ONTABLE(A)$

- (a) Show how STRIPS would solve this problem.
- (b) Show how TWEAK would solve this problem.
- (c) Did these processes produce optimal plans? If not, could they be modified to do so?
2. Consider the problem of devising a plan for cleaning the kitchen.
- (a) Write a set of STRIPS-style operators that might be used. When you describe the operators, take into account such considerations as:
- Cleaning the stove or the refrigerator will get the floor dirty.
 - To clean the oven, it is necessary to apply oven cleaner and then to remove the cleaner.
 - Before the floor can be washed, it must be swept.
 - Before the floor can be swept, the garbage must be taken out.
 - Cleaning the refrigerator generates garbage and messes up the counters.
 - Washing the counters or the floor gets the sink dirty.
- (b) Write a description of a likely initial state of a kitchen in need of cleaning. Also write a description of a desirable (but perhaps rarely obtained) goal state.
- (c) Show how the technique of planning using a goal stack could be used to solve this problem. (Hint—you may want to modify the definition of an ADD condition so that when a condition is added to the database, its negation is automatically deleted if present.)
3. In Section 13.4, we showed an example of a situation in which a search path could be terminated because it led back to one of its earlier goals. Describe a mechanism by which a program could detect this situation.
4. Consider the problem of swapping the contents of two registers, A and B. Suppose that there is available the single operator $ASSIGN(x, v, lv, ov)$, which assigns the value v , which is stored in location lv , to location x , which previously contained the value ov :

$$\begin{aligned}
 &ASSIGN(x, v, lv, ov) \\
 &P: CONTAINS(lv, v) \wedge CONTAINS(x, ov) \\
 &D: CONTAINS(x, ov) \\
 &A: CONTAINS(x, v)
 \end{aligned}$$

Assume that there is at least one additional register, C, available.

- (a) What would STRIPS do with this problem?
- (b) What would TWEAK do with this problem?
- (c) How might you design a program to solve this problem?

Chapter 14

Understanding

14.1 What Is Understanding?

To understand something is to transform it from one representation into another, where this second representation has been chosen to correspond to a set of available actions that could be performed and where the mapping has been designed so that for each event, an *appropriate* action will be performed. There is very little absolute in the notion of understanding. If you say to an airline database system "I need to go to New York as soon as possible," the system will have "understood" if it finds the first available plane to New York. If you say the same thing to your best friend, who knows that your family lives in New York, she will have "understood" if she realizes that there may be a problem in your family and you may need some emotional support. As we talk about understanding, it is important to keep in mind that the success or failure of an "understanding" program can rarely be measured in an absolute sense but must instead be measured with respect to a particular task to be performed. This is true both of language-understanding programs and also of understanders in other domains, such as vision.

For people, understanding applies to inputs from all the senses. Computer understanding has so far been applied primarily to images, speech, and typed language. In this chapter we discuss issues that cut across all of these modalities. In Chapter 15, we explore the problem of typed natural language in more detail, and in Chapter 21, we look at speech and vision problems. Although we have defined understanding above as the process of mapping into appropriate *actions*, we are not precluding a view of understanding in which inputs are simply interpreted and stored for later. In such a system, the appropriate action is to store the proper representation. This view of understanding describes what occurs in most image understanding programs and some language understanding programs. Taking direct action describes what happens in systems in which language, either typed or spoken, is used in the interface between user and computer.

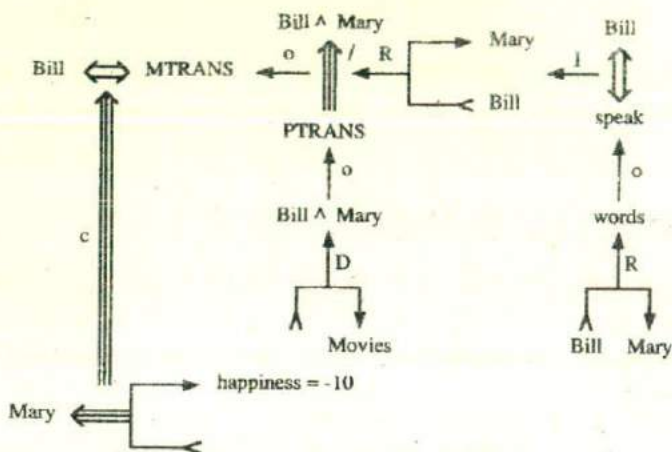


Figure 14.1: The Conceptual Dependency Representation of a Paragraph

14.2 What Makes Understanding Hard?

There are four major factors that contribute to the difficulty of an understanding problem:

1. The complexity of the target representation into which the matching is being done
2. The type of the mapping: one-one, many-one, one-many, or many-many
3. The level of interaction of the components of the source representation
4. The presence of noise in the input to the understander

A few examples will illustrate the importance of each of these factors.

Complexity of the Target Representation

Suppose English sentences are being used for communication with a keyword-based data retrieval system. Then the sentence

I want to read all about the last Presidential election.

would need to be translated into a representation such as

(SEARCH KEYWORDS = ELECTION & PRESIDENT)

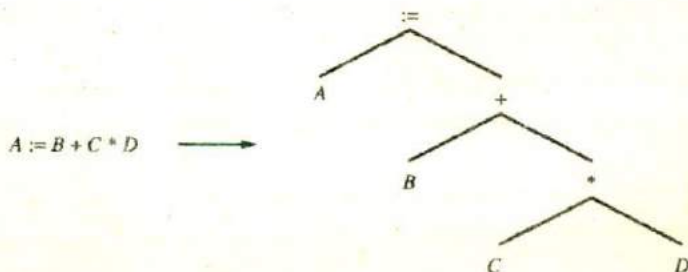
But now suppose that English sentences are being used to provide input to a program that records events so that it can answer a variety of questions about those events and their relationships. For example, consider the following story:

Bill told Mary he would not go to the movies with her.
Her feelings were hurt.

The result of understanding this story could be represented, using the conceptual dependency model that we discussed in Chapter 10, as shown in Figure 14.1. This representation is considerably more complex than that for the simple query. All other things being equal, constructing such a complex representation is more difficult than constructing a simple one since more information must be extracted from the input sentences. Extracting that information often requires the use of additional knowledge about the world described by the sentences.

Type of Mapping

Recall that understanding is the process of mapping an input from its original form to a more useful one. The simplest kind of mapping to deal with is one-to-one (i.e., each different statement maps to a single target representation that is different from that arising from any other statement). Very few input systems are totally one-to-one. But as an example of an almost one-to-one mapping, consider the language of arithmetic expressions in many programming languages. In such a language, a mapping such as the following might occur:



Although one-to-one mappings are, in general, the simplest to perform, they are rare in interesting input systems for several reasons. One important reason is that in many domains, inputs must be interpreted not absolutely, but relatively, with respect to some reference point. For example, when images are being interpreted, size and perspective will change as a function of the viewing position. Thus a single object will look different in different images. To see this, look at Figure 14.2, which shows two line drawings representing the same scene, one of which corresponds to a picture taken close to the scene and one of which represents a picture taken from farther away. A similar phenomenon occurs in English. The word "tall" specifies one height range in the phrase "a tall giraffe" and a different one in the phrase "a tall poodle."

A second reason that many-to-one mappings are frequent is that free variation is often allowed, either because of the physical limitations of the system that produces the inputs or because such variation simply makes the task of generating the inputs manageable. Both of these factors help to explain why natural languages, both in their spoken and their written forms, require many-to-one mappings. Examples from speech abound. No two people speak identically. In fact, one person does not always say a given word the

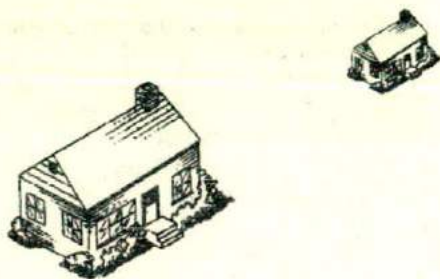


Figure 14.2: Relative Differences in Pictures of the Same Scene

same way. Figure 14.3 illustrates this problem. It shows a spectrogram produced by the beginning of the utterance "Alpha gets alpha minus beta." A spectrogram shows how the sound energy is distributed over the auditory frequency range as a function of time. In this example, you can see two different patterns, each produced by the word "alpha." Even when we ignore the variability of the speech signal, natural languages admit variability because of their richness. This is particularly noticeable when mapping from a natural language (with its richness of structure and vocabulary) to a small, simple target representation. So, for example, we might find many-to-one mappings, such as the following one, occurring in the English front end to a keyword data retrieval system:

Tell me all about the last presidential election.	→	
		(SEARCH
I'd like to see all the stories on the last presidential election.	→	KEYWORDS =
		ELECTION
		&
		PRESIDENT)
I am interested in the last presidential election.	→	

Many-to-one mappings require that the understanding system know about all the ways that a target representation can be expressed in the source language. As a result, they typically require a structured analysis of the input rather than a simple, exact pattern match. But they often do not require much other knowledge.

One-to-many mappings, on the other hand, often require a great deal of domain knowledge (in addition to the input itself) in order to make the correct choice among the available target representations. An example of such a mapping (in which the input can be said to be *ambiguous*) is the following sentence:

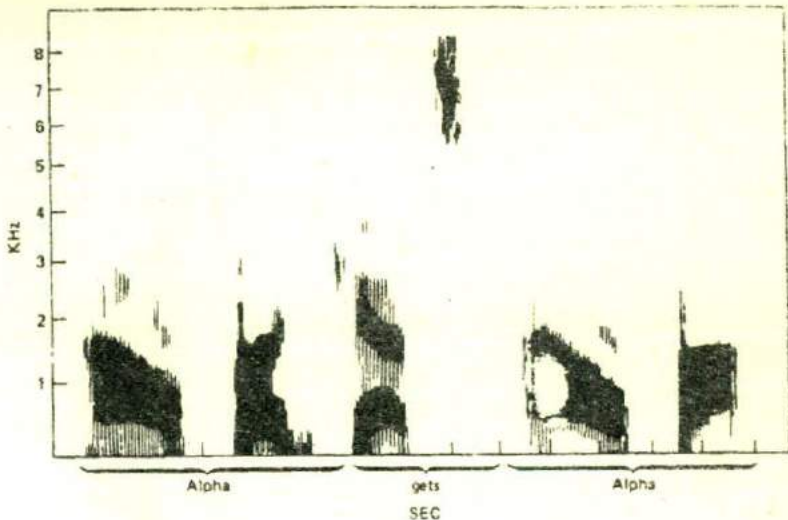


Figure 14.3: Differences in Speech Signals

- They are flying planes.
- (They are (flying airplanes))
 - (They (are flying) airplanes)
 - (They are (flying planing-tools))
 - (They (are flying) planing-tools)

Notice that although this sentence, taken in isolation, is ambiguous, it would usually not be interpreted as being ambiguous by a human listener in a specific context. Clues both from previous sentences and from the physical context in which the sentence occurs, usually make one of these interpretations appear to be correct. The problem, though, from a processing standpoint, is how to encode this contextual information and how to exploit it while processing each new sentence.

Notice that English, in all its glory, has the properties of both of these last two examples; it involves a *many-to-many* mapping, in which there are many ways to say the same thing and a given statement may have many meanings.

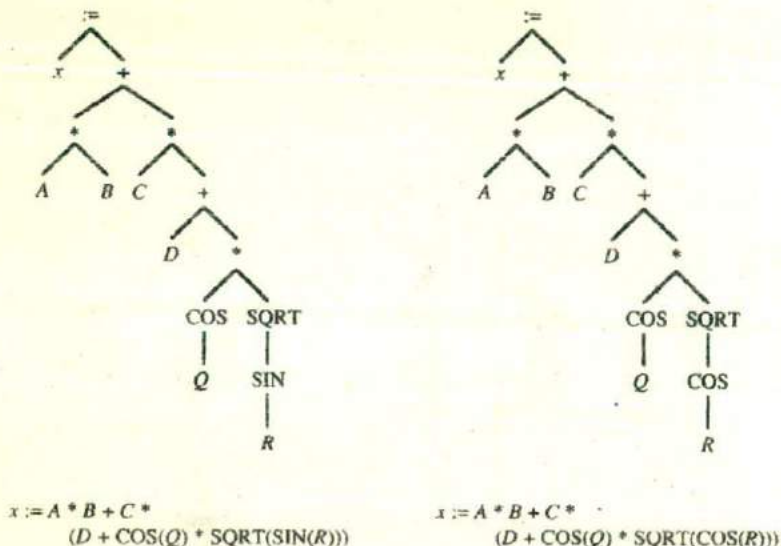


Figure 14.4: Little Interaction among Components

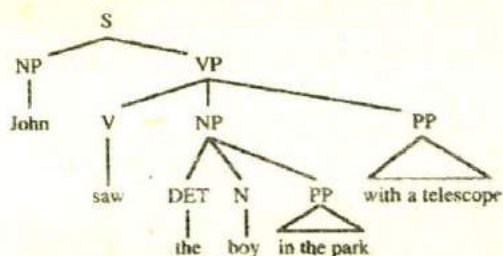
Level of Interaction among Components

In most interesting understanding contexts, each input is composed of several components (lines, words, symbols, or whatever). The mapping process is the simplest if each component can be mapped without concern for the other components of the statement. Otherwise, as the number of interactions increases, so does the complexity of the mapping.

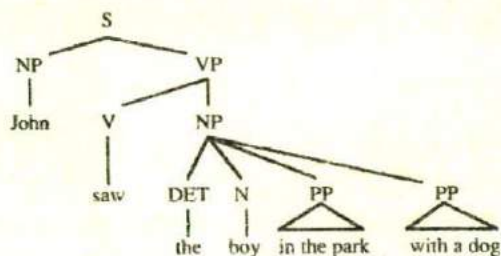
Programming languages provide good examples of languages in which there is very little interaction among the components of an input. For example, Figure 14.4 shows how changing one word of a statement requires only a single change to one node of the corresponding parse tree.

In many natural language sentences, on the other hand, changing a single word can alter not just a single node of the interpretation, but rather its entire structure. An example of this is shown in Figure 14.5. (The triangles in the figure indicate substructures whose further decomposition is not important.) As these examples show, the components of an English sentence typically interact more heavily with each other than do the components of artificial languages, such as programming languages, that have been designed, among other things, to facilitate processing by computer.

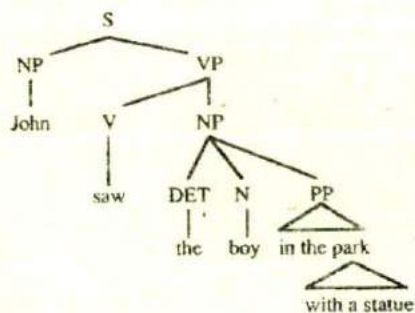
Nonlocality can be a problem at all levels of an understanding process. In the boy in the park example, the problem is in how to group phrases together. But in perceptual understanding tasks, this same problem may make it difficult even to decide



John saw the boy in the park with a telescope.



John saw the boy in the park with a dog.



John saw the boy in the park with a statue.

Figure 14.5: More Interaction among Components

The cat scares all the birds away.
 k a t s k a r s
 A cat's cares are few.

Figure 14.6: Local Ambiguity in a Speech Problem

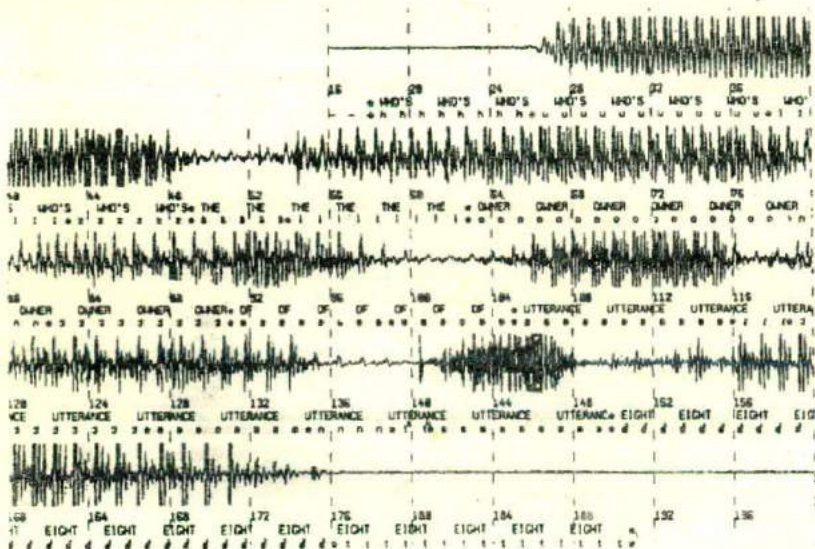


Figure 14.7: A Speech Waveform

on what the basic constituents are. Figure 14.6 shows a simplified example from speech understanding. Assuming that the sounds shown in the figure have been identified, the problem is to group them into words. But the correct grouping cannot be determined without looking at the larger context in which the sounds occurred. Either of the groupings shown is possible, as can be seen from the two sentences in the figure. Figure 14.7 shows an actual speech waveform, in which the lack of local clues, even for segmenting into individual sounds, can be seen.

In image-understanding problems as well, a similar problem involving local indeterminacy arises. Consider the situation shown in Figure 14.8. At this point, lines have been extracted from the original figure and the next task is to separate the figure into objects. But suppose we start at the left and identify the object labeled A. Does it end at the vertical line? It is not possible to tell without looking past the vertical object to see if there is an extension which, in this case, there is.

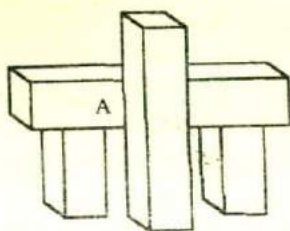


Figure 14.8: A Line Drawing with Local Ambiguity

Noise in the Input

Understanding is the process of interpreting an input and assigning it meaning. Unfortunately, in many understanding situations the input to which meaning should be assigned is not always the input that is presented to the understander. Because of the complex environment in which understanding usually occurs, other things often interfere with the basic input before it reaches the understander. In perceptual tasks, such as speech and image understanding, this problem is common. We rarely have the opportunity to listen to each other against a background of silence. Thus we must take an input signal and separate the speech component from the background noise component in order to understand the speech. The same problem occurs in image understanding. If you look out of your car window in search of a particular store sign, the image you will see of the sign may be interfered with by many things, such as your windshield wipers or the trees alongside the road. Although typed language is less susceptible to noise than is spoken language, noise is still a problem. For example, typing errors are common, particularly if language is being used interactively to communicate with a computer system.

Conclusion

The point of this section has been twofold. On the one hand, it has attempted to describe the sources of complexity in understanding tasks, in order to help you analyze new understanding tasks for tractability. On the other, it has tried to point out specific understanding tasks that turn out, unfortunately, to be hard (such as natural language understanding) but that are nevertheless important in the sense that it would be useful if we could perform them. It is to these understanding tasks that we will need to devote substantial research effort.

14.3 Understanding as Constraint Satisfaction

On the basis of a superficial analysis (such as the one in the last section), many understanding tasks appear impossibly complex. The number of interpretations that can be assigned to individual components of an input is large, and the number of combinations of those components is enormous. But a closer analysis often reveals that many of the combinations cannot actually occur. These natural constraints can be exploited in the understanding process to reduce the complexity from unmanageable to tractable. There are two important steps in the use of constraints in problem solving:

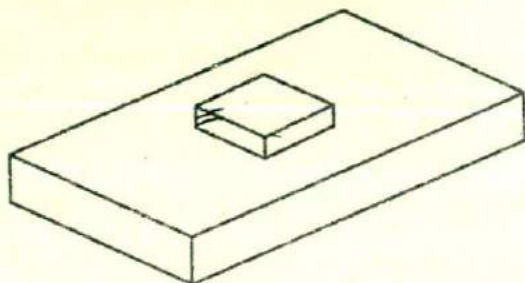


Figure 14.9: A Line Drawing

1. Analyze the problem domain to determine what the constraints are.
2. Solve the problem by applying a constraint satisfaction algorithm that effectively uses the constraints from step 1 to control the search. Recall that we presented such an algorithm in Section 3.5.

In the rest of this section, we look at one example of the use of this approach, the Waltz algorithm for labeling line drawings. In Chapter 15 we then look in depth at the problem of natural language understanding and see how it too can be viewed as a constraint satisfaction process.

Consider the drawing shown in Figure 14.9. Assume either that you have been given this drawing as the input or that lower-level routines have already operated to extract these lines from an input photograph. The next step in the analysis process is to determine the objects described by the lines. To do this, we need first to identify each of the lines in the figure as representing either:

- An Obscuring Edge—A boundary between objects, or between objects and the background
- A Concave Edge—An edge between two faces that form an acute angle when viewed from outside the object
- A Convex Edge—An edge between two faces that form an obtuse angle when viewed from outside the object

For more complex figures, other edge types, such as cracks between coplanar faces and shadow edges between shadows and the background, would also be required. The approach we describe here has, in fact, been extended to handle these other edge types. But to make the explanation straightforward, we consider only these three. In fact, we consider only figures composed exclusively of *trihedral* vertices, which are vertices at which exactly three planes come together. Figure 14.10 shows examples of trihedral figures. Figure 14.11 shows examples of nontrihedral figures.

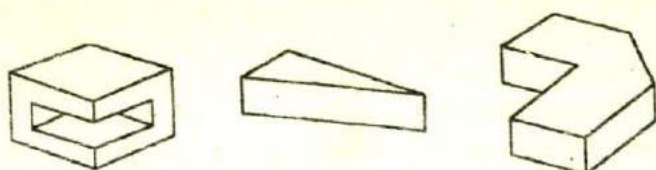


Figure 14.10: Some Trihedral Figures

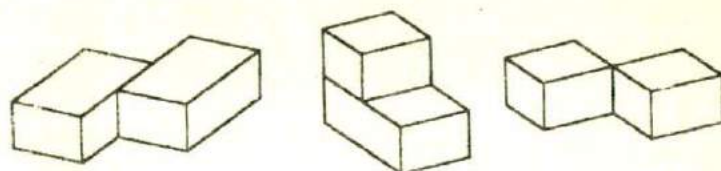


Figure 14.11: Some Nontrihedral Figures

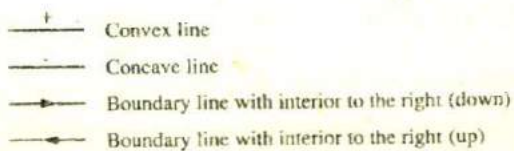


Figure 14.12: Line-Labeling Conventions

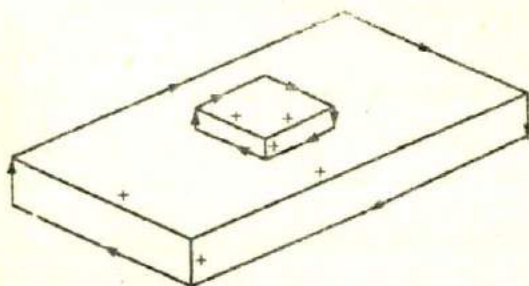


Figure 14.13: An Example of Line Labeling

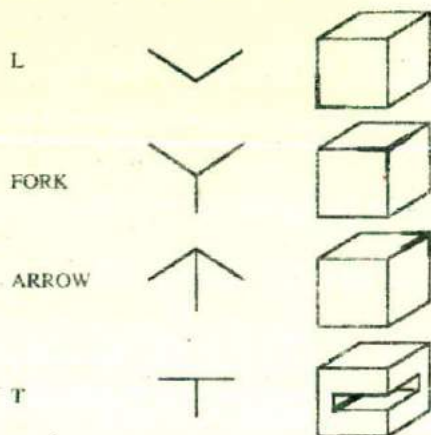


Figure 14.14: The Four Trihedral Vertex Types

Determining the Constraints

The problem we are trying to solve is how to recognize individual objects in a figure. To do that, we intend first to label all the lines in the figure so that we know which ones correspond to boundaries between objects. We use the three line types given above. For boundary lines, we also need to indicate a direction, telling which side of the line corresponds to the object and which to the background. This produces a set of four labels that can be attached to a given line. We use the conventions shown in Figure 14.12 to show line labelings. To illustrate these labelings, Figure 14.13 shows the drawing of Figure 14.9 with each of its lines correctly labeled.

Assuming these four line types, we can calculate that the number of ways of labeling a figure composed of N lines is 4^N . How can we find the correct one? The critical observation here is that every line must meet other lines at a vertex at each of its ends. For the trihedral figures we are considering, there are only four configurations that describe all the possible vertices. These four configurations are shown in Figure 14.14. The rotational position of the vertex is not significant, nor are the sizes of the angles it contains, except that the distinction between acute angles (< 90 degrees) and obtuse angles (> 90 degrees) is important to distinguish between a FORK and an ARROW. If there turn out to be constraints on the kinds of vertices that can occur, then there would be corresponding constraints on the lines entering the vertices and thus the number of possible line labelings would be reduced.

To begin looking for such vertex constraints, we first consider the maximum number of ways that each of the four types of lines might combine with other lines at a vertex. Since an L vertex involves two lines, each of which can have four labels, there must be sixteen ways it could be formed. FORKS, Ts, and ARROWS involve three lines, so they could be formed in sixty-four ways each. Thus there are 208 ways to form a trihedral vertex. But, in fact, only a very small number of these labelings can actually occur in line drawings representing real physical objects. To see this, consider the planes on which the faces that form a vertex of a trihedral figure lie. These three planes must

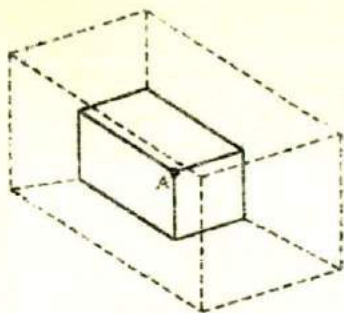


Figure 14.15: A Figure Occupying One Octant

divide 3-space into eight parts (called octants) since each individual face divides the space in half and none of the faces can be coplanar. Trihedral figures may differ in the number of octants that they fill and in the position (which must be one of the unfilled octants) from which they are viewed. Any vertex that can occur in a trihedral figure must correspond to such a division of space with some number (between one and eight) of octants filled, which is viewed from one of the unfilled octants. So to find all the vertex labelings that can occur, we need only consider all the ways of filling the octants and each of the ways of viewing those fillings, and then record the types of the vertices that we find.

To illustrate this process, consider the drawing shown in Figure 14.15, which occupies one of the eight octants formed by the intersection of the planes corresponding to the faces of vertex A. Imagine viewing this figure from each of the remaining seven octants and recording the configuration and the labeling of vertex A. Figure 14.16(a) shows the results of this. When we take those seven descriptions and eliminate rotational and angular variations, we see that only three distinct ones remain, as shown in Figure 14.16(b). If we continue this process for objects filling up to seven octants (there can be no vertices if all eight octants are filled), we get a complete list of the possible trihedral vertices and their labelings (equivalent to that developed by Clowes [1971]). This list is shown in Figure 14.17. Notice that of the 208 labelings that we said were theoretically possible, only eighteen are physically possible. Thus we have found a severe constraint on the way that lines in drawings corresponding to real figures can be labeled.

Of course, at this point we have only found a constraint on the ways in which simple trihedral vertices can be labeled. Many figures, such as those shown in Figure 14.11, contain nontrihedral vertices. In addition, many figures contain shadow areas, which can be of great use in analyzing the scene that is being portrayed. When these variations are considered, there do become more than eighteen allowable vertex labelings. But when these variations are allowed, the number of theoretically possible labelings becomes much larger than 208, and, in fact, the ratio of physically allowable vertices to theoretically possible ones becomes even smaller than $18/208$. Thus not only can this approach be extended to larger domains, it must be.

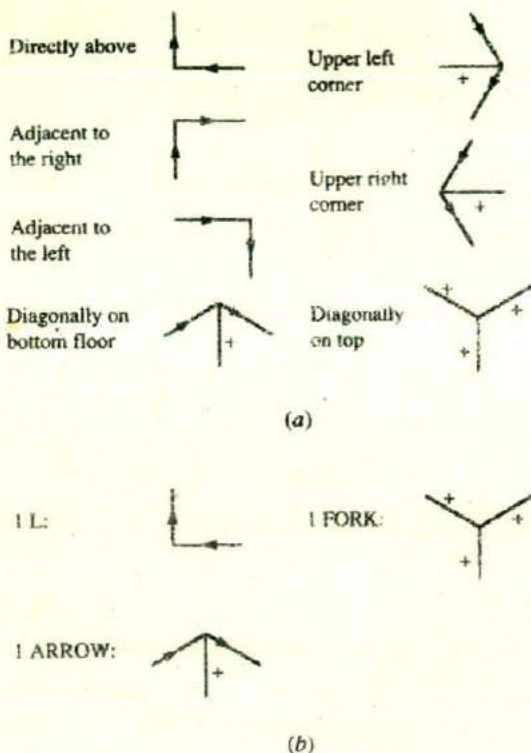


Figure 14.16: The Vertices of a Figure Occupying One Octant

As a result of this analysis, we have been able to articulate one class of constraints that will be needed by a line-labeling procedure. These constraints are static (since the physical rules they are based on never change), and so they do not need to be represented explicitly as part of a problem state. They can be encoded directly into the line-labeling algorithm. The other class of constraints we will need contains the dynamic ones that describe the current options for the labeling of each vertex. These constraints will be represented and manipulated explicitly by the line-labeling algorithm.

Applying Constraints in Analysis Problems

Having analyzed the domain in which we are working and extracted a set of constraints that objects in the domain must satisfy, we need next to apply those constraints to the problem of analyzing inputs in the domain. To do this, we use a form of the constraint satisfaction procedure described in Section 3.5. It turns out that for this problem it is

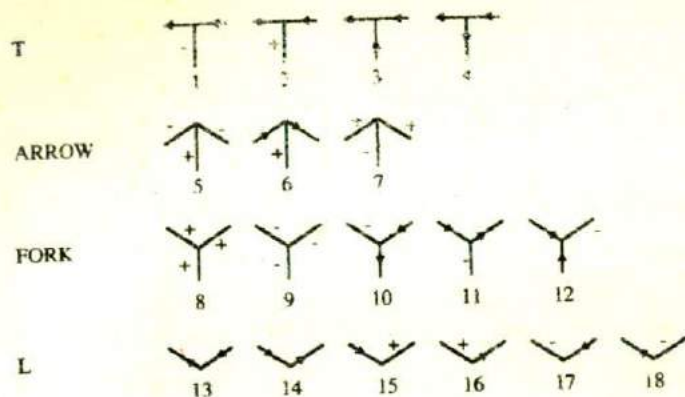


Figure 14.17: The Eighteen Physically Possible Trihedral Vertices

not necessary to use the second part of our constraint satisfaction procedure (the one that makes guesses and results in search). The domain provides sufficiently powerful constraints that it is not necessary to resort to search. Thus the *Waltz algorithm* [Waltz, 1975], which we present here, omits that step entirely.

To label line drawings of the sort we are considering, we first pick one vertex and find all the labelings that are possible for it. Then we move to an adjacent vertex and find all of its possible labelings. The line that we followed to get from the first vertex to the second must end up with only one label, and that label must be consistent with the two vertices it enters. So any labelings for either of the two vertices that require the line to be labeled in a way that is inconsistent with the other vertex can be eliminated. Now another vertex, adjacent to one of the first two, can be labeled. New constraints will arise from this labeling and these constraints can be propagated back to vertices that have already been labeled, so the set of possible labelings for them is further reduced. This process proceeds until all the vertices in the figure have been labeled.

As an example, consider the simple drawing shown in Figure 14.18(a). We can begin by labeling all the boundary edges, as shown in Figure 14.18(b). Suppose we then begin labeling vertices at vertex 1. The only vertex label that is consistent with the known line labels is 13. At vertex 2, the only consistent label is 6. At each of the remaining boundary vertices, there is also only one labeling choice. These labelings are shown in parentheses in Figure 14.18(c). Now consider vertex 7. Just looking at vertex 7 itself, it would appear that any of the five FORK labelings is possible. But from the only labeling we found for vertex 2, we know that the line between vertices 2 and 7 must be labeled +. This makes sense since it obviously represents a convex edge. Using this fact, we can eliminate four of the possible FORK labels. Only label 8 is now possible. The complete labeling just computed is shown in Figure 14.18(d). Thus we see that by exploiting constraints on vertex labelings, we have correctly identified vertex 7 as being formed by three convex edges.

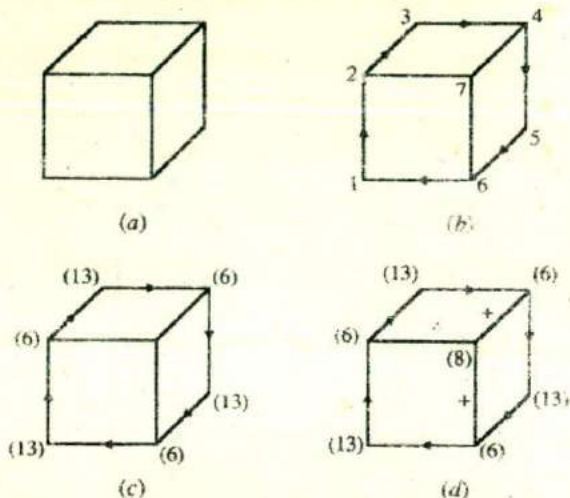


Figure 14.18: A Simple Example of the Labeling Process

We can now specify in more detail this particular version of constraint propagation.

Algorithm: Waltz

1. Find the lines at the border of the scene boundary and label them. These lines can be found by finding an outline such that no vertices are outside it. We do this first because this labeling will impose additional constraints on the other labelings in the figure.
2. Number the vertices of the figure to be analyzed. These numbers will correspond to the order in which the vertices will be visited during the labeling process. To decide on a numbering, do the following:
 - (a) Start at any vertex on the boundary of the figure. Since boundary lines are known, the vertices involving them are more highly constrained than are interior ones.
 - (b) Move from the vertex along the boundary to an adjacent unnumbered vertex and continue until all boundary vertices have been numbered.
 - (c) Number interior vertices by moving from a numbered vertex to some adjacent unnumbered one. By always labeling a vertex next to one that has already been labeled, maximum use can be made of the constraints.
3. Visit each vertex V in order and attempt to label it by doing the following:

- (a) Using the set of possible vertex labelings given in Figure 14.17, attach to V a list of its possible labelings.
- (b) See whether some of these labelings can be eliminated on the basis of local constraints. To do this, examine each vertex A that is adjacent to V and that has already been visited. Check to see that for each proposed labeling for V , there is a way to label the line between V and A in such a way that at least one of the labelings listed for A is still possible. Eliminate from V 's list any labeling for which this is not the case.
- (c) Use the set of labelings just attached to V to constrain the labelings at vertices adjacent to V . For each vertex A that was visited in the last step, do the following:
 - i. Eliminate all labelings of A that are not consistent with at least one labeling of V .
 - ii. If any labelings were eliminated, continue constraint propagation by examining the vertices adjacent to A and checking for consistency with the restricted set of labelings now attached to A .
 - iii. Continue to propagate until there are no adjacent labeled vertices or until there is no change made to the existing set of labelings.

This algorithm will always find the unique, correct figure labeling if one exists. If a figure is ambiguous, however, the algorithm will terminate with at least one vertex still having more than one labeling attached to it.

Actually, this algorithm, as described by Waltz, was applied to a larger class of figures in which cracks and shadows might occur. But the operation of the algorithm is the same regardless of the size of the table of allowable vertex labelings that it uses. In fact, as suggested in the last section, the usefulness of the algorithm increases as the size of the domain increases and thus the ratio of physically possible to theoretically possible vertices decreases. Waltz's program, for example, used shadow information, which appears in the figure locally as shadow lines, as a way of exploiting a global constraint, namely that a single source of light produces consistent shadows.

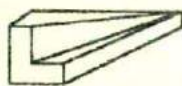
14.4 Summary

In this chapter we outlined the major difficulties that confront programs designed to perform perceptual tasks. We also described the use of the constraint satisfaction procedure as one way of surmounting some of those difficulties.

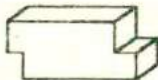
Sometimes the problems of speech and image understanding are important in the construction of stand-alone programs to solve one particular task. But they also play an important role in the larger field of *robotics*, which has as its goal the construction of intelligent robots capable of functioning with some degree of autonomy. For such robots, perceptual abilities are essential. We will return to these issues in Chapter 21.

14.5 Exercises

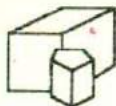
- One of the reasons that understanding complex perceptual patterns is difficult is that if the pattern is composed of more than one object, a variety of difficult-to-predict phenomena may occur at the junctions between objects. For example, when the phrase "Could you go?" is spoken, a *j* sound appears between the words, "could" and "you." Give another example of boundary interference in speech. Also give one example of it in vision.
- Which of the following figures are trihedral?



(a)



(b)



(c)



(d)

- In Section 14.3, we analyzed all the ways that a vertex of a trihedral object that occupies one octant of the space formed by the intersection of its planes could be labeled. Complete this analysis for vertices of objects that occupy two through seven octants.
- For each of the drawings in Figure 14.10, show how the Waltz algorithm would produce a labeling.
- In our description of the Waltz algorithm, we first assigned to each vertex V all the labelings that might be attached to it. Then we looked at all adjacent vertices in an attempt to constrain the set of labelings associated with V . And then we went back to each adjacent vertex A to see if the knowledge about V could be used to further constrain the labelings for A . Why could we not simply visit each adjacent vertex once and perform both these steps then?
- Give an example of an ambiguous figure for which the Waltz algorithm would not find a unique labeling.

Chapter 15

Natural Language Processing

Language is meant for communicating about the world. By studying language, we can come to understand more about the world. We can test our theories about the world by how well they support our attempt to understand language. And, if we can succeed at building a computational model of language, we will have a powerful tool for communicating about the world. In this chapter, we look at how we can exploit knowledge about the world, in combination with linguistic facts, to build computational natural language systems.

Throughout this discussion, it is going to be important to keep in mind that the difficulties we will encounter do not exist out of perversity on the part of some diabolical designer. Instead, what we see as difficulties when we try to analyze language are just the flip sides of the very properties that make language so powerful. Figure 15.1 shows some examples of this. As we pursue our discussion of language processing, it is important to keep the good sides in mind since it is because of them that language is significant enough a phenomenon to be worth all the trouble.

By far the largest part of human linguistic communication occurs as speech. Written language is a fairly recent invention and still plays a less central role than speech in most activities. But processing written language (assuming it is written in unambiguous characters) is easier, in some ways, than processing speech. For example, to build a program that understands spoken language, we need all the facilities of a written language understander as well as enough additional knowledge to handle all the noise and ambiguities of the audio signal.¹ Thus it is useful to divide the entire language-processing problem into two tasks:

- Processing written text, using lexical, syntactic, and semantic knowledge of the language as well as the required real world information
- Processing spoken language, using all the information needed above plus additional knowledge about phonology as well as enough added information to handle the further ambiguities that arise in speech

¹Actually, in understanding spoken language, we take advantage of clues, such as intonation and the presence of pauses, to which we do not have access when we read. We can make the task of a speech-understanding program easier by allowing it, too, to use these cues, but to do so, we must know enough about them to incorporate into the program knowledge of how to use them.

The Problem: English sentences are incomplete descriptions of the information that they are intended to convey:

Some dogs are outside.



Some dogs are on the lawn.

Three dogs are on the lawn.

Rover, Tripp, and Spot are on the lawn.

I called Lynda to ask her
to the movies.

She said she'd love to go.



She was home when I called.

She answered the phone.

I actually asked her.

The Good Side: Language allows speakers to be as vague or as precise as they like. It also allows speakers to leave out things they believe their hearers already know.

The Problem: The same expression means different things in different contexts:

Where's the water? (in a chemistry lab, it must be pure)

Where's the water? (when you are thirsty, it must be potable)

Where's the water? (dealing with a leaky roof, it can be filthy)

The Good Side: Language lets us communicate about an infinite world using a finite (and thus learnable) number of symbols.

The Problem: No natural language program can be complete because new words, expressions, and meanings can be generated quite freely:

I'll fax it to you.

The Good Side: Language can evolve as the experiences that we want to communicate about evolve.

The Problem: There are lots of ways to say the same thing:

Mary was born on October 11.

Mary's birthday is October 11.

The Good Side: When you know a lot, facts imply each other. Language is intended to be used by agents who know a lot.

Figure 15.1: Features of Language That Make It Both Difficult and Useful

In Chapter 14 we described some of the issues that arise in speech understanding, and in Section 21.2.2 we return to them in more detail. In this chapter, though, we concentrate on written language processing (usually called simply *natural language processing*).

Throughout this discussion of natural language processing, the focus is on English. This happens to be convenient and turns out to be where much of the work in the field has occurred. But the major issues we address are common to all natural languages. In fact, the techniques we discuss are particularly important in the task of translating from one natural language to another.

Natural language processing includes both understanding and generation, as well as other tasks such as multilingual translation. In this chapter we focus on understanding, although in Section 15.5 we will provide some references to work in these other areas.

15.1 Introduction

Recall that in the last chapter we defined understanding as the process of mapping from an input form into a more immediately useful form. It is this view of understanding that we pursue throughout this chapter. But it is useful to point out here that there is a formal sense in which a language can be defined simply as a set of strings without reference to any world being described or task to be performed. Although some of the ideas that have come out of this formal study of languages can be exploited in parts of the understanding process, they are only the beginning. To get the overall picture, we need to think of language as a pair (source language, target representation), together with a mapping between elements of each to the other. The target representation will have been chosen to be appropriate for the task at hand. Often, if the task has clearly been agreed on and the details of the target representation are not important in a particular discussion, we talk just about the language itself, but the other half of the pair is really always present.

One of the great philosophical debates throughout the centuries has centered around the question of what a sentence means. We do not claim to have found the definitive answer to that question. But once we realize that understanding a piece of language involves mapping it into some representation appropriate to a particular situation, it becomes easy to see why the questions "What is language understanding?" and "What does a sentence mean?" have proved to be so difficult to answer. We use language in such a wide variety of situations that no single definition of understanding is able to account for them all. As we set about the task of building computer programs that understand natural language, one of the first things we have to do is define precisely what the underlying task is and what the target representation should look like. In the rest of this chapter, we assume that our goal is to be able to reason with the knowledge contained in the linguistic expressions, and we exploit a frame language as our target representation.

15.1.1 Steps in the Process

Before we go into detail on the several components of the natural language understanding process, it is useful to survey all of them and see how they fit together. Roughly, we can break the process down into the following pieces:

- **Morphological Analysis**—Individual words are analyzed into their components, and nonword tokens, such as punctuation, are separated from the words.
- **Syntactic Analysis**—Linear sequences of words are transformed into structures that show how the words relate to each other. Some word sequences may be rejected if they violate the language's rules for how words may be combined. For example, an English syntactic analyzer would reject the sentence "Boy the go the to store."
- **Semantic Analysis**—The structures created by the syntactic analyzer are assigned meanings. In other words, a mapping is made between the syntactic structures and objects in the task domain. Structures for which no such mapping is possible may be rejected. For example, in most universes, the sentence "Colorless green ideas sleep furiously" [Chomsky, 1957] would be rejected as *semantically anomalous*.
- **Discourse Integration**—The meaning of an individual sentence may depend on the sentences that precede it and may influence the meanings of the sentences that follow it. For example, the word "it" in the sentence, "John wanted it," depends on the prior discourse context, while the word "John" may influence the meaning of later sentences (such as, "He always had.")
- **Pragmatic Analysis**—The structure representing what was said is reinterpreted to determine what was actually meant. For example, the sentence "Do you know what time it is?" should be interpreted as a request to be told the time.

The boundaries between these five phases are often very fuzzy. The phases are sometimes performed in sequence, and they are sometimes performed all at once. If they are performed in sequence, one may need to appeal for assistance to another. For example, part of the process of performing the syntactic analysis of the sentence "Is the glass jar peanut butter?" is deciding how to form two noun phrases out of the four nouns at the end of the sentence (giving a sentence of the form "Is the x y ?"). All of the following constituents are syntactically possible: glass, glass jar, glass jar peanut, jar peanut butter, peanut butter, butter. A syntactic processor on its own has no way to choose among these, and so any decision must be made by appealing to some model of the world in which some of these phrases make sense and others do not. If we do this, then we get a syntactic structure in which the constituents "glass jar" and "peanut butter" appear. Thus although it is often useful to separate these five processing phases to some extent, they can all interact in a variety of ways, making a complete separation impossible.

Specifically, to make the overall language understanding problem tractable, it will help if we distinguish between the following two ways of decomposing a program:

- The processes and the knowledge required to perform the task
- The global control structure that is imposed on those processes

In this chapter, we focus primarily on the first of these issues. It is the one that has received the most attention from people working on this problem. We do not completely ignore the second issue, although considerably less of substance is known about it. For

an example of this kind of discussion that talks about interleaving syntactic and semantic processing, see Lytinen [1986].

With that caveat, let's consider an example to see how the individual processes work. In this example, we assume that the processes happen sequentially. Suppose we have an English interface to an operating system and the following sentence is typed:

I want to print Bill's .init file.

Morphological Analysis

Morphological analysis must do the following things:

- Pull apart the word "Bill's" into the proper noun "Bill" and the possessive suffix "'s".
- Recognize the sequence ".init" as a file extension that is functioning as an adjective in the sentence

in addition, this process will usually assign syntactic categories to all the words in the sentence. This is usually done now because interpretations for affixes (prefixes and suffixes) may depend on the syntactic category of the complete word. For example, consider the word "prints." This word is either a plural noun (with the "-s" marking plural) or a third person singular verb (as in "he prints"), in which case the "-s" indicates both singular and third person. If this step is done now, then in our example, there will be ambiguity since "want," "print," and "file" can all function as more than one syntactic category.

Syntactic Analysis

Syntactic analysis must exploit the results of morphological analysis to build a structural description of the sentence. The goal of this process, called *parsing*, is to convert the flat list of words that forms the sentence into a structure that defines the units that are represented by that flat list. For our example sentence, the result of parsing is shown in Figure 15.2. The details of this representation are not particularly significant; we describe alternative versions of them in Section 15.2. What is important here is that a flat sentence has been converted into a hierarchical structure and that that structure has been designed to correspond to sentence units (such as noun phrases) that will correspond to meaning units when semantic analysis is performed. One useful thing we have done here, although not all syntactic systems do, is create a set of entities we call *reference markers*. They are shown in parentheses in the parse tree. Each one corresponds to some entity that has been mentioned in the sentence. These reference markers are useful later since they provide a place in which to accumulate information about the entities as we get it. Thus although we have not tried to do semantic analysis (i.e., assign meaning) at this point, we have designed our syntactic analysis process so that it will find constituents to which meaning can be assigned.

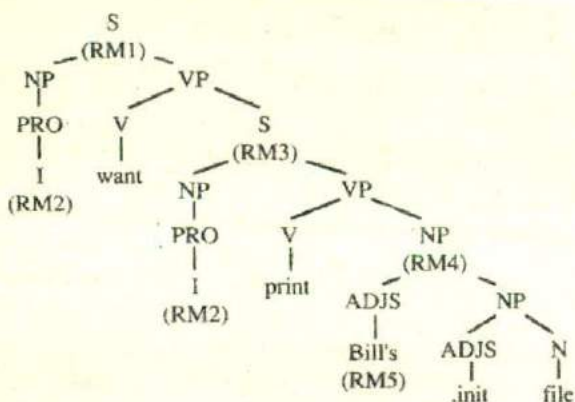


Figure 15.2: The Result of Syntactic Analysis of "I want to print Bill's .init file."

Semantic Analysis

Semantic analysis must do two important things:

- It must map individual words into appropriate objects in the knowledge base or database.
- It must create the correct structures to correspond to the way the meanings of the individual words combine with each other.

For this example, suppose that we have a frame-based knowledge base that contains the units shown in Figure 15.3. Then we can generate a partial meaning, with respect to that knowledge base, as shown in Figure 15.4. Reference marker *RM1* corresponds to the top-level event of the sentence. It is a wanting event in which the speaker (denoted by "I") wants a printing event to occur in which the same speaker prints a file whose extension is ".init" and whose owner is Bill.

Discourse Integration

At this point, we have figured out what kinds of things this sentence is about. But we do not yet know which specific individuals are being referred to. Specifically, we do not know to whom the pronoun "I" or the proper noun "Bill" refers. To pin down these references requires an appeal to a model of the current discourse context, from which we can learn that the current user (who typed the word "I") is *User068* and that the only

<i>User</i>	
<i>isa</i> :	<i>Person</i>
* <i>login-name</i> :	must be <string>
<i>User068</i>	
<i>instance</i> :	<i>User</i>
<i>login-name</i> :	<i>Susan-Black</i>
<i>User073</i>	
<i>instance</i> :	<i>User</i>
<i>login-name</i> :	<i>Bill-Smith</i>
<i>F1</i>	
<i>instance</i> :	<i>File-Struct</i>
<i>name</i> :	stuff
<i>extension</i> :	.init
<i>owner</i> :	<i>User073</i>
<i>in-directory</i> :	/wsmith/
<i>File-Struct</i>	
<i>isa</i> :	<i>Information-Object</i>
<i>Printing</i>	
<i>isa</i> :	<i>Physical-Event</i>
* <i>agent</i> :	must be <animate or program>
* <i>object</i> :	must be <information-object>
<i>Wanting</i>	
<i>isa</i> :	<i>Mental-Event</i>
* <i>agent</i> :	must be <animate>
* <i>object</i> :	must be <state or event>
<i>Commanding</i>	
<i>isa</i> :	<i>Mental-Event</i>
* <i>agent</i> :	must be <animate>
* <i>performer</i> :	must be <animate or program>
* <i>object</i> :	must be <event>
<i>This-System</i>	
<i>instance</i> :	<i>Program</i>

Figure 15.3: A Knowledge Base Fragment

<i>RM1</i>		{the whole sentence}
<i>instance :</i>	<i>Wanting</i>	
<i>agent :</i>	<i>RM2</i>	{I}
<i>object :</i>	<i>RM3</i>	{a printing event}
<i>RM2</i>		{I}
<i>RM3</i>		{a printing event}
<i>instance :</i>	<i>Printing</i>	
<i>agent :</i>	<i>RM2</i>	{I}
<i>object :</i>	<i>RM4</i>	{Bill's .init file}
<i>RM4</i>		{Bill's .init file}
<i>instance :</i>	<i>File-Struct</i>	
<i>extension :</i>	<i>.init</i>	
<i>owner :</i>	<i>RM5</i>	{Bill}
<i>RM5</i>		{Bill}
<i>instance :</i>	<i>Person</i>	
<i>first-name :</i>	<i>Bill</i>	

Figure 15.4: A Partial Meaning for a Sentence

person named "Bill" about whom we could be talking is *User073*. Once the correct referent for Bill is known, we can also determine exactly which file is being referred to: *F1* is the only file with the extension ".init" that is owned by Bill.

Pragmatic Analysis

We now have a complete description, in the terms provided by our knowledge base, of what was said. The final step toward effective understanding is to decide what to do as a result. One possible thing to do is to record what was said as a fact and be done with it. For some sentences, whose intended effect is clearly declarative, that is precisely the correct thing to do. But for other sentences, including this one, the intended effect is different. We can discover this intended effect by applying a set of rules that characterize cooperative dialogues. In this example, we use the fact that when the user claims to want something that the system is capable of performing, then the system should go ahead and do it. This produces the final meaning shown in Figure 15.5.

The final step in pragmatic processing is to translate, when necessary, from the knowledge-based representation to a command to be executed by the system. In this case, this step is necessary, and we see that the final result of the understanding process is

```
lpr /wsmith/stuff.init
```


<i>Meaning</i>	
<i>instance :</i>	<i>Commanding</i>
<i>agent :</i>	<i>User068</i>
<i>performer :</i>	<i>This-System</i>
<i>object :</i>	<i>P27</i>
<i>P27</i>	
<i>instance :</i>	<i>Printing</i>
<i>agent :</i>	<i>This-System</i>
<i>object :</i>	<i>F1</i>

Figure 15.5: Representing the Intended Meaning

where "lpr" is the operating system's file print command.

Summary

At this point, we have seen the results of each of the main processes that combine to form a natural language system. In a complete system, all of these processes are necessary in some form. For example, it may have seemed that we could have skipped the knowledge-based representation of the meaning of the sentence since the final output of the understanding system bore no relationship to it. But it is that intermediate knowledge-based representation to which we usually attach the knowledge that supports the creation of the final answer.

All of the processes we have described are important in a complete natural language understanding system. But not all programs are written with exactly these components. Sometimes two or more of them are collapsed, as we will see in several sections later in this chapter. Doing that usually results in a system that is easier to build for restricted subsets of English but one that is harder to extend to wider coverage. In the rest of this chapter we describe the major processes in more detail and talk about some of the ways in which they can be put together to form a complete system.

15.2 Syntactic Processing

Syntactic processing is the step in which a flat input sentence is converted into a hierarchical structure that corresponds to the units of meaning in the sentence. This process is called *parsing*. Although there are natural language understanding systems that skip this step (for example, see Section 15.3.3), it plays an important role in many natural language understanding systems for two reasons:

- Semantic processing must operate on sentence constituents. If there is no syntactic parsing step, then the semantics system must decide on its own constituents. If parsing is done, on the other hand, it constrains the number of constituents that

semantics can consider. Syntactic parsing is computationally less expensive than is semantic processing (which may require substantial inference). Thus it can play a significant role in reducing overall system complexity.

- Although it is often possible to extract the meaning of a sentence without using grammatical facts, it is not always possible to do so. Consider, for example, the sentences
 - The satellite orbited Mars.
 - Mars orbited the satellite.

In the second sentence, syntactic facts demand an interpretation in which a planet (Mars) revolves around a satellite, despite the apparent improbability of such a scenario.

Although there are many ways to produce a parse, almost all the systems that are actually used have two main components:

- A declarative representation, called a *grammar*, of the syntactic facts about the language
- A procedure, called a *parser*, that compares the grammar against input sentences to produce parsed structures

15.2.1 Grammars and Parsers

The most common way to represent grammars is as a set of production rules. Although details of the forms that are allowed in the rules vary, the basic idea remains the same and is illustrated in Figure 15.6, which shows a simple context-free, phrase structure grammar for English. Read the first rule as, "A sentence is composed of a noun phrase followed by a verb phrase." In this grammar, the vertical bar should be read as "or." The ϵ denotes the empty string. Symbols that are further expanded by rules are called *nonterminal symbols*. Symbols that correspond directly to strings that must be found in an input sentence are called *terminal symbols*.

Grammar formalisms such as this one underlie many linguistic theories, which in turn provide the basis for many natural language understanding systems. Modern linguistic theories include: the government binding theory of Chomsky [1981; 1986], GPSG [Gazdar *et al.*, 1985], LFG [Bresnan, 1982], and categorial grammar [Ades and Steedman, 1982; Oehrle *et al.*, 1987]. The first three of these are also discussed in Sells [1986]. We should point out here that there is general agreement that pure, context-free grammars are not effective for describing natural languages.² As a result, natural language processing systems have less in common with computer language processing systems (such as compilers) than you might expect.

Regardless of the theoretical basis of the grammar, the parsing process takes the rules of the grammar and compares them against the input sentence. Each rule that matches adds something to the complete structure that is being built for the sentence.

²There is, however, still some debate on whether context-free grammars are formally adequate for describing natural languages (e.g., Gazdar [1982].)

$S \rightarrow NP VP$
 $NP \rightarrow \text{the } NP1$
 $NP \rightarrow \text{PRO}$
 $NP \rightarrow \text{PN}$
 $NP \rightarrow \text{NP1}$
 $NP1 \rightarrow \text{ADJS } N$
 $\text{ADJS} \rightarrow \epsilon \mid \text{ADJ } \text{ADJS}$
 $VP \rightarrow V$
 $VP \rightarrow V NP$
 $N \rightarrow \text{file} \mid \text{printer}$
 $\text{PN} \rightarrow \text{Bill}$
 $\text{PRO} \rightarrow I$
 $\text{ADJ} \rightarrow \text{short} \mid \text{long} \mid \text{fast}$
 $V \rightarrow \text{printed} \mid \text{created} \mid \text{want}$

Figure 15.6: A Simple Grammar for a Fragment of English

The simplest structure to build is a *parse tree*, which simply records the rules and how they are matched. Figure 15.7 shows the parse tree that would be produced for the sentence "Bill printed the file" using this grammar. Figure 15.2 contained another example of a parse tree, although some additions to this grammar would be required to produce it.

Notice that every node of the parse tree corresponds either to an input word or to a nonterminal in our grammar. Each level in the parse tree corresponds to the application of one grammar rule. As a result, it should be clear that a grammar specifies two things about a language:

- Its weak generative capacity, by which we mean the set of sentences that are contained within the language. This set (called the set of *grammatical sentences*) is made up of precisely those sentences that can be completely matched by a series of rules in the grammar.
- Its strong generative capacity, by which we mean the structure (or possibly structures) to be assigned to each grammatical sentence of the language.

So far, we have shown the result of parsing to be exactly a trace of the rules that were applied during it. This is not always the case, though. Some grammars contain additional information that describes the structure that should be built. We present an example of such a grammar in Section 15.2.2.

But first we need to look at two important issues that define the space of possible parsers that can exploit the grammars we write.

Top-Down versus Bottom-Up Parsing

To parse a sentence, it is necessary to find a way in which that sentence could have been generated from the start symbol. There are two ways that this can be done:

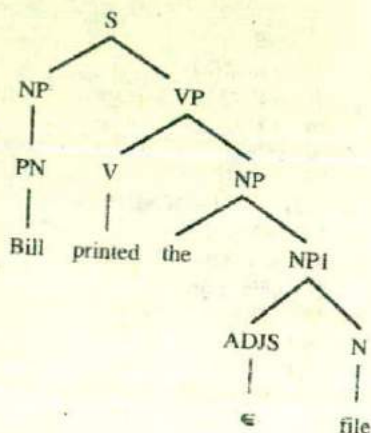


Figure 15.7: A Parse Tree for a Sentence

- *Top-Down Parsing*—Begin with the start symbol and apply the grammar rules forward until the symbols at the terminals of the tree correspond to the components of the sentence being parsed.
- *Bottom-Up Parsing*—Begin with the sentence to be parsed and apply the grammar rules backward until a single tree whose terminals are the words of the sentence and whose top node is the start symbol has been produced.

The choice between these two approaches is similar to the choice between forward and backward reasoning in other problem-solving tasks. The most important consideration is the branching factor. Is it greater going backward or forward? Another important issue is the availability of good heuristics for evaluating progress. Can partial information be used to rule out paths early? Sometimes these two approaches are combined into a single method called *bottom-up parsing with top-down filtering*. In this method, parsing proceeds essentially bottom-up (i.e., the grammar rules are applied backward). But using tables that have been precomputed for a particular grammar, the parser can immediately eliminate constituents that can never be combined into useful higher-level structures.

Finding One Interpretation or Finding Many

As several of the examples above have shown, the process of understanding a sentence is a search process in which a large universe of possible interpretations must be explored to find one that meets all the constraints imposed by a particular sentence. As for any search process, we must decide whether to explore all possible paths or, instead, to explore only a single most likely one and to produce only the result of that one path as the answer.

Suppose, for example, that a sentence processor looks at the words of an input sentence one at a time, from left to right, and suppose that so far, it has seen:

"Have the students who missed the exam—"

There are two paths that the processor could be following at this point:

- "Have" is the main verb of an imperative sentence, such as
"Have the students who missed the exam take it today."
- "Have" is an auxiliary verb of an interrogative sentence, such as
"Have the students who missed the exam taken it today?"

There are four ways of handling sentences such as these:

- *All Paths*—Follow all possible paths and build all the possible intermediate components. Many of the components will later be ignored because the other inputs required to use them will not appear. For example, if the auxiliary verb interpretation of "have" in the previous example is built, it will be discarded if no participle, such as "taken," ever appears. The major disadvantage of this approach is that, because it results in many spurious constituents being built and many deadend paths being followed, it can be very inefficient.
- *Best Path with Backtracking*—Follow only one path at a time, but record, at every choice point, the information that is necessary to make another choice if the chosen path fails to lead to a complete interpretation of the sentence. In this example, if the auxiliary verb interpretation of "have" were chosen first and the end of the sentence appeared with no main verb having been seen, the understander would detect failure and backtrack to try some other path. There are two important drawbacks to this approach. The first is that a good deal of time may be wasted saving state descriptions at each choice point, even though backtracking will occur to only a few of those points. The second is that often the same constituent may be analyzed many times. In our example, if the wrong interpretation is selected for the word "have," it will not be detected until after the phrase "the students who missed the exam" has been recognized. Once the error is detected, a simple backtracking mechanism will undo everything that was done after the incorrect interpretation of "have" was chosen, and the noun phrase will be reinterpreted (identically) after the second interpretation of "have" has been selected. This problem can be avoided using some form of dependency-directed backtracking, but then the implementation of the parser is more complex.
- *Best Path with Patchup*—Follow only one path at a time, but when an error is detected, explicitly shuffle around the components that have already been formed. Again, using the same example, if the auxiliary verb interpretation of "have" were chosen first, then the noun phrase "the students who missed the exam" would be interpreted and recorded as the subject of the sentence. If the word "taken" appears next, this path can simply be continued. But if "take" occurs next, the understander can simply shift components into different slots. "Have" becomes the main verb. The noun phrase that was marked as the subject of the sentence becomes the subject of the embedded sentence "The students who missed the exam take it today." And the subject of the main sentence can be filled in as

"you," the default subject for imperative sentences. This approach is usually more efficient than the previous two techniques. Its major disadvantage is that it requires interactions among the rules of the grammar to be made explicit in the rules for moving components from one place to another. The interpreter often becomes *ad hoc*, rather than being simple and driven exclusively from the grammar.

- *Wait and See*—Follow only one path, but rather than making decisions about the function of each component as it is encountered, procrastinate the decision until enough information is available to make the decision correctly. Using this approach, when the word "have" of our example is encountered, it would be recorded as some kind of verb whose function is, as yet, unknown. The following noun phrase would then be interpreted and recorded simply as a noun phrase. Then, when the next word is encountered, a decision can be made about how all the constituents encountered so far should be combined. Although several parsers have used some form of wait-and-see strategy, one, PARSIFAL [Marcus, 1980], relies on it exclusively. It uses a small, fixed-size buffer in which constituents can be stored until their purpose can be decided upon. This approach is very efficient, but it does have the drawback that if the amount of lookahead that is necessary is greater than the size of the buffer, then the interpreter will fail. But the sentences on which it fails are exactly those on which people have trouble, apparently because they choose one interpretation, which proves to be wrong. A classic example of this phenomenon, called the *garden path sentence*, is

The horse raced past the barn fell down.

Although the problems of deciding which paths to follow and how to handle backtracking are common to all search processes, they are complicated in the case of language understanding by the existence of genuinely ambiguous sentences, such as our earlier example "They are flying planes." If it is important that not just one interpretation but rather all possible ones be found, then either all possible paths must be followed (which is very expensive since most of them will die out before the end of the sentence) or backtracking must be forced (which is also expensive because of duplicated computations). Many practical systems are content to find a single plausible interpretation. If that interpretation is later rejected, possibly for semantic or pragmatic reasons, then a new attempt to find a different interpretation can be made.

Parser Summary

As this discussion suggests, there are many different kinds of parsing systems. There are three that have been used fairly extensively in natural language systems:

- Chart parsers [Winograd, 1983], which provide a way of avoiding backup by storing intermediate constituents so that they can be reused along alternative parsing paths.
- Definite clause grammars [Pereira and Warren, 1980], in which grammar rules are written as PROLOG clauses and the PROLOG interpreter is used to perform top-down, depth-first parsing.

- Augmented transition networks (or ATNs) [Woods, 1970], in which the parsing process is described as the transition from a start state to a final state in a transition network that corresponds to a grammar of English.

We do not have space here to go into all these methods. In the next section, we illustrate the main ideas involved in parsing by working through an example with an ATN. After this, we look at one way of parsing with a more declarative representation.

15.2.2 Augmented Transition Networks

An augmented transition network (ATN) is a top-down parsing procedure that allows various kinds of knowledge to be incorporated into the parsing system so it can operate efficiently. Since the early use of the ATN in the LUNAR system [Woods, 1973], which provided access to a large database of information on lunar geology, the mechanism has been exploited in many language-understanding systems. The ATN is similar to a finite state machine in which the class of labels that can be attached to the arcs that define transitions between states has been augmented. Arcs may be labeled with an arbitrary combination of the following:

- Specific words, such as "in."
- Word categories, such as "noun."
- Pushes to other networks that recognize significant components of a sentence. For example, a network designed to recognize a prepositional phrase (PP) may include an arc that asks for ("pushes for") a noun phrase (NP).
- Procedures that perform arbitrary tests on both the current input and on sentence components that have already been identified.
- Procedures that build structures that will form part of the final parse.

Figure 15.8 shows an example of an ATN in graphical notation. Figure 15.9 shows the top-level ATN of that example in a notation that a program could read. To see how an ATN works, let us trace the execution of this ATN as it parses the following sentence:

The long file has printed.

This execution proceeds as follows:

1. Begin in state S
2. Push to NP.
3. Do a category test to see if "the" is a determiner.
4. This test succeeds, so set the DETERMINER register to DEFINITE and go to state Q6.
5. Do a category test to see if "long" is an adjective.

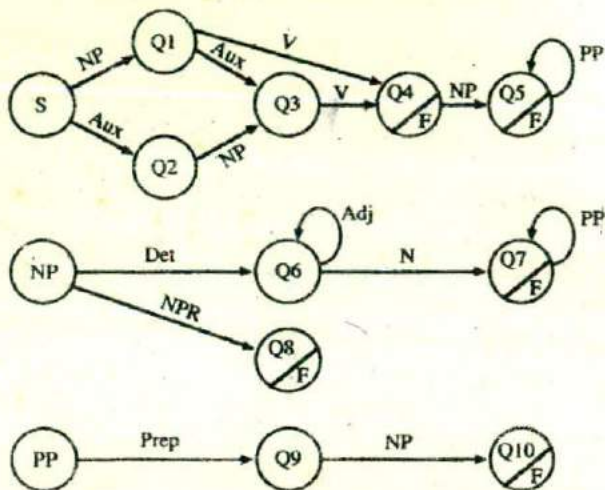


Figure 15.8: An ATN Network for a Fragment of English

6. This test succeeds, so append "long" to the list contained in the ADJS register (This list was previously empty.) Stay in state Q6.
7. Do a category test to see if "file" is an adjective. This test fails.
8. Do a category test to see if "file" is a noun. This test succeeds, so set the NOUN register to "file" and go to state Q7.
9. Push to PP.
10. Do a category test to see if "has" is a preposition. This test fails, so pop and signal failure.
11. There is nothing else that can be done from state Q7, so pop and return the structure (NP (FILE (LONG) DEFINITE))
The return causes the machine to be in state Q1, with the SUBJ register set to the structure just returned and the TYPE register set to DCL.
12. Do a category test to see if "has" is a verb. This test succeeds, so set the AUX register to NIL and set the V register to "has." Go to state Q4.
13. Push to state NP. Since the next word, "printed," is not a determiner or a proper noun, NP will pop and return failure.
14. The only other thing to do in state Q4 is to halt. But more input remains, so a complete parse has not been found. Backtracking is now required.

```

(S/ (PUSH NP/ T
    (SETR SUBJ *)
    (SETR TYPE (QUOTE DCL))
    (TO Q1))
  (CAT AUX T
    (SETR AUX *)
    (SETR TYPE (QUOTE Q))
    (TO Q2)))
(Q1 (CAT V T
    (SETR AUX NIL)
    (SETR V *)
    (TO Q4))
  (CAT AUX T
    (SETR AUX *)
    (TO Q3)))
(Q2 (PUSH NP/ T
    (SETR SUBJ *)
    (TO Q3)))
(Q3 (CAT V T
    (SETR V *)
    (TO Q4)))
(Q4 (POP (BUILDQ (S +++ (VP +))
                TYPE SUBJ AUX V) T)
    (PUSH NP/ T
    (SETR VP (BUILDQ (VP (V +) *) V))
    (TO Q5)))
(Q5 (POP (BUILDQ (S +++ +)
                TYPE SUBJ AUX VP) T)
    (PUSH PP/ T
    (SETR VP (APPEND (GETR VP) (LIST *)))
    (TO Q5)))

```

Figure 15.9: An ATN Grammar in List Form

15. The last choice point was at state Q1, so return there. The registers AUX and V must be unset.
16. Do a category test to see if "has" is an auxiliary. This test succeeds, so set the AUX register to "has" and go to state Q3.
17. Do a category test to see if "printed" is a verb. This test succeeds, so set the V register to "printed." Go to state Q4.
18. Now, since the input is exhausted, Q4 is an acceptable final state. Pop and return the structure


```
(S DCL (NP (FILE (LONG) DEFINITE))
      HAS
      (VP PRINTED))
```

This structure is the output of the parse.

This example grammar illustrates several interesting points about the use of ATNs. A single subnetwork need only occur once even though it is used in more than one place. A network can be called recursively. Any number of internal registers may be used to contain the result of the parse. The result of a network can be built, using the function BUILDQ, out of values contained in the various system registers. A single state may be both a final state, in which a complete sentence has been found, and an intermediate state, in which only a part of a sentence has been recognized. And, finally, the contents of a register can be modified at any time.

In addition, there are a variety of ways in which ATNs can be used which are not shown in this example:

- The contents of registers can be swapped. For example, if the network were expanded to recognize passive sentences, then at the point that the passive was detected, the current contents of the SUBJ register would be transferred to an OBJ register and the object of the preposition "by" would be placed in the SUBJ register. Thus the final interpretation of the following two sentences would be the same.
 - Bill printed the file.
 - The file was printed by Bill.
- Arbitrary tests can be placed on the arcs. In each of the arcs in this example, a test is specified simply as T (always true). But this need not be the case. Suppose that when the first NP is found, its number is determined and recorded in a register called NUMBER. Then the arcs labeled V could have an additional test placed on them that checked that the number of the particular verb that was found is equal to the value stored in NUMBER. More sophisticated tests, involving semantic markers or other semantic features, can also be performed.

15.2.3 Unification Grammars

ATN grammars have substantial procedural components. The grammar describes the order in which constituents must be built. Variables are explicitly given values, and

they must already have been assigned a value before they can be referenced. This procedurality limits the effectiveness of ATN grammars in some cases, for example: in speech processing where some later parts of the sentence may have been recognized clearly while earlier parts are still unknown (for example, suppose we had heard, "The long * * * file printed."), or in systems that want to use the same grammar to support both understanding and generation (e.g., Appelt [1987], Shieber [1988], and Barnett *et al.* [1990]). Although there is no clear distinction between declarative and procedural representations (as we saw in Section 6.1), there is a spectrum and it often turns out that more declarative representations are more flexible than more procedural ones are. So in this section we describe a declarative approach to representing grammars.

When a parser applies grammar rules to a sentence, it performs two major kinds of operations:

- Matching (of sentence constituents to grammar rules)
- Building structure (corresponding to the result of combining constituents)

Now think back to the unification operation that we described in Section 5.4.4 as part of our theorem-proving discussion. Matching and structure building are operations that unification performs naturally. So an obvious candidate for representing grammars is some structure on which we can define a unification operator. Directed acyclic graphs (DAGs) can do exactly that.

Each DAG represents a set of attribute-value pairs. For example, the graphs corresponding to the words "the" and "file" are:

[CAT: DET
LEX: the]

[CAT: N
LEX: file
NUMBER: SING]

Both words have a lexical category (CAT) and a lexical entry. In addition, the word "file" has a value (SING) for the NUMBER attribute. The result of combining these two words to form a simple NP can also be described as a graph:

[NP: [DET: the
HEAD: file
NUMBER: SING]]

The rule that forms this new constituent can also be represented as a graph, but to do so we need to introduce a new notation. Until now, all our graphs have actually been trees. To describe graphs that are not trees, we need a way to label a piece of a graph and then point to that piece elsewhere in the graph. So let $\{n\}$ for any value of n be a label, which is to be interpreted as a label for the next constituent following it in the graph. Sometimes, the constituent is empty (i.e., there is not yet any structure that is known to fill that piece of the graph). In that case, the label functions very much like a variable and will be treated like one by the unification operation. It is this degenerate kind of a label that we need in order to describe the NP rule:

NP \rightarrow DET N

We can write this rule as the following graph:


```

(CONSTITUENT1: [CAT: DET
                LEX: {1}]
  CONSTITUENT2: [CAT: N
                 LEX: {2}
                 NUMBER: {3}]
  BUILD: [NP: [DET: {1}
              HEAD: {2}
              NUMBER: {3}]]]

```

This rule should be read as follows: Two constituents, described in the subgraphs labeled CONSTITUENT1 and CONSTITUENT2, are to be combined. The first must be of CAT DET. We do not care what its lexical entry is, but whatever it is will be bound to the label {1}. The second constituent must be of CAT N. Its lexical entry will be bound to the label {2}, and its number will be bound to the label {3}. The result of combining these two constituents is described in the subgraph labeled BUILD. This result will be a graph corresponding to an NP with three attributes: DET, HEAD, and NUMBER. The values for all these attributes are to be taken from the appropriate pieces of the graphs that are being combined by the rule.

Now we need to define a unification operator that can be applied to the graphs we have just described. It will be very similar to logical unification. Two graphs unify if, recursively, all their subgraphs unify. The result of a successful unification is a graph that is composed of the union of the subgraphs of the two inputs, with all bindings made as indicated. This process bottoms out when a subgraph is not an attribute-value pair but is just a value for an attribute. At that point, we must define what it means for two values to unify. Identical values unify. Anything unifies with a variable (a label with no attached structure) and produces a binding for the label. The simplest thing to do is then to say that any other situation results in failure. But it may be useful to be more flexible. So some systems allow a value to match with a more general one (e.g., PROPER-NOUN matches NOUN). Others allow values that are disjunctions [e.g., (MASCULINE \vee FEMININE)], in which case unification succeeds whenever the intersection of the two values is not empty.

There is one other important difference between logical unification and graph unification. The inputs to logical unification are treated as logical formulas. Order matters, since, for example, $f(g(a), h(b))$ is a different formula than $f(h(b), g(a))$. The inputs to graph unification, on the other hand, must be treated as sets, since the order in which attribute-value pairs are stated does not matter. For example, if a rule describes a constituent as

```

[CAT: DET
 LEX: {1}]

```

we want to be able to match a constituent such as

```

[LEX: the
 CAT: DET]

```

Algorithm: Graph-Unify

1. If either $G1$ or $G2$ is an attribute that is not itself an attribute-value pair then:

- (a) If the attributes conflict (as defined above), then fail.
 - (b) If either is a variable, then bind it to the value of the other and return that value.
 - (c) Otherwise, return the most general value that is consistent with both the original values. Specifically, if disjunction is allowed, then return the intersection of the values.
2. Otherwise, do:
- (a) Set variable *NEW* to empty.
 - (b) For each attribute *A* that is present (at the top level) in either *G1* or *G2* do
 - i. If *A* is not present at the top level in the other input, then add *A* and its value to *NEW*.
 - ii. If it is, then call Graph-Unify with the two values for *A*. If that fails, then fail. Otherwise, take the new value of *A* to be the result of that unification and add *A* with its value to *NEW*.
 - (c) If there are any labels attached to *G1* or *G2*, then bind them to *NEW* and return *NEW*.

A simple parser can use this algorithm to apply a grammar rule by unifying CONSTITUENT1 with a proposed first constituent. If that succeeds, then CONSTITUENT2 is unified with a proposed second constituent. If that also succeeds, then a new constituent corresponding to the value of BUILD is produced. If there are variables in the value of BUILD that were bound during the matching of the constituents, then those bindings will be used to build the new constituent.

There are many possible variations on the notation we have described here. There are also a variety of ways of using it to represent dictionary entries and grammar rules. See Shieber [1986] and Knight [1989] for discussions of some of them.

Although we have presented unification here as a technique for doing syntactic analysis, it has also been used as a basis for semantic interpretation. In fact, there are arguments for using it as a uniform representation for all phases of natural language understanding. There are also arguments against doing so, primarily involving system modularity, the noncompositionality of language in some respects (see Section 15.3.4), and the need to invoke substantial domain reasoning. We will not say any more about this here but to see how this idea could work, see Allen [1989].

15.3 Semantic Analysis

Producing a syntactic parse of a sentence is only the first step toward understanding it. We must still produce a representation of the *meaning* of the sentence. Because understanding is a mapping process, we must first define the language into which we are trying to map. There is no single, definitive language in which all sentence meanings can be described. All of the knowledge representation systems that were described in Part II are candidates, and having selected one or more of them, we still need to define the vocabulary (i.e., the predicates, frames, or whatever) that will be used on top of the

structure. In the rest of this chapter, we call the final meaning representation language, including both the representational framework and the specific meaning vocabulary, the *target language*. The choice of a target language for any particular natural language understanding program must depend on what is to be done with the meanings once they are constructed. There are two broad families of target languages that are used in NL systems, depending on the role that the natural language system is playing in a larger system (if any).

When natural language is being considered as a phenomenon on its own, as, for example, when one builds a program whose goal is to read text and then answer questions about it, a target language can be designed specifically to support language processing. In this case, one typically looks for primitives that correspond to distinctions that are usually made in language. Of course, selecting the right set of primitives is not easy. We discussed this issue briefly in Section 4.3.3, and in Chapter 10 we looked at two proposals for a set of primitives, conceptual dependency and CYC.

When natural language is being used as an interface language to another program (such as a database query system or an expert system), then the target language must be a legal input to that other program. Thus the design of the target language is driven by the backend program. This was the case in the simple example we discussed in Section 15.1.1. But even in this case, it is useful, as we showed in that example, to use an intermediate knowledge-based representation to guide the overall process. So, in the rest of this section, we assume that the target language we are building is a knowledge-based one.

Although the main purpose of semantic processing is the creation of a target language representation of a sentence's meaning, there is another important role that it plays. It imposes constraints on the representations that can be constructed, and, because of the structural connections that must exist between the syntactic structure and the semantic one, it also provides a way of selecting among competing syntactic analyses. Semantic processing can impose constraints because it has access to knowledge about what makes sense in the world. We already mentioned one example of this, the sentence, "Is the glass jar peanut butter?" There are other examples in the rest of this section.

Lexical Processing

The first step in any semantic processing system is to look up the individual words in a dictionary (or *lexicon*) and extract their meanings. Unfortunately, many words have several meanings, and it may not be possible to choose the correct one just by looking at the word itself. For example, the word "diamond" might have the following set of meanings:

- A geometrical shape with four equal sides
- A baseball field
- An extremely hard and valuable gemstone

To select the correct meaning for the word "diamond" in the sentence,

Joan saw Susan's diamond shimmering from across the room.

it is necessary to know that neither geometrical shapes nor baseball fields shimmer, whereas gemstones do.

Unfortunately, if we view English understanding as mapping from English words into objects in a specific knowledge base, lexical ambiguity is often greater than it seems in everyday English. For, example, consider the word "mean." This word is ambiguous in at least three ways: it can be a verb meaning "to signify"; it can be an adjective meaning "unpleasant" or "cheap"; and it can be a noun meaning "statistical average." But now imagine that we have a knowledge base that describes a statistics program and its operation. There might be at least two distinct objects in that knowledge base, both of which correspond to the "statistical average" meaning of "mean." One object is the statistical concept of a mean; the other is the particular function that computes the mean in this program. To understand the word "mean" we need to map it into some concept in our knowledge base. But to do that, we must decide which of these concepts is meant. Because of cases like this, lexical ambiguity is a serious problem, even when the domain of discourse is severely constrained.

The process of determining the correct meaning of an individual word is called *word sense disambiguation* or *lexical disambiguation*. It is done by associating, with each word in the lexicon, information about the contexts in which each of the word's senses may appear. Each of the words in a sentence can serve as part of the context in which the meanings of the other words must be determined.

Sometimes only very straightforward information about each word sense is necessary. For example, the baseball field interpretation of "diamond" could be marked as a LOCATION. Then the correct meaning of "diamond" in the sentence "I'll meet you at the diamond" could easily be determined if the fact that *at* requires a TIME or a LOCATION as its object were recorded as part of the lexical entry for *at*. Such simple properties of word senses are called *semantic markers*. Other useful semantic markers are

- PHYSICAL-OBJECT
- ANIMATE-OBJECT
- ABSTRACT-OBJECT

Using these markers, the correct meaning of "diamond" in the sentence "I dropped my diamond" can be computed. As part of its lexical entry, the verb "drop" will specify that its object must be a PHYSICAL-OBJECT. The gemstone meaning of "diamond" will be marked as a PHYSICAL-OBJECT. So it will be selected as the appropriate meaning in this context.

This technique has been extended by Wilks [1972; 1975a; 1975b] in his *preference semantics*, which relies on the notion that requirements, such as the one described above for an object that is a LOCATION, are rarely hard-and-fast demands. Rather, they can best be described as preferences. For example, we might say that verbs such as "hate" prefer a subject that is animate. Thus we have no difficulty in understanding the sentence

Pop hates the cold.

as describing the feelings of a man and not those of soft drinks. But now consider the sentence

My lawn hates the cold.

Now there is no animate subject available, and so the metaphorical use of lawn acting as an animate object should be accepted.

Unfortunately, to solve the lexical disambiguation problem completely, it becomes necessary to introduce more and more finely grained semantic markers. For example, to interpret the sentence about Susan's diamond correctly, we must mark one sense of diamond as SHIMMERABLE, while the other two are marked NONSHIMMERABLE. As the number of such markers grows, the size of the lexicon becomes unmanageable. In addition, each new entry into the lexicon may require that a new marker be added to each of the existing entries. The breakdown of the semantic marker approach when the number of words and word senses becomes large has led to the development of other ways in which correct senses can be chosen. We return to this issue in Section 15.3.4.

Sentence-Level Processing

Several approaches to the problem of creating a semantic representation of a sentence have been developed, including the following:

- Semantic grammars, which combine syntactic, semantic, and pragmatic knowledge into a single set of rules in the form of a grammar. The result of parsing with such a grammar is a semantic, rather than just a syntactic, description of a sentence.
- Case grammars, in which the structure that is built by the parser contains some semantic information, although further interpretation may also be necessary.
- Conceptual parsing, in which syntactic and semantic knowledge are combined into a single interpretation system that is driven by the semantic knowledge. In this approach, syntactic parsing is subordinated to semantic interpretation, which is usually used to set up strong expectations for particular sentence structures.
- Approximately compositional semantic interpretation, in which semantic processing is applied to the result of performing a syntactic parse. This can be done either incrementally, as constituents are built, or all at once, when a structure corresponding to a complete sentence has been built.

In the following sections, we discuss each of these approaches.

15.3.1 Semantic Grammars

A *semantic grammar* [Burton, 1976; Hendrix *et al.*, 1978; Hendrix and Lewis, 1981] is a context-free grammar in which the choice of nonterminals and production rules is governed by semantic as well as syntactic function. In addition, there is usually a semantic action associated with each grammar rule. The result of parsing and applying all the associated semantic actions is the meaning of the sentence. This close coupling of semantic actions to grammar rules works because the grammar rules themselves are designed around key semantic concepts.

$S \rightarrow$ what is FILE-PROPERTY of FILE?
 {query FILE.FILE-PROPERTY}
 $S' \rightarrow$ I want to ACTION
 {command ACTION}
 FILE-PROPERTY \rightarrow the FILE-PROP
 {FILE-PROP}
 FILE-PROP \rightarrow extension | protection | creation date | owner
 {value}
 FILE \rightarrow FILE-NAME | FILE1
 {value}
 FILE1 \rightarrow USER's FILE2
 {FILE2.owner: USER}
 FILE1 \rightarrow FILE2
 {FILE2}
 FILE2 \rightarrow EXT file
 {instance: file-struct
 extension: EXT}
 EXT \rightarrow .init | .txt | .lsp | .for | .ps | .mss
 value
 ACTION \rightarrow print FILE
 {instance: printing
 object: FILE}
 ACTION \rightarrow print FILE on PRINTER
 {instance: printing
 object: FILE
 printer: PRINTER}
 USER \rightarrow Bill | Susan
 {value}

Figure 15.10: A Semantic Grammar

An example of a fragment of a semantic grammar is shown in Figure 15.10. This grammar defines part of a simple interface to an operating system. Shown in braces under each rule is the semantic action that is taken when the rule is applied. The term "value" is used to refer to the value that is matched by the right-hand side of the rule. The dotted notation $x.y$ should be read as the y attribute of the unit x . The result of a successful parse using this grammar will be either a command or a query.

A semantic grammar can be used by a parsing system in exactly the same ways in which a strictly syntactic grammar could be used. Several existing systems that have used semantic grammars have been built around an ATN parsing system, since it offers a great deal of flexibility.

Figure 15.11 shows the result of applying this semantic grammar to the sentence

I want to print Bill's .init file

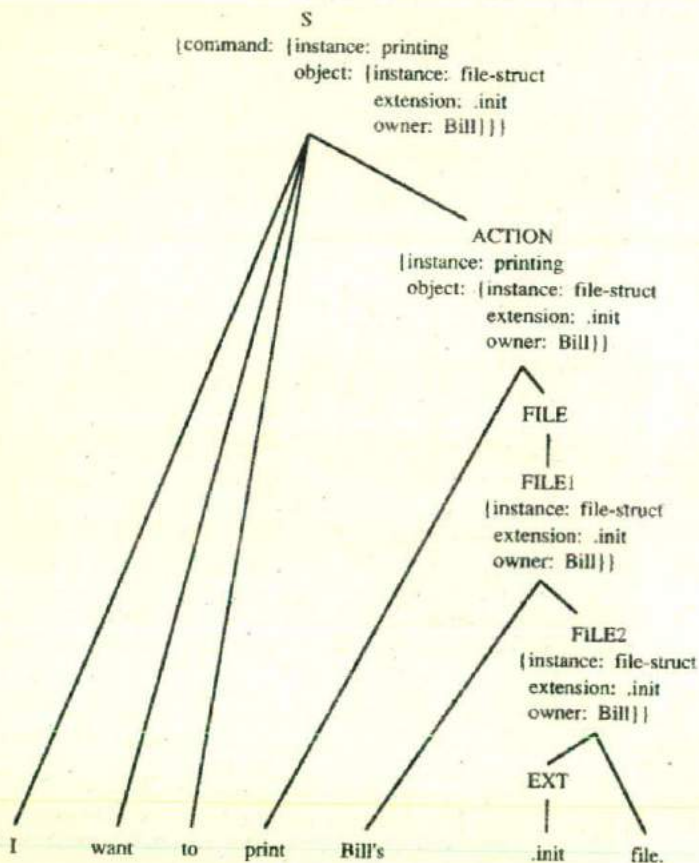


Figure 15.11: The Result of Parsing with a Semantic Grammar

Notice that in this approach, we have combined into a single process all five steps of Section 15.1.1 with the exception of the final part of pragmatic processing in which the conversion to the system's command syntax is done.

The principal advantages of semantic grammars are the following:

- When the parse is complete, the result can be used immediately without the additional stage of processing that would be required if a semantic interpretation had not already been performed during the parse.
- Many ambiguities that would arise during a strictly syntactic parse can be avoided since some of the interpretations do not make sense semantically and thus cannot be generated by a semantic grammar. Consider, for example, the sentence "I want to print stuff.txt on printer3." During a strictly syntactic parse, it would not be possible to decide whether the prepositional phrase, "on printer3" modified "want" or "print." But using our semantic grammar, there is no general notion of a prepositional phrase and there is no attachment ambiguity.
- Syntactic issues that do not affect the semantics can be ignored. For example, using the grammar shown above, the sentence, "What is the extension of .lisp file?" would be parsed and accepted as correct.

There are, however, some drawbacks to the use of semantic grammars:

- The number of rules required can become very large since many syntactic generalizations are missed.
- Because the number of grammar rules may be very large, the parsing process may be expensive.

After many experiments with the use of semantic grammars in a variety of domains, the conclusion appears to be that for producing restricted natural language interfaces quickly, they can be very useful. But as an overall solution to the problem of language understanding, they are doomed by their failure to capture important linguistic generalizations.

15.3.2 Case Grammars

Case grammars [Fillmore, 1968; Bruce, 1975] provide a different approach to the problem of how syntactic and semantic interpretation can be combined. Grammar rules are written to describe syntactic rather than semantic regularities. But the structures the rules produce correspond to semantic relations rather than to strictly syntactic ones. As an example, consider the two sentences and the simplified forms of their conventional parse trees shown in Figure 15.12.

Although the semantic roles of "Susan" and "the file" are identical in these two sentences, their syntactic roles are reversed. Each is the subject in one sentence and the object in another.

Using a case grammar, the interpretations of the two sentences would both be

(printed (agent Susan)
(object File))

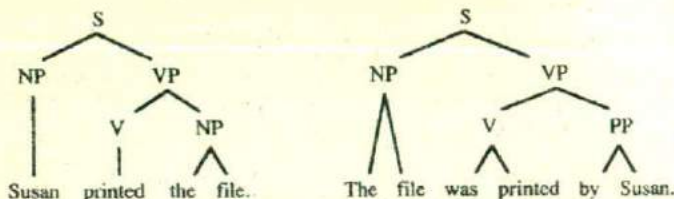


Figure 15.12: Syntactic Parses of an Active and a Passive Sentence

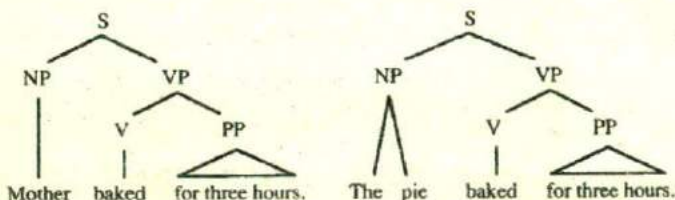


Figure 15.13: Syntactic Parses of Two Similar Sentences

Now consider the two sentences shown in Figure 15.13.

The syntactic structures of these two sentences are almost identical. In one case, "Mother" is the subject of "baked," while in the other "the pie" is the subject. But the relationship between Mother and baking is very different from that between the pie and baking. A case grammar analysis of these two sentences reflects this difference. The first sentence would be interpreted as

(baked (agent Mother)
(timeperiod 3-hours))

The second would be interpreted as

(baked (object Pie)
(timeperiod 3-hours))

In these representations, the semantic roles of "mother" and "the pie" are made explicit. It is interesting to note that this semantic information actually does intrude into the syntax of the language. While it is allowed to conjoin two parallel sentences (e.g., "the pie baked" and "the cake baked" become "the pie and the cake baked"), this is only possible if the conjoined noun phrases are in the same case relation to the verb. This accounts for the fact that we do not say, "Mother and the pie baked."

Notice that the cases used by a case grammar describe relationships between verbs and their arguments. This contrasts with the grammatical notion of surface case, as

exhibited, for example, in English, by the distinction between "I" (nominative case) and "me" (objective case). A given grammatical, or surface, case can indicate a variety of semantic, or deep, cases.

There is no clear agreement on exactly what the correct set of deep cases ought to be, but some obvious ones are the following:

- (A) Agent—Instigator of the action (typically animate)
- (I) Instrument—Cause of the event or object used in causing the event (typically inanimate)
- (D) Dative—Entity affected by the action (typically animate)
- (F) Factitive—Object or being resulting from the event
- (L) Locative—Place of the event
- (S) Source—Place from which something moves
- (G) Goal—Place to which something moves
- (B) Beneficiary—Being on whose behalf the event occurred (typically animate)
- (T) Time—Time at which the event occurred
- (O) Object—Entity that is acted upon or that changes, the most general case

The process of parsing into a case representation is heavily directed by the lexical entries associated with each verb. Figure 15.14 shows examples of a few such entries. Optional cases are indicated in parentheses.

Languages have rules for mapping from underlying case structures to surface syntactic forms. For example, in English, the "unmarked subject"³ is generally chosen by the following rule:

If A is present, it is the subject. Otherwise, if I is present, it is the subject.
Else the subject is O.

These rules can be applied in reverse by a parser to determine the underlying case structure from the superficial syntax.

Parsing using a case grammar is usually *expectation-driven*. Once the verb of the sentence has been located, it can be used to predict the noun phrases that will occur and to determine the relationship of those phrases to the rest of the sentence.

ATNs provide a good structure for case grammar parsing. Unlike traditional parsing algorithms in which the output structure always mirrors the structure of the grammar rules that created it, ATNs allow output structures of arbitrary form. For an example of their use, see Simmons [1973], which describes a system that uses an ATN parser to translate English sentences into a semantic net representing the case structures of sentences. These semantic nets can then be used to answer questions about the sentences.

³The unmarked subject is the one that is used by default; it signals no special focus or emphasis in the sentence.

open	[- _ O (I) (A)] The door opened. John opened the door. The wind opened the door. John opened the door with a chisel.
die	[- _ D] John died.
kill	[- _ D (I) A] Bill killed John. Bill killed John with a knife.
run	[- _ A] John ran.
want	[- _ A O] John wanted some ice cream. John wanted Mary to go to the store.

Figure 15.14: Some Verb Case Frames

The result of parsing in a case representation is usually not a complete semantic description of a sentence. For example, the constituents that fill the case slots may still be English words rather than true semantic descriptions stated in the target representation. To go the rest of the way toward building a meaning representation, we still require many of the steps that are described in Section 15.3.4.

15.3.3 Conceptual Parsing

Conceptual parsing, like semantic grammars, is a strategy for finding both the structure and the meaning of a sentence in one step. Conceptual parsing is driven by a dictionary that describes the meanings of words as conceptual dependency (CD) structures.

Parsing a sentence into a conceptual dependency representation is similar to the process of parsing using a case grammar. In both systems, the parsing process is heavily driven by a set of expectations that are set up on the basis of the sentence's main verb. But because the representation of a verb in CD is at a lower level than that of a verb in a case grammar (in which the representation is often identical to the English word that is used), CD usually provides a greater degree of predictive power. The first step in mapping a sentence into its CD representation involves a syntactic processor that extracts the main noun and verb. It also determines the syntactic category and aspectual class of the verb (i.e., stative, transitive, or intransitive). The conceptual processor then takes over. It makes use of a verb-ACT dictionary, which contains an entry for each environment in which a verb can appear. Figure 15.15 (taken from Schank [1973]) shows the dictionary entries associated with the verb "want." These three entries correspond to the three

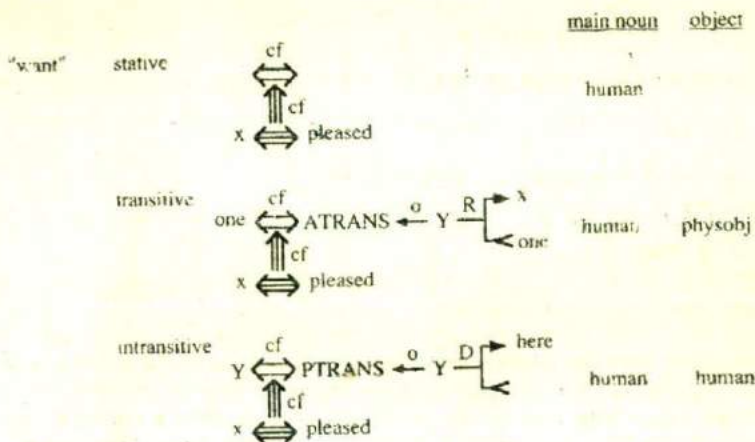


Figure 15.15: The Verb-ACT Dictionary

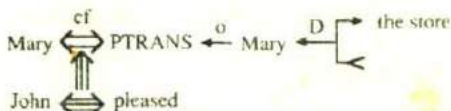


Figure 15.16: A CD Structure

kinds of wanting:

- Wanting something to happen
- Wanting an object
- Wanting a person

Once the correct dictionary entry is chosen, the conceptual processor analyzes the rest of the sentence looking for components that will fit into the empty slots of the verb structure. For example, if the stative form of "want" has been found, then the conceptual processor will look for a conceptualization that can be inserted into the structure. So, if the sentence being processed were

John wanted Mary to go to the store.

the structure shown in Figure 15.16 would be built.

The conceptual processor examines possible interpretations in a well-defined order. For example, if a phrase of the form "with PP" (recall that a PP is a picture producer)

occurs, it could indicate any of the following relationships between the PP and the conceptualization of which it is a part:

1. Object of the instrumental case
2. Additional actor of the main ACT
3. Attribute of the PP just preceding it
4. Attribute of the actor of the conceptualization

Suppose that the conceptual processor were attempting to interpret the prepositional phrase in the sentence

John went to the park with the girl.

First, the system's immediate memory would be checked to see if a park with a girl has been mentioned. If so, a reference to that particular object is generated and the process terminates. Otherwise, the four possibilities outlined above are investigated in the order in which they are presented. Can "the girl" be an instrument of the main ACT (PTRANS) of this sentence? The answer is no, because only MOVE and PROPEL can be instruments of a PTRANS and their objects must be either body parts or vehicles. "Girl" is neither of these. So we move on to consider the second possibility. In order for "girl" to be an additional actor of the main ACT, it must be animate. It is. So this interpretation is chosen and the process terminates. If, however, the sentence had been

John went to the park with the fountain.

the process would not have stopped since a fountain is inanimate and cannot move. Then the third possibility would have been considered. Since parks can have fountains, it would be accepted and the process would terminate there. For a more detailed description of the way a conceptual processor based on CD works, see Schank [1973], Rieger [1975], and Riesbeck [1975].

This example illustrates both the strengths and the weaknesses of this approach to sentence understanding. Because a great deal of semantic information is exploited in the understanding process, sentences that would be ambiguous to a purely syntactic parser can be assigned a unique interpretation. Unfortunately, the amount of semantic information that is required to do this job perfectly is immense. All simple rules have exceptions. For example, suppose the conceptual processor described above were given the sentence

John went to the park with the peacocks.

Since peacocks are animate, they would be acceptable as additional actors of the main verb, "went." Thus, the interpretation that would be produced would be that shown in Figure 15.17(a), while the more likely interpretation, in which John went to a park containing peacocks, is shown in Figure 15.17(b). But if the possible roles for a prepositional phrase introduced by "with" were considered in the order necessary for

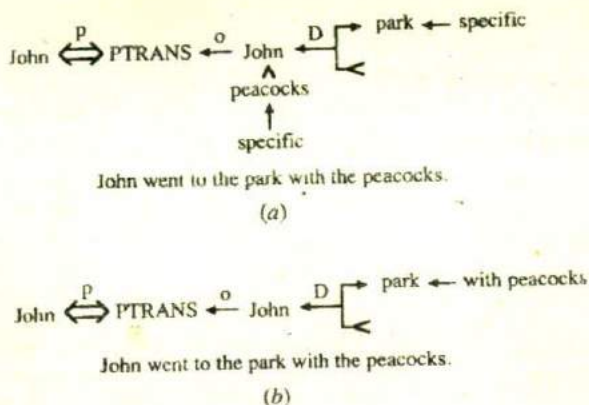


Figure 15.17: Two CD Interpretations of a Sentence

this sentence to be interpreted correctly, then the previous example involving the phrase, "with Mary," would have been misunderstood.

The problem is that the simple check for the property ANIMATE is not sufficient to determine acceptability as an additional actor of a PTRANS. Additional knowledge is necessary. Some more knowledge can be inserted within the framework we have described for a conceptual processor. But to do a very good job of producing correct semantic interpretations of sentences requires knowledge of the larger context in which the sentence appears. Techniques for exploiting such knowledge are discussed in the next section.

15.3.4 Approximately Compositional Semantic Interpretation

The final approach to semantics that we consider here is one in which syntactic parsing and semantic interpretation are treated as separate steps, although they must mirror each other in well-defined ways. This is the approach to semantics that we looked at briefly in Section 15.1.1 when we worked through the example sentence "I want to print Bill's .init file."

If a strictly syntactic parse of a sentence has been produced, then a straightforward way to generate a semantic interpretation is the following:

1. Look up each word in a lexicon that contains one or more definitions for the word, each stated in terms of the chosen target representation. These definitions must describe how the idea that corresponds to the word is to be represented, and they may also describe how the idea represented by this word may combine with the ideas represented by other words in the sentence.
2. Use the structure information contained in the output of the parser to provide additional constraints, beyond those extracted from the lexicon, on the way individual

words may combine to form larger meaning units.

We have already discussed the first of these steps (in Section 15.3). In the rest of this section, we discuss the second.

Montague Semantics

Recall that we argued in Section 15.1.1 that the reason syntactic parsing was a good idea was that it produces structures that correspond to the structures that should result from semantic processing. If we investigate this idea more closely, we arrive at a notion called *compositional semantics*. The main idea behind compositional semantics is that, for every step in the syntactic parsing process, there is a corresponding step in semantic interpretation. Each time syntactic constituents are combined to form a larger syntactic unit, their corresponding semantic interpretations can be combined to form a larger semantic unit. The necessary rules for combining semantic structures are associated with the corresponding rules for combining syntactic structures. We use the word "compositional" to describe this approach because it defines the meaning of each sentence constituent to be a composition of the meanings of its constituents with the meaning of the rule that was used to create it. The main theoretical basis for this approach is modern (i.e., post-Fregean) logic; the clearest linguistic application is the work of Montague [Dowty *et al.*, 1981; Thomason, 1974].

As an example of this approach to semantic interpretation, let's return to the example that we began in Section 15.1.1. The sentence is

I want to print Bill's .init file.

The output of the syntactic parsing process was shown in Figure 15.2, and a fragment of the knowledge base that is being used to define the target representation was shown in Figure 15.3. The result of semantic interpretation was also shown there in Figure 15.4. Although the exact form of semantic mapping rules in this approach depends on the way that the syntactic grammar is defined, we illustrate the idea of compositional semantic rules in Figure 15.18.

The first two rules are examples of verb-mapping rules. Read these rules as saying that they map from a partial syntactic structure containing a verb, its subject, and its object, to some unit with the attributes instance, agent, and object. These rules do two things. They describe the meaning of the verbs ("want" or "print") themselves in terms of events in the knowledge base. They also state how the syntactic arguments of the verbs (their subjects and objects) map into attributes of those events. By the way, do not get confused by the use of the term "object" in two different senses here. The syntactic object of a sentence and its semantic object are two different things. For historical reasons (including the standard usage in case grammars as described in Section 15.3.2), they are often called the same thing, although this problem is sometimes avoided by using some other name, such as *affected-entity*, for the semantic object. Alternatively, in some knowledge bases, much more specialized names, such as *printed-thing*, are sometimes used as attribute names.

The third and fourth rules are examples of modifier rules. Like the verb rules, they too must specify both their own constituent's contribution to meaning as well as how it combines with the meaning of the noun phrase or phrases to which it is attached.

"want"	→	Unit
subject: RM_i		instance: <i>Wanting</i>
object: RM_j		agent: RM_i
		object: RM_j
"print"	→	Unit
subject: RM_i		instance: <i>Printing</i>
object: RM_j		agent: RM_i
		object: RM_j
".init"	→	Unit for NP_1 plus
modifying NP_1		extension: <i>.init</i>
possessive marker	→	Unit for NP_2 plus
NP_1 's NP_2		owner: NP_1
"file"	→	Unit
		instance: <i>File-Struct</i>
"Bill"	→	Unit
		instance: <i>Person</i>
		first-name: <i>Bill</i>

Figure 15.18: Some Semantic Interpretation Rules

The last two rules are simpler. They define the meanings of nouns. Since nouns do not usually take arguments, these rules specify only single-word meanings; they do not need to describe how the meanings of larger constituents are derived from their components.

One important thing to remember about these rules is that since they define mappings from words into a knowledge base, they implicitly make available to the semantic processing system all the information contained in the knowledge base itself. For example, Figure 15.19 contains a description of the semantic information that is associated with the word "want" after applying the semantic rule associated with wanting events and retrieving semantic constraints associated with wanting events in the knowledge base. Notice that we now know where to pick up the agent for the wanting ($RM1$) and we know some property that the agent must have. The semantic interpretation routine will reject any interpretation that does not satisfy all these constraints.

This compositional approach to defining semantic interpretation has proved to be a very powerful idea. (See, for example, the Absity system described in Hirst [1987].) Unfortunately, there are some linguistic constructions that cannot be accounted for naturally in a strictly compositional system. Quantified expressions have this property. Consider, for example, the sentence

Every student who hadn't declared a major took an English class.

<i>Unit</i>	
<i>instance :</i>	<i>Wanting</i>
<i>agent :</i>	<i>RM_i</i> must be <animate>
<i>object :</i>	<i>RM_j</i> must be <state or event>

Figure 15.19: Combining Mapping Knowledge with the Knowledge Base

There are several ways in which the relative scopes of the quantifiers in this sentence can be assigned. In the most likely, both existential quantifiers are within the scope of the universal quantifier. But, in other readings, they are not. These include readings corresponding to, "There is a major such that every student who had not declared it took an English class," and "There is an English class such that every student who had not declared some major took it." In order to generate these meanings compositionally from the parse, it is necessary to produce a separate parse for each scope assignment. But there is no syntactic reason to do that, and it requires substantial additional effort. An alternative is to generate a single parse and then to use a noncompositional algorithm to generate as many alternative scopes as desired.

As a second example, consider the sentence, "John only eats meat on Friday and Mary does too." The syntactic analysis of this sentence must include the verb phrase constituent, "only eats meat on Friday," since that is the constituent that is picked up by the elliptical expression "does too." But the meaning of the first clause has a structure more like

only(meat, { x | John eats x on Friday})

which can be read as, "Meat is the only thing that John eats on Friday."

Extended Reasoning with a Knowledge Base

A significant amount of world knowledge may be necessary in order to do semantic interpretation (and thus, sometimes, to get the correct syntactic parse). Sometimes the knowledge is needed to enable the system to choose among competing interpretations. Consider, for example, the sentences

1. John made a huge wedding cake with chocolate icing.
2. John made a huge wedding cake with Bill's mixer.
3. John made a huge wedding cake with a giant tower covered with roses.
4. John made a cherry pie with a giant tower covered with roses.

Let's concentrate on the problem of deciding to which constituent the prepositional phrase should be attached and of assigning a meaning to the preposition "with." We

have two main choices: either the phrase attaches to the action of making the cake and "with" indicates the instrument relation, or the prepositional phrase attaches to the noun phrase describing the dessert that was made, in which case "with" describes an additional component of the dessert. The first two sentences are relatively straightforward if we imagine that our knowledge base contains the following facts:

- Foods can be components of other foods.
- Mixers are used to make many kinds of desserts.

But now consider the third sentence. A giant tower is neither a food nor a mixer. So it is not a likely candidate for either role. What is required here is the much more specific (and culturally dependent) fact that

- Wedding cakes often have towers and statues and bridges and flowers on them.

The highly specific nature of this knowledge is illustrated by the fact that the last of these sentences does not make much sense to us since we can find no appropriate role for the tower, either as part of a pie or as an instrument used during pie making.

Another use for knowledge is to enable the system to accept meanings that it has not been explicitly told about. Consider the following sentences as examples:

1. Sue likes to read Joyce.
2. Washington backed out of the summit talks.
3. The stranded explorer ate squirrels.

Suppose our system has only the following meanings for the words "Joyce," "Washington," and "squirrel" (actually we give only the relevant parts of the meanings):

1. Joyce—*instance*: Author; *last-name*: Joyce
2. Washington—*instance*: City; *name*: Washington
3. squirrel—*isa*: Rodent; ...

But suppose that we also have only the following meanings for the verbs in these sentences:

1. read—*isa*: Mental-Event; *object*: must be <printed-material>
2. back out—*isa*: Mental-Event; *agent*: must be <animate-entity>
3. eat—*isa*: Ingestion-Event; *object*: must be <food>

The problem is that it is not possible to construct coherent interpretations for any of these sentences with these definitions. An author is not a <printed-material>. A city is not an <animate-entity>. A rodent is not a <food>. One solution is to create additional dictionary entries for the nouns: Joyce as a set of literary works, Washington as the people who run the U.S. government, and a squirrel as a food. But a better solution is to use general knowledge to derive these meanings when they are needed. By better, here we mean that since less knowledge must be entered by hand, the resulting system will be less brittle. The general knowledge that is necessary to handle these examples is:

- The name of a person can be used to refer to things the person creates. *Auditing* is a kind of creating.
- The name of a place can be used to stand for an organization headquartered in that place if the association between the organization and the place is salient in the context. An organization can in turn stand for the people who run it. The headquarters of the U.S. government is in Washington.
- Food (meat) can be made out of almost any animal. Usually the word for the animal can be used to refer to the meat made from the animal.

Of course, this problem can become arbitrarily complex. For example, metaphors are a rich source for linguistic expressions [Lakoff and Johnson, 1980]. And the problem becomes even more complex when we move beyond single sentences and attempt to extract meaning from texts and dialogues. We delve briefly into those issues in Section 15.4.

The Interaction between Syntax and Semantics

If we take a compositional approach to semantics, then we apply semantic interpretation rules to each syntactic constituent, eventually producing an interpretation for an entire sentence. But making a commitment about what to do implies no specific commitment about when to do it. To implement a system, however, we must make some decision on how control will be passed back and forth between the syntactic and the semantic processors. Two extreme positions are:

- Every time a syntactic constituent is formed, apply semantic interpretation to it immediately.
- Wait until the entire sentence has been parsed, and then interpret the whole thing.

There are arguments in favor of each approach. The theme of most of the arguments is search control and the opportunity to prune dead-end paths. Applying semantic processing to each constituent as soon as it is produced allows semantics to rule out right away those constituents that are syntactically valid but that make no sense. Syntactic processing can then be informed that it should not go any further with those constituents. This approach would pay off, for example, for the sentence, "Is the glass jar peanut butter?" But this approach can be costly when syntactic processing builds constituents that it will eventually reject as being syntactically unacceptable, regardless of their semantic acceptability. The sentence, "The horse raced past the barn fell down," is an example of this. There is no point in doing a semantic analysis of the sentence "The horse raced past the barn," since that constituent will not end up being part of any complete syntactic parse. There are also additional arguments for waiting until a complete sentence has been parsed to do at least some parts of semantic interpretation. These arguments involve the need for large constituents to serve as the basis of those semantic actions, such as the ones we discussed in Section 15.3.4, that are hard to define completely compositionally. There is no magic solution to this problem. Most systems use one of these two extremes or a heuristically driven compromise position.

15.4 Discourse and Pragmatic Processing

To understand even a single sentence, it is necessary to consider the discourse and pragmatic context in which the sentence was uttered (as we saw in Section 15.1.1). These issues become even more important when we want to understand texts and dialogues, so in this section we broaden our concern to these larger linguistic units. There are a number of important relationships that may hold between phrases and parts of their discourse contexts, including:

- Identical entities. Consider the text

- Bill had a red balloon.
- John wanted it.

The word "it" should be identified as referring to the red balloon. References such as this are called *anaphoric references* or *anaphora*.

- Parts of entities. Consider the text

- Sue opened the book she just bought.
- The title page was torn.

The phrase "the title page" should be recognized as being part of the book that was just bought.

- Parts of actions. Consider the text

- John went on a business trip to New York.
- He left on an early morning flight.

Taking a flight should be recognized as part of going on a trip.

- Entities involved in actions. Consider the text

- My house was broken into last week.
- They took the TV and the stereo.

The pronoun "they" should be recognized as referring to the burglars who broke into the house.

- Elements of sets. Consider the text

- The decals we have in stock are stars, the moon, item and a flag.
- I'll take two moons.

The moons in the second sentence should be understood to be some of the moons mentioned in the first sentence. Notice that to understand the second sentence a **ll** requires that we use the context of the first sentence to establish that the word "moons" means moon decals.

- Names of individuals. Consider the text

- Dave went to the movies.

Dave should be understood to be some person named Dave. Although there are many, the speaker had one particular one in mind and the discourse context should tell us which.

- Causal chains. Consider the text

- There was a big snow storm yesterday.

- The schools were closed today.

The snow should be recognized as the reason that the schools were closed.

- Planning sequences. Consider the text

- Sally wanted a new car.

- She decided to get a job.

Sally's sudden interest in a job should be recognized as arising out of her desire for a new car and thus for the money to buy one.

- Illocutionary force. Consider the sentence

- It sure is cold in here.

In many circumstances, this sentence should be recognized as having, as its intended effect, that the hearer should do something like close the window or turn up the thermostat.

- Implicit presuppositions. Consider the query

- Did Joe fail CS101?

The speaker's presuppositions, including the fact that CS101 is a valid course, that Joe is a student, and that Joe took CS101, should be recognized so that if any of them is not satisfied, the speaker can be informed.

In order to be able to recognize these kinds of relationships among sentences, a great deal of knowledge about the world being discussed is required. Programs that can do multiple-sentence understanding rely either on large knowledge bases or on strong constraints on the domain of discourse so that only a more limited knowledge base is necessary. The way this knowledge is organized is critical to the success of the understanding program. In the rest of this section, we discuss briefly how some of the knowledge representations described in Chapters 9 and 10 can be exploited by a language-understanding program. In particular, we focus on the use of the following kinds of knowledge:

- The current focus of the dialogue

- A model of each participant's current beliefs
- The goal-driven character of dialogue
- The rules of conversation shared by all participants

Although these issues are complex, we discuss them only briefly here. Most of the hard problems are not peculiar to natural language processing. They involve reasoning about objects, events, goals, plans, intentions, beliefs, and likelihoods, and we have discussed all these issues in some detail elsewhere. Our goal in this section is to tie those reasoning mechanisms into the process of natural language understanding.

15.4.1 Using Focus in Understanding

There are two important parts of the process of using knowledge to facilitate understanding:

- Focus on the relevant part(s) of the available knowledge base.
- Use that knowledge to resolve ambiguities and to make connections among things that were said.

The first of these is critical if the amount of knowledge available is large. Some techniques for handling this were outlined in Section 4.3.5, since the problem arises whenever knowledge structures are to be used.

The linguistic properties of coherent discourse, however, provide some additional mechanisms for focusing. For example, the structure of task-oriented discourses typically mirrors the structure of the task. Consider the following sequence of (highly simplified) instructions:

To make the torte, first make the cake, then, while the cake is baking, make the filling. To make the cake, combine all ingredients. Pour them into the pans, and bake for 30 minutes. To make the filling, combine the ingredients. Mix until light and fluffy. When the cake is done, alternate layers of cake and filling.

This task decomposes into three subtasks: making the cake, making the filling, and combining the two components. The structure of the paragraph of instructions is: overall sketch of the task, instructions for step 1, instructions for step 2, and then instructions for step 3.

A second property of coherent discourse is that dramatic changes of focus are usually signaled explicitly with phrases such as "on the other hand," "to return to an earlier topic," or "a second issue is."

Assuming that all this knowledge has been used successfully to focus on the relevant part(s) of the knowledge base, the second issue is how to use the focused knowledge to help in understanding. There are as many ways of doing this as there are discourse phenomena that require it. In the last section, we presented a sample list of those phenomena. To give one example, consider the problem of finding the meaning of definite noun phrases. Definite noun phrases are ones that refer to specific individual

objects, for example, the first noun phrase in the sentence, "The title page was torn." The title page in question is assumed to be one that is related to an object that is currently in focus. So the procedure for finding a meaning for it involves searching for ways in which a title page could be related to a focused object. Of course, in some sense, almost any object in a knowledge base relates somehow to almost any other. But some relations are far more salient than others, and they should be considered first. Highly salient relations include *physical-part-of*, *temporal-part-of*, and *element-of*. In this example, *physical-part-of* relates the title page to the book that is in focus as a result of its mention in the previous sentence.

Other ways of using focused information also exist. We examine some of them in the remaining parts of this section.

15.4.2 Modeling Beliefs

In order for a program to be able to participate intelligently in a dialogue, it must be able to represent not only its own beliefs about the world, but also its knowledge of the other dialogue participant's beliefs about the world, that person's beliefs about the computer's beliefs, and so forth. The remark "She knew I knew she knew I knew she knew"⁴ may be a bit extreme, but we do that kind of thinking all the time. To make computational models of belief, it is useful to divide the issue into two parts: those beliefs that can be assumed to be shared among all the participants in a linguistic event and those that cannot.

Modeling Shared Beliefs

Shared beliefs can be modeled without any explicit notion of belief in the knowledge base. All we need to do is represent the shared beliefs as facts, and they will be accessed whenever knowledge about anyone's beliefs is needed. We have already discussed techniques for doing this. For example, much of the knowledge described in Chapter 10 is exactly the sort that people presume is shared by other people they are communicating with. Scripts, in particular, have been used extensively to aid in natural language understanding. Recall that scripts record commonly occurring sequences of events. There are two steps in the process of using a script to aid in language understanding:

- Select the appropriate script(s) from memory.
- Use the script(s) to fill in unspecified parts of the text to be understood.

Both of these aspects of reasoning with scripts have already been discussed in Section 10.2. The story-understanding program SAM [Cullingford, 1981] demonstrated the usefulness of such reasoning with scripts in natural language understanding. To understand a story, SAM first employed a parser that translated the English sentences into their conceptual dependency representation. Then it built a representation of the entire text using the relationships indicated by the relevant scripts.

⁴From Kingsley Amis' *Jake's Thing*.

Modeling Individual Beliefs

As soon as we decide to represent individual beliefs, we need to introduce some explicit predicate(s) to indicate that a fact is believed. Up until now, belief has been indicated only by the presence or absence of assertions in the knowledge base. To model belief, we need to move to a logic that supports reasoning about belief propositions. The standard approach is to use a *modal logic* such as that defined in Hintikka [1962]. Logic, or "classical" logic, deals with the truth or falsehood of different statements as they are. Modal logic, on the other hand, concerns itself with the different "modes" in which a statement may be true. Modal logics allow us to talk about the truth of a set of propositions not only in the current state of the real world, but also about their truth or falsehood in the past or the future (these are called *temporal logics*), and about their truth or falsehood under circumstances that might have been, but were not (these are sometimes called *conditional logics*). We have already used one idea from modal logic, namely the notion *necessarily true*. We used it in Section 13.5, when we talked about nonlinear planning in TWEAK.

Modal logics also allow us to talk of the truth or falsehood of statements concerning the beliefs, knowledge, desires, intentions, and obligations of people and robots, which may, in fact be, respectively, false, unjustified, unsatisfiable, irrational, or mutually contradictory. Modal logics thus provide a set of powerful tools for understanding natural language utterances, which often involve reference to other times and circumstances, and to the mental states of people.

In particular, to model individual belief we define a modal operator BELIEVE, that enables us to make assertions of the form BELIEVE(A, P), which is true whenever A believes P to be true. Notice that this can occur even if P is believed by someone else to be false or even if P is false.

Another useful modal operator is KNOW:

$$\text{BELIEVE}(A, P) \wedge P \rightarrow \text{KNOW}(A, P)$$

A third useful modal operator is KNOW-WHAT(A, P), which is true if A knows the value of the function P . For example, we might say that A knows the value of his age.

An alternative way to represent individual beliefs is to use the idea of knowledge base partitioning that we discussed in Section 9.1. Partitioning enables us to do two things:

1. Represent efficiently the large set of beliefs shared by the participants. We discussed one way of doing this above.
2. Represent accurately the smaller set of beliefs that are not shared.

Requirement 1 makes it imperative that shared beliefs not be duplicated in the representation. This suggests that a single knowledge base must be used to represent the beliefs of all the participants. But requirement 2 demands that it be possible to separate the beliefs of one person from those of another. One way to do this is to use partitioned semantic nets. Figure 15.20 shows an example of a partitioned belief space.

Three different belief spaces are shown:

- S1 believes that Mary hit Bill.

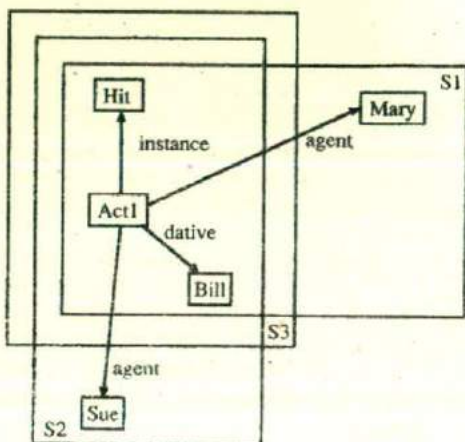


Figure 15.20: A Partitioned Semantic Net Showing Three Belief Spaces

- S2 believes that Sue hit Bill.
- S3 believes that someone hit Bill. It is important to be able to handle incomplete beliefs of this kind, since they frequently serve as the basis for questions, such as, in this case, "Who hit Bill?"

15.4.3 Using Goals and Plans for Understanding

Consider the text

John was anxious to get his daughter's new bike put together before Christmas Eve. He looked high and low for a screwdriver.

To understand this story, we need to recognize that John had

1. A goal, getting the bike put together.
2. A plan, which involves putting together the various subparts until the bike is complete. At least one of the resulting subplans involves using a screwdriver to screw two parts together.

Some of the common goals that can be identified in stories of all sorts (including children's stories, newspaper reports, and history books) are

- Satisfaction goals, such as sleep, food, and water.
- Enjoyment goals, such as entertainment and competition.
- Achievement goals, such as possession, power, and status.

- Preservation goals, such as health and possessions.
- Pleasing goals, which involve satisfying some other kind of goal for someone else.
- Instrumental goals, which enable preconditions for other, higher-level goals.

To achieve their goals, people exploit plans. In Chapter 13, we talked about several computational representations of plans. These representations can be used to support natural language processing, particularly if they are combined with a knowledge base of operators and stored plans that describe the ways that people often accomplish common goals. These stored operators and plans enable an understanding system to form a coherent representation of a text even when steps have been omitted, since they specify things that must have occurred in the complete story. For example, to understand this simple text about John, we need to make use of the fact that John was exploiting the operator USE (by A of P to perform G), which can be described as:

USE(A, P, G):
 precondition: KNOW-WHAT($A, \text{LOCATION}(P)$)
 NEAR(A, P)
 HAS-CONTROL-OF(A, P)
 READY(P)
 postcondition: DONE(G)

In other words, for A to use P to perform G , A must know the location of P , A must be near P , A must have control of P (for example, I cannot use a screwdriver that you are holding and refuse to give to me), and P must be ready for use (for example, I cannot use a broken screwdriver).

In our story, John's plan for constructing the bike includes using a screwdriver. So he needs to establish the preconditions for that use. In particular, he needs to know the location of the screwdriver. To find that out, he makes use of the operator LOOK-FOR:

LOOK-FOR(A, P):
 precondition: CAN-RECOGNIZE(A, P)
 postcondition: KNOW-WHAT($A, \text{LOCATION}(P)$)

A story understanding program can connect the goal of putting together the bike with the activity of looking for a screwdriver by recognizing that John is looking for a screwdriver so that he can use it as part of putting the bike together.

Often there are alternative operators or plans for achieving the same goal. For example, to find out where the screwdriver was, John could have asked someone. Thus the problem of constructing a coherent interpretation of a text or a discourse may involve considering many partial plans and operators.

Plan recognition has served as the basis for many understanding programs. PAM [Wilensky, 1981] is an early example; it translated stories into a CD representation.

Another such program was BORIS [Dyer, 1983]. BORIS used a memory structure called the Thematic Abstraction Unit to organize knowledge about plans, goals, interpersonal relationships, and emotions. For other examples, see Allen and Perrault [1980] and Sidner [1985].

15.4.4 Speech Acts

Language is a form of behavior. We use it as one way to accomplish our goals. In essence, we make communicative plans in much the same sense that we make plans for anything else [Austin, 1962]. In fact, as we just saw in the example above, John could have achieved his goal of locating a screwdriver by asking someone where it was rather than by looking for it. The elements of communicative plans are called *speech acts* [Searle, 1969]. We can axiomatize speech acts just as we axiomatized other operators in the previous section, except that we need to make use of modal operators that describe states of belief, knowledge, wanting, etc. For example, we can define the basic speech act A INFORM B of P as follows:

INFORM(A, B, P)

precondition: BELIEVE(A, P)

KNOW-WHAT($A, \text{LOCATION}(B)$)

postcondition: BELIEVE($B, \text{BELIEVE}(A, P)$)

BELIEVE-IN(B, A) \rightarrow BELIEVE(B, P)

To execute this operation, A must believe P and A must know where B is. The result of this operator is that B believes that A believes P , and if B believes in the truth of what A says, then B also believes P .

We can define other speech acts similarly. For example, we can define ASK-WHAT (in which A asks B the value of some predicate P):

ASK-WHAT(A, B, P):

precondition: KNOW-WHAT($A, \text{LOCATION}(B)$)

KNOW-WHAT(B, P)

WILLING-TO-PERFORM

($B, \text{INFORM}(B, A, P)$)

postcondition: KNOW-WHAT(A, P)

This is the action that John could have performed as an alternative way of finding a screwdriver.

We can also define other speech acts, such as A REQUEST B to perform R :

REQUEST(A, B, R)

precondition: KNOW-WHAT($A, \text{LOCATION}(B)$)

CAN-PERFORM(B, R)

WILLING-TO-PERFORM(*B, R*)postcondition: WILL(PERFORM(*B, R*))

15.4.5 Conversational Postulates

Unfortunately, this analysis of language is complicated by the fact that we do not always say exactly what we mean. Instead, we often use *indirect speech acts*, such as "Do you know what time it is?" or "It sure is cold in here." Searle [1975] presents a linguistic theory of such indirect speech acts. Computational treatments of this phenomenon usually rely on models of the speaker's goals and of ways that those goals might reasonably be achieved by using language. See, for example, Cohen and Perrault [1979].

Fortunately, there is a certain amount of regularity in people's goals and in the way language can be used to achieve them. This regularity gives rise to a set of *conversational postulates*, which are rules about conversation that are shared by all speakers. Usually these rules are followed. Sometimes they are not, but when this happens, the violation of the rules communicates something in itself. Some of these conversational postulates are:

- **Sincerity Conditions**—For a request by *A* of *B* to do *R* to be sincere, *A* must want *B* to do *R*, *A* must assume *B* can do *R*, *A* must assume *B* is willing to do *R*, and *A* must believe that *B* would not have done *R* anyway. If *A* attempts to verify one of these conditions by asking a question of *B*, that question should normally be interpreted by *B* as equivalent to the request *R*. For example,

A: Can you open the door?

- **Reasonableness Conditions**—For a request by *A* of *B* to do *R* to be reasonable, *A* must have a reason for wanting *R* done, *A* must have a reason for assuming that *B* can do *R*, *A* must have a reason for assuming that *B* is willing to do *R*, and *A* must have a reason for assuming that *B* was not already planning to do *R*. Reasonableness conditions often provide the basis for challenging a request. Together with the sincerity conditions described above, they account for the coherence of the following interchange:

A: Can you open the door?

B: Why do you want it open?

- **Appropriateness Conditions**—For a statement to be appropriate, it must provide the correct amount of information, it must accurately reflect the speaker's beliefs, it must be concise and unambiguous, and it must be polite. These conditions account for *A*'s response in the following interchange:

A: Who won the race?

B: Someone with long, dark hair.

A: I thought you knew all the runners.

A inferred from *B*'s incomplete response that *B* did not know *who* won the race, because if *B* had known she would have provided a name.

Of course, sometimes people "cop out" of these conventions. In the following dialogue, *B* is explicitly copping out:

A: Who is going to be nominated for the position?

B: I'm sorry, I cannot answer that question.

But in the absence of such a cop out, and assuming a cooperative relationship between the parties to a dialogue, the shared assumption of these postulates greatly facilitates communication. For a more detailed discussion of conversational postulates, see Grice [1975] and Gordon and Lakoff [1975].

We can axiomatize these conversational postulates by augmenting the preconditions for the speech acts that we have already defined. For example, we can describe the sincerity conditions by adding the following clauses to the precondition for REQUEST(*A*, *B*, *R*):

WANT(*A*, PERFORM(*B*, *R*))

BELIEVE(*A*, CAN-PERFORM(*B*, *R*))

BELIEVE(*A*, WILLING-TO-PERFORM(*B*, *R*))

BELIEVE(*A*, ¬WILL(PERFORM(*B*, *R*)))

If we assume that each participant in a dialogue is following these conventions, then it is possible to infer facts about the participants' belief states from what they say. Those facts can then be used as a basis for constructing a coherent interpretation of a discourse as a whole.

To summarize, we have just described several techniques for representing knowledge about how people act and talk. This knowledge plays an important role in text and discourse understanding, since it enables an understander to fill in the gaps left by the original writer or speaker. It turns out that many of these same mechanisms, in particular those that allow us to represent explicitly the goals and beliefs of multiple agents, will also turn out to be useful in constructing distributed reasoning systems, in which several (at least partially independent) agents interact to achieve a single goal. We come back to this topic in Section 16.3.

15.5 Summary

In this chapter, we presented a brief introduction to the surprisingly hard problem of language understanding. Recall that in Chapter 14, we showed that at least one understanding problem, line labeling, could effectively be viewed as a constraint satisfaction problem. One interesting way to summarize the natural language understanding problem that we have described in this chapter is to view it too as a constraint satisfaction problem. Unfortunately, many more kinds of constraints must be considered, and even when they are all exploited, it is usually not possible to avoid the guess and search part of the constraint satisfaction procedure. But constraint satisfaction does provide a reasonable framework in which to view the whole collection of steps that together create a meaning for a sentence. Essentially each of the steps described in this chapter exploits a particular kind of knowledge that contributes a specific set of constraints that must be satisfied by any correct final interpretation of a sentence.

Syntactic processing contributes a set of constraints derived from the grammar of the language. It imposes constraints such as:

- Word order, which rules out, for example, the constituent, "manager the key," in the sentence, "I gave the apartment manager the key."
- Number agreement, which keeps "trial run" from being interpreted as a sentence in "The first trial run was a failure."
- Case agreement, which rules out, for example, the constituent, "me and Susan gave one to Bob," in the sentence, "Mike gave the program to Alan and me and Susan gave one to Bob."

Semantic processing contributes an additional set of constraints derived from the knowledge it has about entities that can exist in the world. It imposes constraints such as:

- Specific kinds of actions involve specific classes of participants. We thus rule out the baseball field meaning of the word "diamond" in the sentence, "John saw Susan's diamond shimmering from across the room."
- Objects have properties that can take on values from a limited set. We thus rule out Bill's mixer as a component of the cake in the sentence, "John made a huge wedding cake with Bill's mixer."

Discourse processing contributes a further set of constraints that arise from the structure of coherent discourses. These include:

- The entities involved in the sentence must either have been introduced explicitly or they must be related to entities that were. Thus the word "it" in the discourse "John had a cold. Bill caught it," must refer to John's cold. This constraint can propagate through other constraints. For example, in this case, it can be used to determine the meaning of the word "caught" in this discourse, in contrast to its meaning in the discourse, "John threw the ball. Bill caught it."
- The overall discourse must be coherent. Thus, in the discourse, "I needed to deposit some money, so I went down to the bank," we would choose the financial institution reading of bank over the river bank reading. This requirement can even cause a later sentence to impose a constraint on the interpretation of an earlier one, as in the discourse, "I went down to the bank. The river had just flooded, and I wanted to see how bad things were."

And finally, pragmatic processing contributes yet another set of constraints. For example,

- The meaning of the sentence must be consistent with the known goals of the speaker. So, for example, in the sentence, "Mary was anxious to get the bill passed this session, so she moved to table it," we are forced to choose the (normally British) meaning of table (to put it on the table for discussion) over the (normally American) meaning (to set it aside for later).

There are many important issues in natural language processing that we have barely touched on here. To learn more about the overall problem, see Allen [1987], Cullingford [1986], Dowty *et al.* [1985], and Grosz *et al.* [1986]. For more information on syntactic processing, see Winograd [1983] and King [1983]. See Joshi *et al.* [1981] for more discussion of the issues involved in discourse understanding. Also, we have restricted our discussion to natural language understanding. It is often useful to be able to go the other way as well, that is, to begin with a logical description and render it into English. For discussions of natural language generation systems, see McKeown and Swartout [1987] and McDonald and Bolc [1988]. By combining understanding and generation systems, it is possible to attack the problem of *machine translation*, by which we understand text written in one language and then generate it in another language. See Slocum [1988], Nirenburg [1987], Lehrberger and Bourbeau [1988], and Nagao [1989] for discussions of a variety of approaches to this problem.

15.6 Exercises

1. Consider the sentence

The old man's glasses were filled with sherry.

What information is necessary to choose the correct meaning for the word "glasses"? What information suggests the incorrect meaning?

2. For each of the following sentences, show a parse tree. For each of them, explain what knowledge, in addition to the grammar of English, is necessary to produce the correct parse. Expand the grammar of Figure 15.6 as necessary to do this.

- John wanted to go to the movie with Sally.
- John wanted to go to the movie with Robert Redford.
- I heard the story listening to the radio.
- I heard the kids listening to the radio.
- All books and magazines that deal with controversial topics have been removed from the shelves.
- All books and magazines that come out quarterly have been removed from the shelves.

3. In the following paragraph, show the antecedents for each of the pronouns. What knowledge is necessary to determine each?

John went to the store to buy a shirt. The salesclerk asked him if he could help him. He said he wanted a blue shirt. The salesclerk found one and he tried it on. He paid for it and left.

4. Consider the following sentence:

Put the red block on the blue block on the table.

- (a) Show all the syntactically valid parses of this sentence. Assume any standard grammatical formalism you like.

- (b) How could semantic information and world knowledge be used to select the appropriate meaning of this command in a particular situation?

After you have done this, you might want to look at the discussion of this problem in Church and Patil [1982].

5. Each of the following sentences is ambiguous in at least two ways. Because of the type of knowledge represented by each sentence, different target languages may be useful to characterize the different meanings. For each of the sentences, choose an appropriate target language and show how the different meanings would be represented:
- Everyone doesn't know everything.
 - John saw Mary and the boy with a telescope.
 - John flew to New York.
6. Write an ATN grammar that recognizes verb phrases involving auxiliary verbs. The grammar should handle such phrases as
- "went"
 - "should have gone"
 - "had been going"
 - "would have been going"
 - "would go"

Do not expect to produce an ATN that can handle all possible verb phrases. But do design one with a reasonable structure that handles most common ones, including the ones above. The grammar should create structures that reflect the structures of the input verb phrases.

7. Show how the ATN of Figures 15.8 and 15.9 could be modified to handle passive sentences.
8. Write the rule "S \rightarrow NP VP" in the graph notation that we defined in Section 15.2.3. Show how unification can be used to enforce number agreement between the subject and the verb.
9. Consider the problem of providing an English interface to a database of employee records.
- (a) Write a semantic grammar to define a language for this task.
 - (b) Show a parse, using your grammar, of each of the two sentences
What is Smith's salary?
Tell me who Smith's manager is.
 - (c) Show parses of the two sentences of part (b) using a standard syntactic grammar of English. Show the fragment of the grammar that you use.
 - (d) How do the parses of parts (b) and (c) differ? What do these differences say about the differences between syntactic and semantic grammars?

10. How would the following sentences be represented in a case structure:
- The plane flew above the clouds.
 - John flew to New York.
 - The co-pilot flew the plane.
11. Both case grammar and conceptual dependency produce representations of sentences in which noun phrases are described in terms of their semantic relationships to the verb. In what ways are the two approaches similar? In what ways are they different? Is one a more general version of the other? As an example, compare the representation of the sentence
- John broke the window with a hammer.
- in the two formalisms.
12. Use compositional semantics and a knowledge base to construct a semantic interpretation of each of the following sentences:
- A student deleted my file.
 - John asked Mary to print the file.
- To do this, you will need to do all the following things:
- Define the necessary knowledge base objects.
 - Decide what the output of your parser will be assumed to be.
 - Write the necessary semantic interpretation rules.
 - Show how the process proceeds.
13. Show how conversational postulates can be used to get to the most common, coherent interpretation of each of the following discourses:
- A: Do you have a comb?
 - A: Would Jones make a good programmer?
B: He's a great guy. Everyone likes him.
 - A (in a store): Do you have any money?
B (A's friend): What do you want to buy?
14. Winograd and Flores [1986] present an argument that it is wrong to attempt to make computers understand language. Analyze their arguments in light of what was said in this chapter.

Chapter 16

Parallel and Distributed AI

Recent years have seen significant advances in parallel computation and distributed systems. What are the implications of these advances for AI? There are three main areas in which parallel and distributed architectures can contribute to the study of intelligent systems:

- Psychological modeling
- Improving efficiency
- Helping to organize systems in a modular fashion

These areas are often overlapping and complementary. For example, consider the production system model that we described in Chapter 2. The ideas of short-term and long-term memory, independently operating productions, matching, and so forth first arose in the psychological literature. When researchers began building AI systems based on these principles, they realized that parallel computers might be used to increase significantly the speed at which the systems could run. Even on single processor systems, however, the production system architecture turned out to have many benefits over conventional programming. One benefit is better modularity. When rules operate more or less independently, it is easy to add, delete, or modify them without changing the structure of the entire program. In this chapter, we discuss all these issues. First we briefly discuss psychological modeling. Then, in the following two sections we present some specific techniques that can be exploited in constructing parallel and distributed reasoning systems.

16.1 Psychological Modeling

The production system was originally proposed as a model of human information processing, and it continues to play a role in psychological modeling. Some production system models stress the sequential nature of production systems, i.e., the manner in which short-term memory is modified over time by the rules. Other models stress the parallel aspect, in which all productions match and fire simultaneously, no matter how

many there are. Both types of models have been used to explain timing data from experiments on human problem solving.

SOAR [Laird *et al.*, 1987] is the production system architecture that we mentioned in Chapter 6. SOAR has a dual mission. On the one hand, it is intended as an architecture for building integrated AI systems; on the other hand, it is intended as a model of human intelligence [Newell, 1991]. SOAR incorporates both sequential and parallel aspects of production systems by operating in cycles. In the *elaboration phase* of the processing cycle, productions fire in parallel. In the *decision phase*, operators and states are chosen, and working memory is modified, thus setting the stage for another elaboration phase. By tying these phases to particular timings, SOAR accounts for a number of psychological phenomena.

Another approach to psychological modeling draws its inspiration from the physical organization of the human brain itself. While individual neurons are quite slow compared to digital computer circuits, there are vast numbers of these richly interconnected components, and they all operate concurrently. If we wish to model the brain or use it as a source of ideas for AI, we must consider the powers and constraints imposed by the brain's architecture at the neural level. Unfortunately, we do not understand very well how neurons are wired in the brain, so modeling at this level is difficult. But we return to this idea in Chapter 18, where we describe the use of *neural networks* as a way of representing and using knowledge.

16.2 Parallelism in Reasoning Systems

AI programs consume significant time and space resources. It is therefore important that AI algorithms make use of advances in parallel computation. In this section, we describe several ways of doing this without substantially changing the programs that we write. Then, in the next section, we explore ways in which techniques from parallel and distributed computing can be used in the overall design of AI systems.

16.2.1 Parallelizing AI Architectures

As we mentioned above, production systems have both sequential and parallel aspects. The question arises, how much speedup can we expect from parallel processing? There are several sources of parallel speedup in production systems:

- Match-level parallelism, in which multiple processors are used to speed up the handling of individual match-resolve-act cycles
 - Production-level parallelism, in which all of the productions match themselves against working memory in parallel
 - Condition-level parallelism, in which all of the conditions of a single production are matched in parallel
 - Action-level parallelism, in which all of the actions of a single production are executed in parallel
- Task-level parallelism, in which several cycles are executed simultaneously

The amount of task-level parallelism available is completely dependent on the nature of the task. In a medical diagnosis system, for example, each production firing might be dependent on the previous production firing, thus enabling a long, sequential chain of reasoning to occur. However, if the system were diagnosing five patients simultaneously, productions involving different patients would not interact with one another and could be executed in parallel.

Match-level parallelism is more widely applicable. Since production systems spend nearly all of their time in the matching phase, it was expected early on that match-level parallelism would lead to vast speedups. In a system with a thousand productions, for example, one processor could be assigned to every production, possibly speeding up every match cycle by a factor of a thousand. However, as Gupta [1985] showed, having n processors does not lead to an n -fold speedup. Some reasons for this effect are:

1. Only a few productions are affected by each change in working memory. With some bookkeeping to save state information, sequential implementations such as RETE [Forgy, 1982] (Section 6.4.2) can avoid processing large numbers of productions. Parallel implementations must be judged with respect to the speedups they offer over efficient sequential algorithms, not inefficient ones.
2. Some productions are very expensive to match, while others are cheap. This means that many processors may sit idle waiting for others to finish. When processors are idle, the speedup available from parallel processing diminishes.
3. Overhead resulting from communication costs among multiple processors can further reduce the benefits of parallelism.

Other architectures behave differently with respect to parallel implementation. The brain-style architectures mentioned above are naturally parallel; in fact, simulating them on sequential machines is often prohibitive because of the high degree of parallelism they assume. In Section 16.3, we discuss some other parallel AI architectures.

16.2.2 Parallelizing AI Programming Languages

In the last section, we discussed the benefits of parallelizing a particular kind of program, namely a production system interpreter. Other frequently used interpreters in AI include those for the programming languages LISP and PROLOG.

Writing parallel programs is a difficult task for humans, and there is some hope that parallel implementations of these languages (perhaps augmented with parallel programming constructs) will make effective speedups more practical. Parallel LISP models include Multilisp [Halstead, 1988], QLISP [Gabriel and McCarthy, 1988], and the Parallelation Model [Sabot, 1988]. Parallel PROLOG models include Concurrent PROLOG [Shapiro, 1987], PARLOG [Clark and Gregory, 1986], and Guarded Horn Clauses [Ueda, 1985].

Research into parallel logic programming languages was an important focus of the Japanese Fifth Generation project [ICOT, 1984]. Languages like PROLOG immediately suggest two types of parallelism. In *OR-parallelism*, multiple paths to the same goal are taken in parallel. For example, suppose we have the following clauses:


```
uncle(X,Y) :- mother(Z,Y), sibling(X,Z).
uncle(X,Y) :- father(Z,Y), sibling(X,Z).
```

Then the query

```
?- uncle(John,Bill)
```

could be satisfied in two different ways since John could be the sibling of Bill's mother or of Bill's father. A sequential implementation would try to satisfy the first condition, and then, if that failed, try the second condition. There is no reason, however, why these two paths could not be pursued in parallel.¹

In *AND-parallelism*, the portions of a conjunctive goal are pursued in parallel. Consider the clause:

```
infieldfly(X) :- fly(X), infieldcatchable(X),
                occupiedbase(first), outs(zero).
```

Here, the four conditions can be checked in parallel, possibly leading to a four-fold speedup in processing *infieldfly* queries. Such *AND-parallelism* is not so straightforward when variables are shared across goals, as in:

```
uncle(X,Y) :- mother(Z,Y), sibling(X,Z).
```

The *mother(Z,Y)* and *sibling(X,Z)* conditions cannot be satisfied independently, since they must instantiate the variable *Z* in the same manner.

Research on *parallel logic programming* shares the same goal as that on *parallel production systems*: to permit the efficient execution of high-level, easily written code for AI systems.

16.2.3 Parallelizing AI Algorithms

Some problems are more amenable to parallel solutions than others. While nine authors may be able to write a book much faster than one author (if they each write separate chapters), nine women cannot bear a child any faster than one can. Likewise, throwing more processors at an AI problem may not bring the desired benefits. One example of an inherently sequential problem in AI is unification (recall Section 5.4.4). While multiple processors can help somewhat [Vitter and Simons, 1986], formal arguments [Dwork *et al.*, 1984] show that vast speedups in the unification of large terms are not possible.

Many problems can be solved efficiently by parallel methods, but it is not always a simple matter to convert a sequential algorithm into an efficient parallel one. Some AI algorithms whose parallel aspects have been studied are best-first search [Kumar *et al.*, 1988], alpha-beta pruning [Hsu, 1989], constraint satisfaction [Kasif, 1986], natural language parsing [Thompson, 1989], resolution theorem proving [Cheng and Juang, 1987], and property inheritance [Fahlman, 1979].

¹In PROLOG, clauses are matched sequentially from top to bottom. If PROLOG programmers write code that depends on this behavior, OR-parallelism may yield undesired results.

16.2.4 Custom Parallel Hardware

Finally, we must ask how these parallel algorithms can be implemented in hardware. One approach is to code an algorithm in a programming language supported by a general-purpose parallel computer. Another approach is to build custom parallel hardware that directly implements a single algorithm. This last approach has led to striking performance increases, as demonstrated in the SPHINX [Lee and Hon, 1988] speech recognition system, where real-time performance was achieved through the use of a beam search accelerator [Bisiani *et al.*, 1989], and in the DEEP THOUGHT chess machine [Hsu, 1989], which uses a parallel tree-search algorithm for searching game trees.

16.3 Distributed Reasoning Systems

In all of our discussions of problem-solving systems until now, we have focused on the design of single systems. In this section, we expand our view, and look at *distributed reasoning systems*. We define a distributed reasoning system to be one that is composed of a set of separate modules (often called *agents* since each module is usually expected to act as a problem-solving entity in its own right) and a set of communication paths between them. This definition is intentionally very vague. It admits systems everywhere along a spectrum that ranges from tightly coupled systems in which there is a completely centralized control mechanism and a shared knowledge base to ones in which both control and knowledge are fully distributed. In fact, of course, most real distributed reasoning systems lie somewhere in the middle. This definition also includes systems that are distributed at varying levels of granularity, although we do not intend it to include systems with very fine granularity (such as connectionist systems in which the individual nodes do not perform reasoning in the same sense that we have been using the term).

For many kinds of applications, distributed reasoning systems have significant advantages over large monolithic systems. These advantages can include:

1. System Modularity—It is easier to build and maintain a collection of quasi-independent modules than one huge one.²
2. Efficiency—Not all knowledge is needed for all tasks. By modularizing it, we gain the ability to focus the problem-solving system's efforts in ways that are most likely to pay off.
3. Fast Computer Architectures—As problem solvers get more complex, they need more and more cycles. Although machines continue to get faster, the real speed-ups are beginning to come not from a single processor with a huge associated memory, but from clusters of smaller processors, each with its own memory. Distributed systems are better able to exploit such architectures.
4. Heterogeneous Reasoning—The problem-solving techniques and knowledge representation formalisms that are best for working on one part of a problem may not be the best for working on another part.

²In this respect, reasoning programs are no different from other large programs [Dijkstra, 1972].

5. **Multiple Perspectives**—The knowledge required to solve a problem may not reside in the head of a single person. It is very difficult to get many diverse people to build a single, coherent knowledge base, and sometimes it is impossible because their models of the domain are actually inconsistent.
6. **Distributed Problems**—Some problems are inherently distributed. For example, there may be different data available in each of several distinct physical locations.
7. **Reliability**—If a problem is distributed across agents on different systems, problem solving can continue even if one system fails.

An architecture for distributed reasoning must provide:

1. A mechanism for ensuring that the activities of the various agents in the system are coordinated so that the overall problem-solving system achieves its goal(s).
2. A communication structure that enables information to be passed back and forth among agents.
3. Distributed versions of the necessary reasoning techniques. These mechanisms are likely to differ from their monolithic counterparts since they will be presumed to operate on a set of local knowledge bases rather than on a global one that can be assumed to possess a set of global properties (such as consistency).

In the rest of this section, we address each of these issues.

16.3.1 Coordination and Cooperation

The biggest issue that needs to be faced in the design of any distributed reasoning system is how the actions of the individual agents can be coordinated so that they work together effectively. There are a variety of approaches that can be taken here, including the following:

- One agent is in charge. That master agent makes a plan and distributes pieces of the plan to other "slave" agents, who then do as they are told and report back their results. They may also communicate with other slave agents if necessary to accomplish their goals.
- One agent is in charge and that agent decomposes the problem into subproblems, but then negotiation occurs to decide what agents will take responsibility for which subtasks.
- No one agent is in charge, although there is a single shared goal among all the agents. They must cooperate both in forming a plan and in executing it.
- No one agent is in charge, and there is no guarantee that a single goal will be shared among all the agents. They may even compete with each other.

Although these approaches differ considerably, there is one modification to a simple, single agent view of reasoning that is necessary to support all of them in anything other than a trivial way. We need a way to represent models of agents, including what they know, what they can do, and what their goals are. Fortunately, what we need is exactly the set of mechanisms that we introduced in Section 15.4. But now, instead of using modal operators and predicates (such as BELIEVE, KNOW, KNOW-WHAT, CAN-PERFORM, and WILLING-TO-PERFORM) to model writers and speakers, we use them to model agents in a distributed system. Using such operators, it is possible for each agent to build a model of both itself and the other agents with which it must interact. The self-descriptive model is necessary to enable the agent to know when it should get help from others and to allow it to represent itself accurately to other agents who may wish to get help from it. The model of other agents is necessary to enable an agent to know how best to get help from them.

Planning for Multi-agent Execution

The least distributed form of distributed reasoning is that in which a single agent:

1. Decomposes the goal into subgoals, and
2. Assigns the subgoals to the various other agents

This kind of reasoning is usually called *multi-agent planning*. The first step, problem decomposition, is essentially the same as it is for single-agent planning systems. Ideally, the decomposition results in a set of subproblems that are mutually independent. This is often not possible, however, so various of the techniques that we described in Chapter 13 must be exploited.

Once a decomposition has been produced, the subproblems must be allocated to the available agents for execution. At this point, distributed planning differs from single agent planning in the following important ways:

- Unless all the slave agents are identical, the master agent must have access to models of the capabilities of the various slaves. These models make it possible to allocate tasks to the agents that are best able to perform them.
- Even if all the slave agents are identical, the master must do load balancing to assure that the overall goal is completed as soon as possible.
- Once the tasks have been distributed, synchronization among the slaves is necessary unless all the tasks are completely independent. In single-agent planning, dependencies are usually handled at plan creation time. In a multiple agent system, it is not usually possible to do that, since any such static scheme will be defeated if the various agents take unpredictable amounts of time to perform their tasks.

Let's consider this last issue in a bit more detail. Suppose the task is to do spelling correction on a document with several chapters, and then to print it. We can distribute this among several spelling correcting agents and one printing agent. But to get the desired result, we need to ensure that the printing agent does not begin printing any

chapter until spelling correction on that chapter is complete. Distributed reasoning systems exploit a wide variety of synchronization techniques to guarantee this, ranging from simple ones (e.g., in which the printing process does not begin until all the spelling correctors are done and have so informed the master) to more sophisticated ones in which the slave processes communicate directly with each other (e.g., the spelling correctors each inform the printer when they have finished). These more sophisticated techniques require that each slave agent be told some information about the other slaves at the time that it is given its task.

For relatively simple tasks, such as the one we just described, the various agents can communicate effectively with each other just by announcing when operations have been completed. For other kinds of tasks, though, it is not enough to know when an agent has completed its task. It may also be necessary to know what state the system is in during task execution. For example, suppose that there is a single resource that the various agents share, such as an input or output device. Then one agent may want to know whether any other is currently using the device. To support this kind of interaction, it is useful to introduce a state-based model, such as that described by Georgeff [1983; 1984]. In this kind of a model, each available action is characterized as a sequence of state changes that it effects. The various agents may share a single model, which they all update as necessary, or they may each have their own model, in which case they must also inform all other relevant agents whenever they make a change to their internal state that could be important externally.

Planning and Negotiation: Contract Nets

A slightly more distributed kind of reasoning occurs when a single agent performs the problem decomposition but then negotiates with the other agents to determine who will take on which subtasks. The *contract net* mechanism [Davis and Smith, 1983] supports this kind of interaction. In a contract net, there are two roles that the agents can assume:

1. Manager, who decomposes a problem, looks for contractors to attack pieces of the problem, and monitors the problem's execution.
2. Contractor, who executes a subtask, possibly by actually doing the job and possibly by recursively becoming a manager and subcontracting subparts of the job to other contractors.

Managers and contractors find each other through a process of bidding:

1. A manager announces a task.
2. Contractors evaluate the task with respect to their own abilities and the resource requirements necessary to accomplish it.
3. Contractors make bids to the manager.
4. The manager chooses a single contractor and waits for the result.

Thus managers and contractors select each other by communicating in a completely distributed fashion. A node can be both a manager and a contractor simultaneously; rather than sit idle, waiting for results from its contractors, a manager can take on work in the meantime.

Distributed Control and Communication

So far, we have focused on systems in which there is a single agent who maintains control of the overall problem-solving process. In this section, we look at the problem of *distributed planning*, in which there is no such centralized controller. In the extreme form of such systems, we can make no assumptions about how the various agents will behave. But without any such assumptions, it is impossible to construct problem-solving algorithms. So we start by assuming that each agent is rational. We can define rationality as follows:

An agent is *rational* if it behaves in a manner that is optimal with respect to its goals.

Unfortunately, in a complex world, an agent may not have enough processing power to behave optimally. This leads to a slightly weaker, but more useful, notion of bounded rationality [Simon, 1957]:

Bounded rationality is a property of an agent that behaves in a manner that is as nearly optimal with respect to its goals as its resources allow.

Bounded rationality is akin to the notion of satisficing that we discussed in Chapter 2.

Using these ideas, we can define techniques that an individual agent can use to attain its goals, taking into account what will probably happen as a result of what the other agents in its environment are likely to do. Sometimes, the other agents are cooperating to achieve the same goal. Sometimes they are working on other goals, which may be competitive or simply orthogonal. We consider two classes of approaches to this problem:

- Planning with communication
- Planning without communication

The first approach is one in which the agents can communicate freely with each other during problem solving. In this case, the agents can each create their own plans, which are composed both of problem-solving actions and of communication actions of the sort we described in Section 15.4. Sometimes the communication actions are addressed to a specific other agent who is believed to be able to satisfy a request (either for information or to perform some other task). In other systems, the agents do not know explicitly about each other. Instead, each agent can broadcast to a shared memory structure, which other agents can be counted on to read. Each agent can then reply to those messages to which it chooses to pay attention. We describe one specific way of implementing such a broadcast structure as a blackboard system in Section 16.3.2.

One specific technique that several communicating agents can use is called the *functionally accurate, cooperative* (FA/C) approach [Lesser and Corkill, 1981] to distributed problem solving. Each agent begins by forming a tentative, incomplete plan. These plans are then shared among the agents, who are able to help refine each other's plans by adding information that they possess. Ideally, the entire system converges on a complete plan.

		Q	
		c	d
P	a	2 4	1 2
	b	1 3	5 1

Figure 16.1: A Payoff Matrix for Two Agents and Two Actions

The second approach is one in which we assume that the agents cannot communicate. This may seem to be a very serious restriction, but it is useful to consider it both because it does sometimes arise in the extreme (perhaps because the agents are geographically isolated) and because it often arises at small granularity levels where the cost of constant communication may come to dominate the cost of actual problem solving.

If we assume that the agents cannot communicate and that they are all rational, then we can use many of the standard notions of game theory to describe how each of them should act. The most basic technique is that of a payoff matrix, such as the one shown in Figure 16.1. We assume that there are only two agents, P and Q, and that there are only two actions that each of them can perform (a and b for P, and c and d for Q). Then the matrix shows the payoff for each of them for each of the possible joint actions. The number in the lower left of each box is the payoff for P; the number in the upper right is the payoff for Q. Each agent's goal is to maximize its own payoff. For example, P comes out best if it makes move b and Q makes move d . On the other hand, Q comes out best if P makes move a and Q makes move c .

Of course, no one of the agents can force such a dual move. Each must make its own decision independently. In this case, for example, P should choose move a (rather than b , even though the best case for P included move b). Why? The answer is that P should assume that Q will behave rationally. In this matrix, the c column dominates the d column for Q, by which we mean that in every row, the payoff for Q is higher in the c column than in the d column. Thus Q can be predicted to choose c , and P should plan accordingly. Given that Q will choose c , P sees that it does better to choose move a than move b .

We can now view our discussion of game-playing programs (Chapter 12) from a different perspective, that of noncommunicating agents trying to solve their own goals. Both payoff matrices and tree-search algorithms can be generalized to more than two players (e.g., Korf [1989]), but there are some important differences. In board games, players usually take turns making moves, whereas payoff matrices model the kind of simultaneous decision making common in the real world. Also, games are usually zero-sum, meaning that one player's gain is another player's loss. Payoff matrices are

sometimes zero-sum, but need not be. See Genesereth *et al.* [1987] and Rosenschein and Breese [1989] for more substantial discussions of operations on payoff matrices.

16.3.2 Communication: Blackboards and Messages

The specific communication architectures that have been proposed to support distributed reasoning fall into two classes with respect to communication structure:

- *Blackboard systems*, in which communication takes place through a shared knowledge structure called a blackboard. Modules can post items on the blackboard, and they can read and act on messages that are posted by other modules.
- *Message-passing systems*, in which one reasoning module sends messages (both requests for services and information as well as replies to such requests) to one or more other modules whose names are explicitly known.

Although on the surface, these two techniques appear quite different, they turn out in practice to offer essentially the same support for distributed reasoning. In fact, they can be used to simulate each other, as we see below. In the rest of this section, we describe examples of each approach.

Blackboard Systems

The blackboard approach to the organization of large AI programs was first developed in the context of the HEARSAY-II speech-understanding project [Erman *et al.*, 1980]. The idea behind the blackboard approach is simple. The entire system consists of:

- A set of independent modules, called knowledge sources (or KSs), that contain the system's domain-specific knowledge
- A blackboard, which is the shared data structure through which the knowledge sources communicate with each other
- A control system, which determines the order in which knowledge sources will operate on the entries on the blackboard

To see how these pieces work together, let's look at the HEARSAY-II system. Here, the KSs correspond to the levels of knowledge about speech, language (syllables, words, phrases, and sentences), and the task being discussed. The blackboard contains hypotheses about interpretations at each of these levels. Control is performed by a specialized knowledge source that reasons about such factors as cost of execution and likelihood of achieving a result.

When a KS is activated (as described below), it examines the current contents of the blackboard and applies its knowledge either to create a new hypothesis and write it on the blackboard, or to modify an existing one. Although the execution of the entire HEARSAY-II system consists of the asynchronous execution of a collection of KSs, the execution of an individual KS is a sequential process. Once a KS is activated, it executes without being interrupted until it is finished.

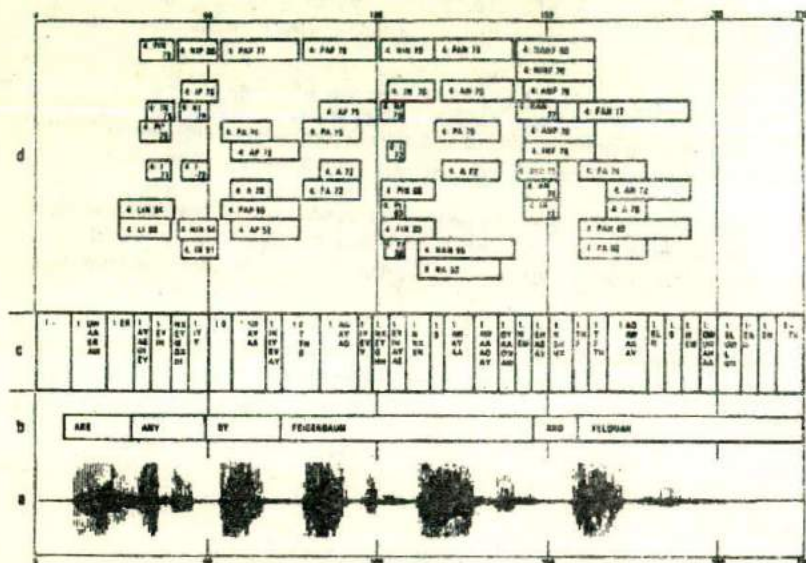


Figure 16.2: A Snapshot of a HEARSAY-II Blackboard

The hypotheses on the blackboard are arranged along two dimensions: level (from small, low-level hypotheses about individual sounds to large, high-level hypotheses about the meaning of an entire sentence) and time (corresponding to periods of the utterance being analyzed). The goal of the system is to create a single hypothesis that represents a solution to a problem. For HEARSAY-II, such a solution would be an acceptable interpretation of an entire utterance. Figures 16.2 and 16.3 show a snapshot of a HEARSAY-II blackboard. Figure 16.2 shows the lowest three levels of the blackboard, and Figure 16.3 shows the top three. The levels are the following:

- The waveform corresponding to the sentence "Are any by Feigenbaum and Feldman?"
- The correct words shown just for reference
- The sound segments
- The syllable classes
- The words as created by one word KS
- The words as created by a second word KS
- Word sequences
- Phrases

Associated with each KS is a set of *triggers* that specify conditions under which the KS should be activated. These triggers are an example of the general idea of a *demon*, which is, conceptually, a procedure that watches for some condition to become true and then activates an associated process.³

When a trigger fires, it creates an *activation record* describing the KS that should be activated and the specific event that fired the trigger. This latter information can be used to focus the attention of the KS when it is actually activated. Of course, a single event, such as the addition of a particular kind of hypothesis to the blackboard, could cause several triggers to fire at once, causing several activation records to be created. The KS that caused the triggering event to occur need not know about any of these subsequent activations. The actual determination of which KS should be activated next is done by a special KS, called the *scheduler*, on the basis of its knowledge about how best to conduct the search in the particular domain. The scheduler uses ratings supplied to it by each of the independent KSs. If the scheduler ever discovers that there are no activation records pending, then the system's execution terminates. For more information on the HEARSAY-II scheduler, see Hayes-Roth and Lesser [1977].

The techniques developed in HEARSAY-II have since been generalized in several multipurpose blackboard systems, including HEARSAY-III [Balzer *et al.*, 1980; Erman *et al.*, 1981], GBB [Corkill *et al.*, 1987], and BB1 [Hayes-Roth, 1985; Hayes-Roth and Hewett, 1989]. For example, the use of time as an explicit dimension on the blackboard is not appropriate in all domains, so it has been removed from these more general systems.

But these new blackboard systems also provide facilities that HEARSAY-II lacked. In HEARSAY-II, control was data-driven. This worked well for speech understanding. But for other kinds of problem solving, other kinds of control are more appropriate. Examples include control that is driven either by goals or by plans. The newer blackboard systems provide explicit support for these other control mechanisms. One important way in which they do that is to allow the use of multiple blackboards. Although this idea can also be exploited as a way to modularize domain reasoning, one of its important uses is to exploit one blackboard for reasoning in the problem domain and another for controlling that reasoning. In addition, these systems provide a goal-structured agenda mechanism that can be used in the control space to allow problem solving to be driven by an explicit goal structure. See Englemore and Morgan [1989] and Jagannathan *et al.* [1989] for further descriptions of these systems and some applications that have been built on top of them.

Message-Passing Systems

Message-passing systems provide an alternative way for agents in a distributed reasoning system to communicate with each other. In such a framework, the agents tend to know more about each other than they do in a blackboard system. This knowledge enables them to direct their messages to those agents who are most likely to be able to do what needs to be done. As an example of a message-passing distributed system, we describe MACE [Gasser *et al.*, 1987], which provides a general architecture for distributed reasoning

³Of course, demons usually are not actually implemented as processes that watch for things, but rather the things they are watching for are set up to activate them when appropriate.

systems (in the same sense that systems such as BBI provide a general architecture for blackboard systems). A MACE system is composed of five kinds of components:

1. Problem-solving agents, which are specialized to a problem domain
2. System agents, which provide such facilities as command interpretation, error handling, tracing
3. Facilities, which are built-in functions that agents can use for such things as pattern matching and simulation
4. A description database, which maintains descriptions of the agents
5. Kernels, of which there is one for each processor, which handle such functions as message routing and I/O transfers

A MACE problem-solving agent maintains models of other agents. A model that an agent P has of some other agent A contains the following information:

1. Name: A 's name
2. Class: A 's class
3. Address: A 's location
4. Role: A 's relationship to P . This relationship can be identity, creator, or member of an organization.
5. Skills: P 's knowledge about what A can do
6. Goals: P 's beliefs about A 's goals
7. Plans: P 's beliefs about A 's plans for achieving its goals

This architecture supports many of the kinds of distributed reasoning systems that we have been discussing. Let's consider a few.

First, suppose we want to build a system in which a controlling agent will decompose the problem and then negotiate with other agents to perform subtasks using a contract net mechanism. Then each of the agents can be represented as a problem-solving agent in MACE. The manager decomposes the problem. It then sends requests for bids to all the other agents, about which it knows nothing except their addresses. As the other agents respond, the manager can build up its model of them. Using that model, it can choose the agents to whom it wishes to award bids. The chosen agents perform their tasks and then send reply messages to the manager.

At another extreme, suppose we want to build a system that is composed of competing agents. We can model such a system in a MACE architecture, again by building a set of problem-solving agents, but this time their models of each other must be more sophisticated. In particular, it will be necessary to model each other's goals and plans.

Although MACE directly supports a message-passing communication protocol, it can be used to simulate a blackboard system. A single problem-solving agent, or a collection of them, can be used to simulate each blackboard knowledge source. Additional

agents can simulate the blackboard itself. Agents send messages to the blackboard, which in turn routes the messages to the other agents that should be triggered as a result of the posting.

As this example suggests, there really is no dichotomy between blackboard and message-passing systems so much as there is a continuum. At one extreme, an agent can do nothing but broadcast its message to everyone. At the other, an agent can do nothing except send its message to a specific other agent. There are many in-between positions that can be very useful. For example, an agent may not know exactly which other agent should receive its message, but it may know that it is some agent belonging to a particular class. In a message-passing architecture, this can be implemented by arranging agents into a hierarchy of classes and allowing messages to be sent to a class and thus delivered to all members of the class. In a blackboard system, this same capability can be implemented by creating a type hierarchy for blackboard elements. Then each KS is marked with the types of elements that will be considered as triggering events. When an element is posted, only those KSs that are defined for elements of that type will be given a chance to trigger on it.

16.3.3 Distributed Reasoning Algorithms

So far we have discussed various issues that arise when planning and plan execution are distributed across multiple agents. But we have not considered any modifications to any other reasoning algorithms. We have implicitly assumed that such standard procedures as matching and inheritance would work in a distributed system just as they do in a single-agent system. In many cases they do. But there are some reasoning algorithms, particularly ones that operate globally on a knowledge base, that need to be redesigned to support distributed reasoning. We consider one example of such an algorithm here.

Consider again the justification-based truth maintenance system (JTMS) that we described in Section 7.5.2. The JTMS works by considering an entire knowledge base and labeling the nodes in the knowledge base so that the labeling is consistent and well-founded. Both of these are global properties. But consider a distributed reasoning system in which there are several agents, each of which has its own knowledge base. Although we expect that each of these knowledge bases will be locally consistent, we do not want to insist that, taken together, they be globally consistent. This is important, since one of the benefits of a distributed system is that agents that represent different points of view and positions can interact. So what does it mean to label the nodes in such an inconsistent knowledge base?

A second question arises when we extend the notion of a JTMS to a distributed system. In a single-agent system, a justification is created as part of the reasoning process. It stays with the resulting node and can be used to update the belief status of the node if any of the assumptions on which the reasoning depended ever change. But what if one agent does the reasoning and then communicates its result to another? It may not make sense to communicate the justification, since it may involve knowledge-base objects that the receiver of the result knows nothing about. This will often happen if one agent asks another to solve a problem about which it knows very little.

Both of these problems can be solved by introducing the idea of a distributed truth maintenance system. In this system, interagent justifications work as follows. Assume A1 solves a problem and reports the result to A2. Then A1 also reports to

A2 a justification that says "Because A1 says so." This justification is treated by A2 essentially like a premise justification. But A1 must also remember the justification, and it must remember that it sent this justification to A2. If the justification ever becomes invalid in A1, then A1 must send a message to A2 saying that A1 no longer says so. At that point, the conclusion must go OUT in A2 unless there exists some other justification that is still valid.

Node labeling in the distributed truth maintenance system works similarly to node labeling in a single-agent system except that we need to redefine consistency. Rather than insisting on global consistency, we instead insist on extended local consistency, by which we mean that the labels within the knowledge base of a single agent must be consistent and the labels that are attached to nodes that have been explicitly shared among agents must be consistent across agents. But we do not insist that the labels attached to nodes that have not been explicitly shared be consistent across agents. For more information on how to do this, see Bridgeland and Huhns [1990]. For a similar discussion of ways to create a distributed assumption-based truth maintenance system, see Mason and Johnson [1989].

16.4 Summary

In this chapter, we discussed parallel and distributed aspects of AI. We examined psychological factors as well as efficiency concerns. The last section described the issues that arise when we attempt to extend the problem-solving mechanisms of earlier chapters to distributed reasoning systems. We have by no means covered all of them. For more information in this area, see the following collections: Huhns [1987], Bridgeland and Gasser [1988], and Gasser and Huhns [1989].

Before we end this chapter, we should point out that as distributed systems become more complex, it becomes harder to see how best to organize them. One thing that has proved promising is to look for analogies in the organization of other complex systems. One of the most promising sources of such analogies is the structure of human organizations, such as societies and corporations. A team or a corporation or a government is, after all, a distributed goal-oriented system. We have already seen one example of this idea, namely the bidding that is exploited in the contract net framework. See Fox [1981], Malone [1987], and Kornfeld and Hewitt [1981] for further discussion of this idea.

Another source of ideas is the way a single human brain functions. The book, *The Society of Mind* [Minsky, 1985] explores the notion that single minds are also distributed systems, composed of collections of heterogeneous agents that simultaneously cooperate and compete.

16.5 Exercises

1. Consider a situation in which one agent A1 requests help from a second agent A2 to help find a movie it would like. A1 knows what it likes and A2 knows about movies.

- (a) Using the belief and communication operators that we have defined (plus any others that you find it useful to define), write a plan that could be used by A1.
- (b) Write a similar plan for A2.
2. Consider the following payoff matrix:

		Q	
		<i>c</i>	<i>d</i>
P	<i>a</i>	2 4	4 5
	<i>b</i>	5 3	1 1

If Q assumes P is rational, what move should Q make?

3. Show how the HEARSAY-II blackboard system could be extended to support the whole natural language understanding process that we described in Chapter 15.
4. Show how a speech understanding system could be built using a MACE-style architecture.

Chapter 17

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 closure¹ 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.

All 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

¹The transitive closure of a program's knowledge is that knowledge plus whatever the program can logically deduce from it.

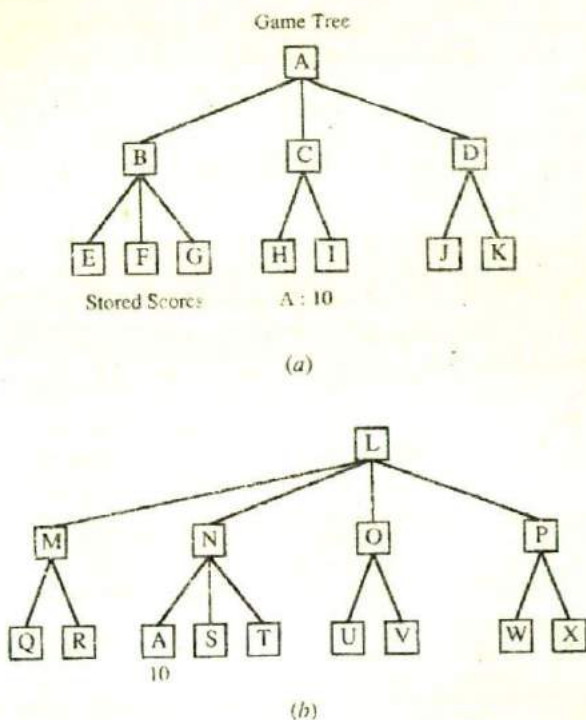


Figure 17.1: Storing Backed-Up Values

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 Figure 17.1(a).

Now suppose that in a later game, the situation shown in Figure 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.

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 p1 (players) (play-card p1))
               (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

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

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), STACK(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 postconditions ON(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:



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_1 , 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:



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 where 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, but 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 reevaluates 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 "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 També 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:

$$c_1t_1 + c_2t_2 + c_3t_3 + \dots$$

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 Figure 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.

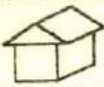
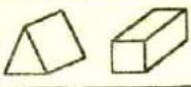
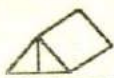

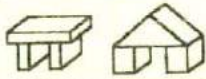
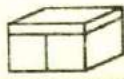
	Concept	Near Miss
House		
Tent		
Arch		

Figure 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 Figure 17.2 is shown in Figure 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 Figure 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 Figure 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.
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

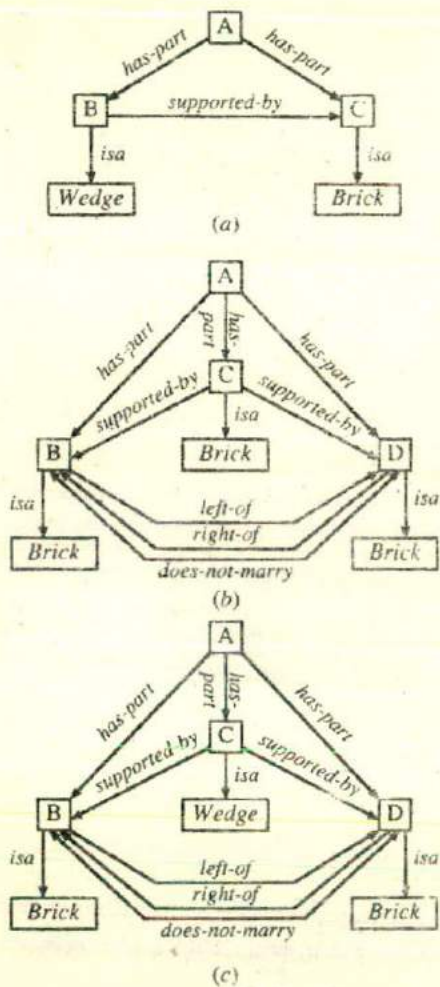


Figure 17.3: Structural Descriptions

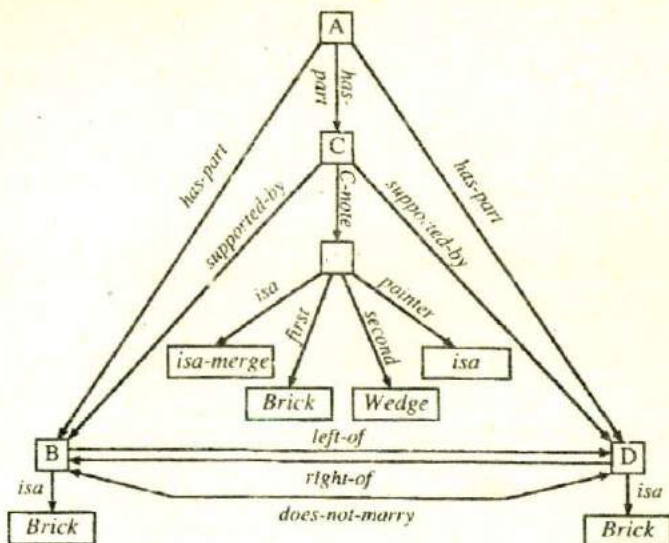


Figure 17.4: The Comparison of Two Arches

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 Figure 17.3(b) is presented first. It then becomes the definition of the concept *Arch*. Then suppose that the arch description of Figure 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 Figure 17.4. The *c-note* link from node C describes 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 Figure 17.5 can be built.

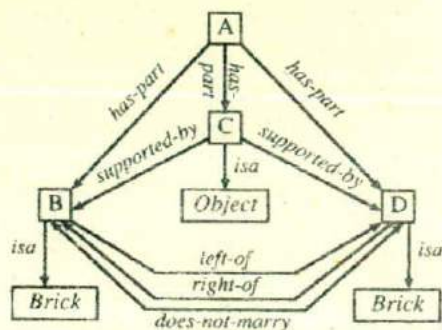


Figure 17.5: The Arch Description after Two Examples

Next, suppose that the near miss arch shown in Figure 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 Figure 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*.

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 describe 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

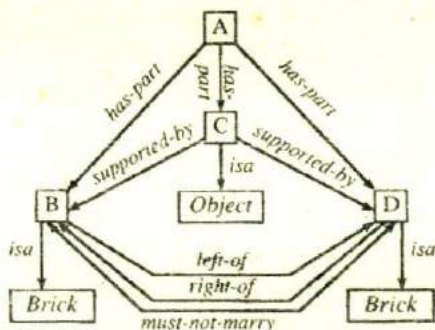


Figure 17.6: The Arch Description after a Near Miss

Car023

<i>origin :</i>	<i>Japan</i>
<i>manufacturer :</i>	<i>Honda</i>
<i>color :</i>	<i>Blue</i>
<i>decade :</i>	<i>1970</i>
<i>type :</i>	<i>Economy</i>

Figure 17.7: An Example of the Concept *Car*

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 Figure 17.7, a frame representing an individual car.

Now, suppose that each slot may contain only the discrete values shown in Figure 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.

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 Figure 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

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

Figure 17.8: Representation Language for Cars

<i>origin</i> :	Japan
<i>manufacturer</i> :	x_1
<i>color</i> :	x_2
<i>decade</i> :	x_3
<i>type</i> :	Economy

Figure 17.9: The Concept "Japanese economy car"

economy car. Now the learning problem is: Given a representation language such as in Figure 17.8, and given positive and negative training examples such as those in Figure 17.7, how can we produce a concept description such as that in Figure 17.9 that is consistent with all the training examples?

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 Figure 17.9 is more general than the one in Figure 17.7. In fact, the representation language defines a partial ordering of descriptions. A portion of that partial ordering is shown in Figure 17.10.

The entire partial ordering is called the *concept space*, and can be depicted as in Figure 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.

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

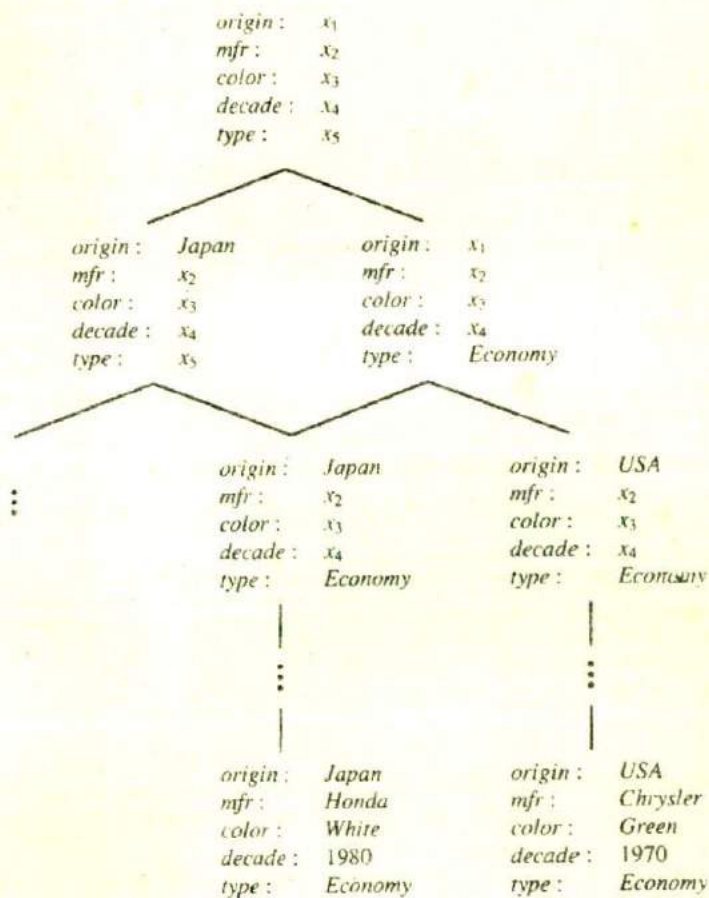


Figure 17.10: Partial Ordering of Concepts Specified by the Representation Language

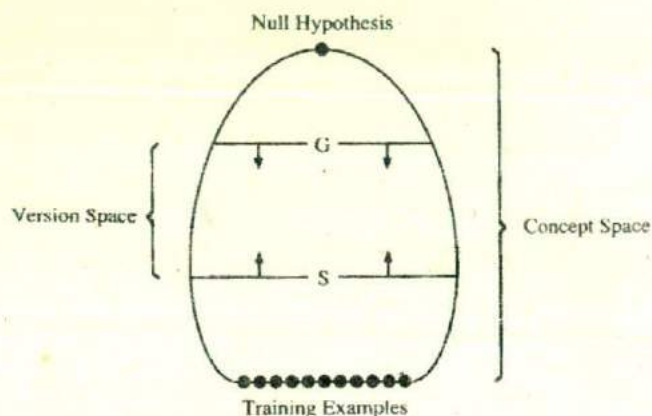


Figure 17.11: Concept and Version Spaces

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.

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*.

Algorithm: Candidate Elimination

Given: A representation language and a set of positive and negative examples expressed in that language.

Compute: A concept description that is consistent with all the positive examples and none of the negative examples.

1. Initialize G to contain one element: the null description (all features are variables).
2. Initialize S to contain one element: the first positive example.
3. Accept a new training example.

If it is a *positive example*, first remove from G any descriptions that do not cover the example. Then, update the S set to contain the most specific set of descriptions in the version space that cover the example and the current elements of the S set.

<i>origin</i> : Japan	<i>origin</i> : Japan	<i>origin</i> : Japan
<i>mfr</i> : Honda	<i>mfr</i> : Toyota	<i>mfr</i> : Toyota
<i>color</i> : Blue	<i>color</i> : Green	<i>color</i> : Blue
<i>decade</i> : 1980	<i>decade</i> : 1970	<i>decade</i> : 1990
<i>type</i> : Economy	<i>type</i> : Sports	<i>type</i> : Economy
(+)	(-)	(+)
<i>origin</i> : USA	<i>origin</i> : Japan	
<i>mfr</i> : Chrysler	<i>mfr</i> : Honda	
<i>color</i> : Red	<i>color</i> : White	
<i>decade</i> : 1980	<i>decade</i> : 1980	
<i>type</i> : Economy	<i>type</i> : Economy	
(-)	(+)	

Figure 17.12: Positive and Negative Examples of the Concept "Japanese economy car"

That is, generalize the elements of S as little as possible so that they cover the new training example.

If it is a *negative* example, first remove from S any descriptions that cover the example. Then, update the G set to contain the most general set of descriptions in the version space that *do not* cover the example. That is, specialize the elements of G as little as possible so that the negative example is no longer covered by any of the elements of G .

4. If S and G are both singleton sets, then if they are identical, output their value and halt. If they are both singleton sets but they are different, then the training cases were inconsistent. Output this result and halt. Otherwise, go to step 3.

Let us trace the operation of the candidate elimination algorithm. Suppose we want to learn the concept of "Japanese economy car" from the examples in Figure 17.12. G and S both start out as singleton sets. G contains the null description (see Figure 17.11), and S contains the first positive training example. The version space now contains all descriptions that are consistent with this first example:²

$$G = \{(x_1, x_2, x_3, x_4, x_5)\}$$

$$S = \{(Japan, Honda, Blue, 1980, Economy)\}$$

Now we are ready to process the second example. The G set must be specialized in such a way that the negative example is no longer in the version space. In our representation language, specialization involves replacing variables with constants. (Note: The G set must be specialized only to descriptions that are *within* the current version space, not outside of it.) Here are the available specializations:

²To make this example concise, we skip slot names in the descriptions. We just list slot values in the order in which the slots have been shown in the preceding figures.

$$G = \{(x_1, \text{Honda}, x_3, x_4, x_5), (x_1, x_2, \text{Blue}, x_4, x_5), \\ (x_1, x_2, x_3, 1980, x_5), (x_1, x_2, x_3, x_4, \text{Economy})\}$$

The S set is unaffected by the negative example. Now we come to the third example, a positive one. The first order of business is to remove from the G set any descriptions that are inconsistent with the positive example. Our new G set is:

$$G = \{(x_1, x_2, \text{Blue}, x_4, x_5), (x_1, x_2, x_3, x_4, \text{Economy})\}$$

We must now generalize the S set to include the new example. This involves replacing constants with variables. Here is the new S set:

$$S = \{(\text{Japan}, x_2, \text{Blue}, x_4, \text{Economy})\}$$

At this point, the S and G sets specify a version space (a space of candidate descriptions) that can be translated roughly into English as: "The target concept may be as specific as 'Japanese, blue economy car,' or as general as either 'blue car' or 'economy car.'"

Next, we get another negative example, a car whose *origin* is *USA*. The S set is unaffected, but the G set must be specialized to avoid covering the new example. The new G set is:

$$G = \{(\text{Japan}, x_2, \text{Blue}, x_4, x_5), (\text{Japan}, x_2, x_3, x_4, \text{Economy})\}$$

We now know that the car must be Japanese, because *all* of the descriptions in the version space contain *Japan* as *origin*.³ Our final example is a positive one. We first remove from the G set any descriptions that are inconsistent with it, leaving:

$$G = \{(\text{Japan}, x_2, x_3, x_4, \text{Economy})\}$$

We then generalize the S set to include the new example:

$$S = \{(\text{Japan}, x_2, x_3, x_4, \text{Economy})\}$$

S and G are both singletons, so the algorithm has converged on the target concept. No more examples are needed.

There are several things to note about the candidate elimination algorithm. First, it is a *least-commitment* algorithm. The version space is pruned as little as possible at each step. Thus, even if all the positive training examples are Japanese cars, the algorithm will not reject the possibility that the target concept may include cars of other origin—until it receives a negative example that forces the rejection. This means that if the training data are sparse, the S and G sets may never converge to a single description; the system may learn only partially specified concepts. Second, the algorithm involves exhaustive, breadth-first search through the version space. We can see this in the algorithm for

³It could be the case that our target concept is "not Chrysler," but we will ignore this possibility because our representation language is not powerful enough to express negation and disjunction.

updating the G set. Contrast this with the depth-first behavior of Winston's learning program. Third, in our simple representation language, the S set always contains exactly one element, because any two positive examples always have exactly one generalization. Other representation languages may not share this property.

The version space approach can be applied to a wide variety of learning tasks and representation languages. The algorithm above can be extended to handle continuously valued features and hierarchical knowledge (see Exercises). However, version spaces have several deficiencies. One is the large space requirements of the exhaustive, breadth-first search mentioned above. Another is that inconsistent data, also called *noise*, can cause the candidate elimination algorithm to prune the target concept from the version space prematurely. In the car example above, if the third training instance had been mislabeled (-) instead of (+), the target concept of "Japanese economy car" would never be reached. Also, given enough erroneous negative examples, the G set can be specialized so far that the version space becomes empty. In that case, the algorithm concludes that *no* concept fits the training examples.

One solution to this problem [Mitchell, 1978] is to maintain several G and S sets. One G set is consistent with all the training instances, another is consistent with all but one, another with all but two, etc. (and the same for the S set). When an inconsistency arises, the algorithm switches to G and S sets that are consistent with most, but not all, of the training examples. Maintaining multiple version spaces can be costly, however, and the S and G sets are typically very large. If we assume *bounded inconsistency*, i.e., that instances close to the target concept boundary are the most likely to be misclassified, then more efficient solutions are possible. Hirsh [1990] presents an algorithm that runs as follows. For each instance, we form a version space consistent with that instance plus other nearby instances (for some suitable definition of nearby). This version space is then intersected with the one created for all previous instances. We keep accepting instances until the version space is reduced to a small set of candidate concept descriptions. (Because of inconsistency, it is unlikely that the version space will converge to a singleton.) We then match each of the concept descriptions against the entire data set, and choose the one that classifies the instances most accurately.

Another problem with the candidate elimination algorithm is the learning of disjunctive concepts. Suppose we wanted to learn the concept of "European car," which, in our representation, means either a German, British, or Italian car. Given positive examples of each, the candidate elimination algorithm will generalize to cars of any *origin*. Given such a generalization, a negative instance (say, a Japanese car) will only cause an inconsistency of the type mentioned above.

Of course, we could simply extend the representation language to include disjunctions. Thus, the concept space would hold descriptions such as "Blue car of German or British origin" and "Italian sports car or German luxury car." This approach has two drawbacks. First, the concept space becomes much larger and specialization becomes intractable. Second, generalization can easily degenerate to the point where the S set contains simply one large disjunction of all positive instances. We must somehow force generalization while allowing for the introduction of disjunctive descriptions. Mitchell [1978] gives an iterative approach that involves several passes through the training data. On each pass, the algorithm builds a concept that covers the largest number of positive training instances without covering any negative training instances. At the end of the pass, the positive training instances covered by the new concept are removed from the

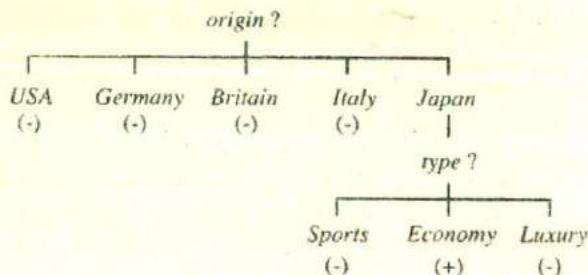


Figure 17.13: A Decision Tree

training set, and the new concept then becomes one disjunct in the eventual disjunctive concept description. When all positive training instances have been removed, we are left with a disjunctive concept that covers all of them without covering any negative instances.

There are a number of other complexities, including the way in which features interact with one another. For example, if the *origin* of a car is *Japan*, then the *manufacturer* cannot be *Chrysler*. The version space algorithm as described above makes no use of such information. Also in our example, it would be more natural to replace the *decade* slot with a continuously valued *year* field. We would have to change our procedures for updating the *S* and *G* sets to account for this kind of numerical data.

17.5.3 Decision Trees

A third approach to concept learning is the induction of *decision trees*, as exemplified by the ID3 program of Quinlan [1986]. ID3 uses a tree representation for concepts, such as the one shown in Figure 17.13. To classify a particular input, we start at the top of the tree and answer questions until we reach a leaf, where the classification is stored. Figure 17.13 represents the familiar concept "Japanese economy car." ID3 is a program that builds decision trees automatically, given positive and negative instances of a concept.⁴

ID3 uses an iterative method to build up decision trees, preferring simple trees over complex ones, on the theory that simple trees are more accurate classifiers of future inputs. It begins by choosing a random subset of the training examples. This subset is called the *window*. The algorithm builds a decision tree that correctly classifies all examples in the window. The tree is then tested on the training examples outside the window. If all the examples are classified correctly, the algorithm halts. Otherwise, it adds a number of training examples to the window and the process repeats. Empirical evidence indicates that the iterative strategy is more efficient than considering the whole training set at once.

⁴Actually, the decision tree representation is more general: Leaves can denote any of a number of classes, not just positive and negative.

So how does ID3 actually construct decision trees? Building a node means choosing some attribute to test. At a given point in the tree, some attributes will yield more information than others. For example, testing the attribute *color* is useless if the color of a car does not help us to classify it correctly. Ideally, an attribute will separate training instances into subsets whose members share a common label (e.g., positive or negative). In that case, branching is terminated, and the leaf nodes are labeled.

There are many variations on this basic algorithm. For example, when we add a test that has more than two branches, it is possible that one branch has no corresponding training instances. In that case, we can either leave the node unlabeled, or we can attempt to guess a label based on statistical properties of the set of instances being tested at that point in the tree. Noisy input is another issue. One way of handling noisy input is to avoid building new branches if the information gained is very slight. In other words, we do not want to overcomplicate the tree to account for isolated noisy instances. Another source of uncertainty is that attribute values may be unknown. For example a patient's medical record may be incomplete. One solution is to guess the correct branch to take; another solution is to build special "unknown" branches at each node during learning.

When the concept space is very large, decision tree learning algorithms run more quickly than their version space cousins. Also, disjunction is more straightforward. For example, we can easily modify Figure 17.13 to represent the disjunctive concept "American car or Japanese economy car," simply by changing one of the negative (-) leaf labels to positive (+). One drawback to the ID3 approach is that large, complex decision trees can be difficult for humans to understand, and so a decision tree system may have a hard time explaining the reasons for its classifications.

17.6 Explanation-Based Learning

The previous section illustrated how we can induce concept descriptions from positive and negative examples. Learning complex concepts using these procedures typically requires a substantial number of training instances. But people seem to be able to learn quite a bit from single examples. Consider a chess player who, as Black, has reached the position shown in Figure 17.14. The position is called a "fork" because the white knight attacks both the black king and the black queen. Black must move the king, thereby leaving the queen open to capture. From this single experience, Black is able to learn quite a bit about the fork trap: the idea is that if any piece x attacks both the opponent's king and another piece y , then piece y will be lost. We don't need to see dozens of positive and negative examples of fork positions in order to draw these conclusions. From just one experience, we can learn to avoid this trap in the future and perhaps to use it to our own advantage.

What makes such single-example learning possible? The answer, not surprisingly, is knowledge. The chess player has plenty of domain-specific knowledge that can be brought to bear, including the rules of chess and any previously acquired strategies. That knowledge can be used to identify the critical aspects of the training example. In the case of the fork, we know that the double simultaneous attack is important while the precise position and type of the attacking piece is not.

Much of the recent work in machine learning has moved away from the empirical, data-intensive approach described in the last section toward this more analytical.

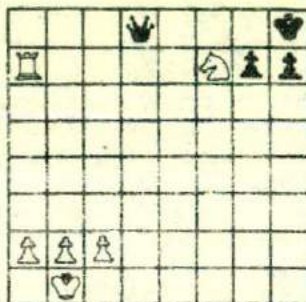


Figure 17.14: A Fork Position in Chess

knowledge-intensive approach. A number of independent studies led to the characterization of this approach as *explanation-based learning*. An EBL system attempts to learn from a single example x by explaining why x is an example of the target concept. The explanation is then generalized, and the system's performance is improved through the availability of this knowledge.

Mitchell *et al.* [1986] and DeJong and Mooney [1986] both describe general frameworks for EBL programs and give general learning algorithms. We can think of EBL programs as accepting the following as input:

- A Training Example—What the learning program “sees” in the world, e.g., the car of Figure 17.7
- A Goal Concept—A high-level description of what the program is supposed to learn
- An Operability Criterion—A description of which concepts are usable
- A Domain Theory—A set of rules that describe relationships between objects and actions in a domain

From this, EBL computes a *generalization* of the training example that is sufficient to describe the goal concept, and also satisfies the operability criterion.

Let's look more closely at this specification. The training example is a familiar input—it is the same thing as the example in the version space algorithm. The goal concept is also familiar, but in previous sections, we have viewed the goal concept as an output of the program, not an input. The assumption here is that the goal concept is not operational, just like the high-level card-playing advice described in Section 17.3. An EBL program seeks to operationalize the goal concept by expressing it in terms that a problem-solving program can understand. These terms are given by the operability criterion. In the chess example, the goal concept might be something like “bad position for Black,” and the operationalized concept would be a generalized description of situations similar to the training example, given in terms of pieces and their relative positions. The last input to an EBL program is a domain theory, in our case, the rules of

chess. Without such knowledge, it is impossible to come up with a correct generalization of the training example.

Explanation-based generalization (EBG) is an algorithm for EBL described in Mitchell *et al.* [1986]. It has two steps: (1) explain and (2) generalize. During the first step, the domain theory is used to prune away all the unimportant aspects of the training example with respect to the goal concept. What is left is an *explanation* of why the training example is an instance of the goal concept. This explanation is expressed in terms that satisfy the operability criterion. The next step is to generalize the explanation as far as possible while still describing the goal concept. Following our chess example, the first EBL step chooses to ignore White's pawns, king, and rook, and constructs an explanation consisting of White's knight, Black's king, and Black's queen, each in their specific positions. Operability is ensured: all chess-playing programs understand the basic concepts of piece and position. Next, the explanation is generalized. Using domain knowledge, we find that moving the pieces to a different part of the board is still bad for Black. We can also determine that other pieces besides knights and queens can participate in fork attacks.

In reality, current EBL methods run into difficulties in domains as complex as chess, so we will not pursue this example further. Instead, let's look at a simpler case. Consider the problem of learning the concept *Cup* [Mitchell *et al.*, 1986]. Unlike the arch-learning program of Section 17.5.1, we want to be able to generalize from a single example of a cup. Suppose the example is:

- Training Example:

$$\text{owner}(\text{Object23}, \text{Ralph}) \wedge \text{has-part}(\text{Object23}, \text{Concavity12}) \wedge \\ \text{is}(\text{Object23}, \text{Light}) \wedge \text{color}(\text{Object23}, \text{Brown}) \wedge \dots$$

Clearly, some of the features of *Object23* are more relevant to its being a cup than others. So far in this chapter, we have seen several methods for isolating relevant features. These methods all require many positive and negative examples. In EBL we instead rely on domain knowledge, such as:

- Domain Knowledge:

$$\text{is}(x, \text{Light}) \wedge \text{has-part}(x, y) \wedge \text{isa}(y, \text{Handle}) \rightarrow \text{liftable}(x) \\ \text{has-part}(x, y) \wedge \text{isa}(y, \text{Bottom}) \wedge \text{is}(y, \text{Flat}) \rightarrow \text{stable}(x) \\ \text{has-part}(x, y) \wedge \text{isa}(y, \text{Concavity}) \wedge \text{is}(y, \text{Upward-Pointing}) \rightarrow \text{open-vessel}(x)$$

We also need a goal concept to operationalize:

- Goal Concept: *Cup*
 x is a Cup iff x is *liftable*, *stable*, and *open-vessel*.
- Operability Criterion: Concept definition must be expressed in purely structural terms (e.g., *Light*, *Flat*, etc.).

Given a training example and a functional description, we want to build a general structural description of a cup. The first step is to explain why *Object23* is a cup. We do this by constructing a proof, as shown in Figure 17.15. Standard theorem-proving

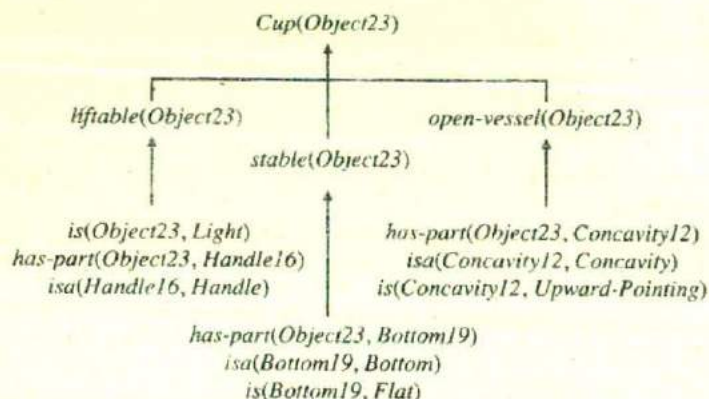


Figure 17.15: An Explanation

techniques can be used to find such a proof. Notice that the proof isolates the relevant features of the training example; nowhere in the proof do the predicates *owner* and *color* appear. The proof also serves as a basis for a valid generalization. If we gather up all the assumptions and replace constants with variables, we get the following description of a cup:

$$\begin{aligned}
 & \text{has-part}(x, y) \wedge \text{isa}(y, \text{Concavity}) \wedge \text{is}(y, \text{Upward-Pointing}) \wedge \\
 & \text{has-part}(x, z) \wedge \text{isa}(z, \text{Bottom}) \wedge \text{is}(z, \text{Flat}) \wedge \\
 & \text{has-part}(x, w) \wedge \text{isa}(w, \text{Handle}) \wedge \text{is}(x, \text{Light})
 \end{aligned}$$

This definition satisfies the operationality criterion and could be used by a robot to classify objects.

Simply replacing constants by variables worked in this example, but in some cases it is necessary to retain certain constants. To catch these cases, we must reprove the goal. This process, which we saw earlier in our discussion of learning in STRIPS, is called *goal regression*.

As we have seen, EBL depends strongly on a domain theory. Given such a theory, why are examples needed at all? We could have operationalized the goal concept *Cup* without reference to an example, since the domain theory contains all of the requisite information. The answer is that examples help to focus the learning on relevant operationalizations. Without an example cup, EBL is faced with the task of characterizing the entire range of objects that satisfy the goal concept. Most of these objects will never be encountered in the real world, and so the result will be overly general.

Providing a tractable domain theory is a difficult task. There is evidence that humans do not learn with very primitive relations. Instead, they create incomplete and

inconsistent domain theories. For example, returning to chess, such a theory might include concepts like "weak pawn structure." Getting EBL to work in ill-structured domain theories is an active area of research (see, e.g., Tadepalli [1989]).

EBL shares many features of all the learning methods described in earlier sections. Like concept learning, EBL begins with a positive example of some concept. As in learning by advice taking, the goal is to operationalize some piece of knowledge. And EBL techniques, like the techniques of chunking and macro-operators, are often used to improve the performance of problem-solving engines. The major difference between EBL and other learning methods is that EBL programs are built to take advantage of domain knowledge. Since learning is just another kind of problem solving, it should come as no surprise that there is leverage to be found in knowledge.

17.7 Discovery

Learning is the process by which one entity acquires knowledge. Usually that knowledge is already possessed by some number of other entities who may serve as teachers. *Discovery* is a restricted form of learning in which one entity acquires knowledge without the help of a teacher.⁵ In this section, we look at three types of automated discovery systems.

17.7.1 AM: Theory-Driven Discovery

Discovery is certainly learning. But it is also, perhaps more clearly than other kinds of learning, problem solving. Suppose that we want to build a program to discover things, for example, in mathematics. We expect that such a program would have to rely heavily on the problem-solving techniques we have discussed. In fact, one such program was written by Lenat [1977; 1982]. It was called AM, and it worked from a few basic concepts of set theory to discover a good deal of standard number theory.

AM exploited a variety of general-purpose AI techniques. It used a frame system to represent mathematical concepts. One of the major activities of AM is to create new concepts and fill in their slots. An example of an AM concept is shown in Figure 17.16. AM also uses heuristic search, guided by a set of 250 heuristic rules representing hints about activities that are likely to lead to "interesting" discoveries. Examples of the kind of heuristics AM used are shown in Figure 17.17. Generate-and-test is used to form hypotheses on the basis of a small number of examples and then to test the hypotheses on a larger set to see if they still appear to hold. Finally, an agenda controls the entire discovery process. When the heuristics suggest a task, it is placed on a central agenda, along with the reason that it was suggested and the strength with which it was suggested. AM operates in cycles, each time choosing the most promising task from the agenda and performing it.

In one run, AM discovered the concept of prime numbers. How did it do that? Having stumbled onto the natural numbers, AM explored operations such as addition, multiplication, and their inverses. It created the concept of divisibility and noticed that some numbers had very few divisors. AM has a built-in heuristic that tells it to explore

⁵Sometimes, there is no one in the world who has the knowledge we seek. In that case, the kind of action we must take is called *scientific discovery*.

name : Prime-Numbers
 definitions :
 origin : Number-of-divisors-of(x) = 2
 predicate-calculus : Prime(x) \leftrightarrow ($\forall z$)($z \mid x \Rightarrow (z = 1 \otimes z = x)$)
 iterative : (for $x > 1$): For i from 2 to \sqrt{x} , $i \nmid x$
 examples : 2, 3, 5, 7, 11, 13, 17
 boundary : 2, 3
 boundary-failures : 0, 1
 failures : 12
 generalizations : Number, numbers with an even number of divisors
 specializations : Odd primes, prime pairs, prime uniquely addables
 conjecs : Unique factorization, Goldbach's conjecture, extrema of number-of-divisors-of
 intus : A metaphor to the effect that primes are the building blocks of all numbers
 analogies :
 Maximally divisible numbers are converse extremes of number-of-divisors-of
 Factor a nonsimple group into simple groups
 interest : Conjectures tying primes to times, to divisors of, to related operations
 worth : 800

Figure 17.16: An AM Concept: Prime Number

- If f is a function from A to B and B is ordered, then consider the elements of A that are mapped into extremal elements of B . Create a new concept representing this subset of A .
- If some (but not most) examples of some concept X are also examples of another concept Y , create a new concept representing the intersection of X and Y .
- If very few examples of a concept X are found, then add to the agenda the task of finding a generalization of X .

Figure 17.17: Some AM Heuristics

extreme cases. It attempted to list all numbers with zero divisors (finding none), one divisor (finding one: 1), and two divisors. AM was instructed to call the last concept "primes." Before pursuing this concept, AM went on to list numbers with three divisors, such as 49. AM tried to relate this property with other properties of 49, such as its being odd and a perfect square. AM generated other odd numbers and other perfect squares to test its hypotheses. A side effect of determining the equivalence of perfect squares with numbers with three divisors was to boost the "interestingness" rating of the divisor concept. This led AM to investigate ways in which a number could be broken down into factors. AM then noticed that there was only one way to break a number down into prime factors (known as the Unique Factorization Theorem).

Since breaking down numbers into multiplicative components turned out to be interesting, AM decided, by analogy, to pursue additive components as well. It made several uninteresting conjectures, such as that every number could be expressed as a sum of 1's. It also found more interesting phenomena, such as that many numbers were expressible as the sum of two primes. By listing cases, AM determined that all

even numbers greater than 2 seemed to have this property. This conjecture, known as Goldbach's Conjecture, is widely believed to be true, but a proof of it has yet to be found in mathematics.

AM contains a great many general-purpose heuristics such as the ones it used in this example. Often different heuristics point in the same place. For example, while AM discovered prime numbers using a heuristic that involved looking at extreme cases, another way to derive prime numbers is to use the following two rules:

- If there is a strong analogy between A and B but there is a conjecture about A that does not hold for all elements of B, define a new concept that includes the elements of B for which it does hold.
- If there is a set whose complement is much rarer than itself, then create a new concept representing the complement.

There is a strong analogy between addition and multiplication of natural numbers. But that analogy breaks down when we observe that all natural numbers greater than 1 can be expressed as the sum of two smaller natural numbers (excluding the identity). This is not true for multiplication. So the first heuristic described above suggests the creation of a new concept representing the set of composite numbers. Then the second heuristic suggests creating a concept representing the complement of that, namely the set of prime numbers.

Two major questions came out of the work on AM. One question was: "Why was AM ever turned off?" That is, why didn't AM simply keep discovering new interesting facts about numbers, possibly facts unknown to human mathematics? Lenat [1983b] contends that AM's performance was limited by the static nature of its heuristics. As the program progressed, the concepts with which it was working evolved away from the initial ones, while the heuristics that were available to work on those concepts stayed the same. To remedy this problem, it was suggested that heuristics be treated as full-fledged concepts that could be created and modified by the same sorts of processes (such as generalization, specialization, and analogy) as are concepts in the task domain. In other words, AM would run in discovery mode in the domain of "Heuristics," the study of heuristics themselves, as well as in the domain of number theory. An extension of AM called EURISKO [Lenat, 1983a] was designed with this goal in mind.

The other question was: "Why did AM work as well as it did?" One source of power for AM was its huge collection of heuristics about what constitute interesting things. But AM had another less obvious source of power, namely, the natural relationship between number theoretical concepts and their compact representations in AM [Lenat and Brown, 1983]. AM worked by syntactically mutating old concept definitions—stored essentially as short LISP programs—in the hopes of finding new, interesting concepts. It turns out that a mutation in a small LISP program very likely results in another well-formed, meaningful LISP program. This accounts for AM's ability to generate so many novel concepts. But while humans interpret AM as exploring number theory, it was actually exploring the space of small LISP programs. AM succeeded in large part because of this intimate relationship between number theory and LISP programs. When AM and EURISKO were applied to other domains, including the study of heuristics themselves, problems arose. Concepts in these domains were larger and more complex than number theory concepts, and the syntax of the representation

n	T	p	V	pV	pV/T	pV/nT
1	300	100	24.96			
1	300	200	12.48			
1	300	300	8.32	2496		
1	310			2579.2		
1	320			2662.4	8.32	
2	320				16.64	
3	320				24.96	8.32

Figure 17.18: BACON Discovering the Ideal Gas Law

language no longer closely mirrored the semantics of the domain. As a result, syntactic mutation of a concept definition almost always resulted in an ill-formed or useless concept, severely hampering the discovery procedure.

Perhaps the moral of AM is that learning is a tricky business. We must be careful how we interpret what our AI programs are doing [Ritchie and Hanna, 1984]. AM had an implicit *bias* toward learning concepts in number theory. Only after that bias was explicitly recognized was it possible to understand why AM performed well in one domain and poorly in another.

17.7.2 BACON: Data-Driven Discovery

AM showed how discovery might occur in a theoretical setting. Empirical scientists see things somewhat differently. They are confronted with data from the world and must make sense of it. They make hypotheses, and in order to validate them, they design and execute experiments. Scientific discovery has inspired a number of computer models. Langley *et al.* [1981a] present a model of data-driven scientific discovery that has been implemented as a program called BACON, named after Sir Francis Bacon, an early philosopher of science.

BACON begins with a set of variables for a problem. For example, in the study of the behavior of gases, some variables are p , the pressure on the gas, V , the volume of the gas, n , the amount of gas in moles, and T , the temperature of the gas. Physicists have long known a law, called the *ideal gas law*, that relates these variables. BACON is able to derive this law on its own. First, BACON holds the variables n and T constant, performing experiments at different pressures p_1 , p_2 , and p_3 . BACON notices that as the pressure increases, the volume V decreases. Therefore, it creates a theoretical term pV . This term is constant. BACON systematically moves on to vary the other variables. It tries an experiment with different values of T , and finds that pV changes. The two terms are linearly related with an intercept of 0, so BACON creates a new term pV/T . Finally, BACON varies the term n and finds another linear relation between n and pV/T . For all values of n , p , V , and T , $pV/nT = 8.32$. This is, in fact, the ideal gas law. Figure 17.18 shows BACON's reasoning in a tabular format.

BACON has been used to discover a wide variety of scientific laws, such as Kepler's third law, Ohm's law, the conservation of momentum, and Joule's law. The heuristics BACON uses to discover the ideal gas law include noting constancies, finding linear

relations, and defining theoretical terms. Other heuristics allow BACON to postulate intrinsic properties of objects and to reason by analogy. For example, if BACON finds a regularity in one set of parameters, it will attempt to generate the same regularity in a similar set of parameters. Since BACON's discovery procedure is state-space search, these heuristics allow it to reach solutions while visiting only a small portion of the search space. In the gas example, BACON comes up with the ideal gas law using a minimal number of experiments.

A better understanding of the science of scientific discovery may lead one day to programs that display true creativity. Much more work must be done in areas of science that BACON does not model, such as determining what data to gather, choosing (or creating) instruments to measure the data, and using analogies to previously understood phenomena. For a thorough discussion of scientific discovery programs, see Langley *et al.* [1987].

17.7.3 Clustering

A third type of discovery, called *clustering*, is very similar to induction, as we described it in Section 17.5. In inductive learning, a program learns to classify objects based on the labelings provided by a teacher. In clustering, no class labelings are provided. The program must discover for itself the natural classes that exist for the objects, in addition to a method for classifying instances.

AUTOCLASS [Cheeseman *et al.*, 1988] is one program that accepts a number of training cases and hypothesizes a set of classes. For any given case, the program provides a set of probabilities that predict into which class(es) the case is likely to fall. In one application, AUTOCLASS found meaningful new classes of stars from their infrared spectral data. This was an instance of true discovery by computer, since the facts it discovered were previously unknown to astronomy. AUTOCLASS uses statistical Bayesian reasoning of the type discussed in Chapter 8.

17.8 Analogy

Analogy is a powerful inference tool. Our language and reasoning are laden with analogies. Consider the following sentences:

- Last month, the stock market was a roller coaster.
- Bill is like a fire engine.
- Problems in electromagnetism are just like problems in fluid flow.

Underlying each of these examples is a complicated mapping between what appear to be dissimilar concepts. For example, to understand the first sentence above, it is necessary to do two things: (1) pick out one key property of a roller coaster, namely that it travels up and down rapidly and (2) realize that physical travel is itself an analogy for numerical fluctuations (in stock prices). This is no easy trick. The space of possible analogies is very large. We do not want to entertain possibilities such as "the stock market is like a roller coaster because it is made of metal."

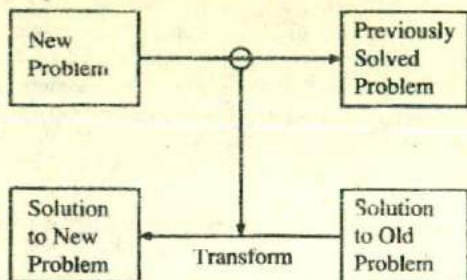


Figure 17.19: Transformational Analogy

Lakoff and Johnson [1980] make the case that everyday language is filled with such analogies and metaphors. An AI program that is unable to grasp analogy will be difficult to talk to and, consequently, difficult to teach. Thus, analogical reasoning is an important factor in learning by advice taking. It is also important to learning in problem solving.

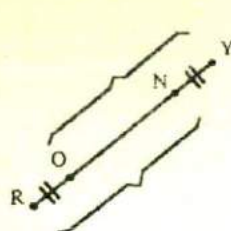
Humans often solve problems by making analogies to things they already understand how to do. This process is more complex than storing macro-operators (as discussed in Section 17.4.2) because the old problem might be quite different from the new problem on the surface. The difficulty comes in determining what things are similar and what things are not. Two methods of analogical problem solving that have been studied in AI are *transformational* and *derivational* analogy.

17.8.1 Transformational Analogy

Suppose you are asked to prove a theorem in plane geometry. You might look for a previous theorem that is very similar and “copy” its proof, making substitutions when necessary. The idea is to transform a solution to a previous problem into a solution for the current problem. Figure 17.19 shows this process.

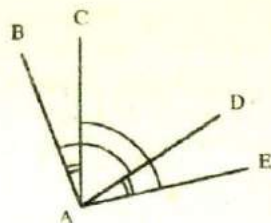
An example of transformational analogy is shown in Figure 17.20 [Anderson and Kline, 1979]. The program has seen proofs about points and line segments; for example, it knows a proof that the line segment RN is exactly as long as the line segment OY, given that RO is exactly as long as NY. The program is now asked to prove a theorem about angles, namely that the angle BD is equivalent to the angle CE, given that angles BC and DE are equivalent. The proof about line segments is retrieved and transformed into a proof about angles by substituting the notion of line for point, angle for line segment, AB for R, AC for O, AD for N, and AE for Y.

Carbonell [1983] describes one method for transforming old solutions into new solutions. Whole solutions are viewed as states in a problem space called *T-space*. *T-operators* prescribe the methods of transforming solutions (states) into other solutions. Reasoning by analogy becomes search in *T-space*: starting with an old solution, we use means-ends analysis or some other method to find a solution to the current problem.



Old Proof:

$RO = NY$ (given)
 $ON = ON$ (reflexive)
 $RO + ON = ON + NY$ (additive)
 $RN = OY$ (transitive)



New Proof:

$BAC = DAE$
 $CAD = CAD$
 $BAC + CAD = CAD + DAE$
 $BAD = CAE$

Figure 17.20: Solving a Problem by Transformational Analogy

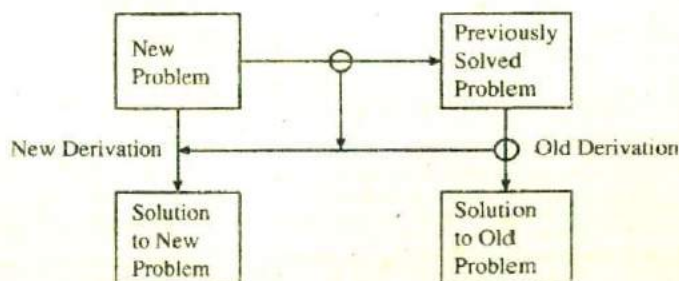


Figure 17.21: Derivational Analogy

17.8.2 Derivational Analogy

Notice that transformational analogy does not look at *how* the old problem was solved: it only looks at the final solution. Often the twists and turns involved in solving an old problem are relevant to solving a new problem. The detailed history of a problem-solving episode is called its *derivation*. Analogical reasoning that takes these histories into account is called *derivational analogy* (see Figure 17.21).

Carbonell [1986] claims that derivational analogy is a necessary component in the transfer of skills in complex domains. For example, suppose you have coded an efficient sorting routine in Pascal, and then you are asked to recode the routine in LISP. A line-by-line translation is not appropriate, but you will reuse the major structural and control decisions you made when you constructed the Pascal program. One way to model this behavior is to have a problem-solver "replay" the previous derivation and modify it when necessary. If the original reasons and assumptions for a step's existence still hold in the

new problem, the step is copied over. If some assumption is no longer valid, another assumption must be found. If one cannot be found, then we can try to find justification for some alternative stored in the derivation of the original problem. Or perhaps we can try some step marked as leading to search failure in the original derivation, if the reasons for failure conditions are not valid in the current derivation.

Analogy in problem solving is a very open area of research. For a survey of recent work, see Hall [1989].

17.9 Formal Learning Theory

Like many other AI problems, learning has attracted the attention of mathematicians and theoretical computer scientists. Inductive learning in particular has received considerable attention. Valiant [1984] describes a "theory of the learnable" which classifies problems by how difficult they are to learn. Formally, a device learns a concept if it can, given positive and negative examples, produce an algorithm that will classify future examples correctly with probability $1/h$. The complexity of learning a concept is a function of three factors: the error tolerance (h), the number of binary features present in the examples (t), and the size of the rule necessary to make the discrimination (f). If the number of training examples required is polynomial in h , t , and f , then the concept is said to be *learnable*.

Some interesting results have been demonstrated for concept learning. Consider the problem of learning conjunctive feature descriptions. For example, from the list of positive and negative examples of elephants shown in Figure 17.22, we want to induce the description "gray, mammal, large." It has been shown that in conjunctive learning the number of randomly chosen training examples is proportional to the logarithm of the total number of features [Haussler, 1988; Littlestone, 1988].⁶ Since very few training examples are needed to solve this induction problem, it is called *learnable*. Even if we restrict the learner to *positive* examples only, conjunctive learning can be achieved when the number of examples is linearly proportional to the number of attributes [Ehrenfeucht *et al.*, 1989]. Learning from positive examples only is a phenomenon not modeled by least-commitment inductive techniques such as version spaces. The introduction of the error tolerance h makes this possible: After all, even if all the elephants in our training set are gray, we may later encounter a genuine elephant that happens to be white. Fortunately, we can extend the size of our randomly sampled training set to ensure that the probability of misclassifying an elephant as something else (such as a polar bear) is an arbitrarily small $1/h$.

Formal techniques have been applied to a number of other learning problems. For example, given positive and negative examples of strings in some regular language, can we efficiently induce the finite automaton that produces all and only the strings in that language? The answer is no; an exponential number of computational steps is required [Kearns and Valiant, 1989].⁷ However, if we allow the learner to make specific queries (e.g., "Is string x in the language?"), then the problem is learnable [Angluin, 1987].

⁶However, the number of examples must be *linear* in the number of *relevant* attributes, i.e., the number of attributes that appear in the learned conjunction.

⁷The proof of this result rests on some unproven hypotheses about the complexity of certain number theoretic functions.

gray?	mammal?	large?	vegetarian?	wild?		
+	+	+	+	+	+	(Elephant)
+	+	+	-	+	+	(Elephant)
+	+	-	+	+	-	(Mouse)
-	+	+	+	+	-	(Giraffe)
+	-	+	-	+	-	(Dinosaur)
+	+	+	+	-	+	(Elephant)

Figure 17.22: Six Positive and Negative Examples of the Concept *Elephant*

It is difficult to tell how such mathematical studies of learning will affect the ways in which we solve AI problems in practice. After all, people are able to solve many exponentially hard problems by using knowledge to constrain the space of possible solutions. Perhaps mathematical theory will one day be used to quantify the use of such knowledge, but this prospect seems far off. For a critique of formal learning theory as well as some of the inductive techniques described in Section 17.5, see Amsterdam [1988].

17.10 Neural Net Learning and Genetic Learning

The very first efforts in machine learning tried to mimic animal learning at a neural level. These efforts were quite different from the symbolic manipulation methods we have seen so far in this chapter. Collections of idealized neurons were presented with stimuli and prodded into changing their behavior via forms of reward and punishment. Researchers hoped that by imitating the learning mechanisms of animals, they might build learning machines from very simple parts. Such hopes proved elusive. However, the field of neural network learning has seen a resurgence in recent years, partly as a result of the discovery of powerful new learning algorithms. Chapter 18 describes these algorithms in detail.

While neural network models are based on a computational "brain metaphor," a number of other learning techniques make use of a metaphor based on evolution. In this work, learning occurs through a selection process that begins with a large population of random programs. Learning algorithms inspired by evolution are called *genetic algorithms* [Holland, 1975; de Jong, 1988; Goldberg, 1989]

17.11 Summary

The most important thing to conclude from our study of automated learning is that learning itself is a problem-solving process. We can cast various learning strategies in terms of the methods of Chapters 2 and 3.

- Learning by taking advice
 - Initial state: high-level advice

- Final state: an operational rule
- Operators: unfolding definitions, case analysis, matching, etc.
- Learning from examples
 - Initial state: collection of positive and negative examples
 - Final state: concept description
 - Search algorithms: candidate elimination, induction of decision trees
- Learning in problem solving
 - Initial state: solution traces to example problems
 - Final state: new heuristics for solving new problems efficiently
 - Heuristics for search: generalization, explanation-based learning, utility considerations
- Discovery
 - Initial state: some environment
 - Final state: unknown
 - Heuristics for search: interestingness, analogy, etc.

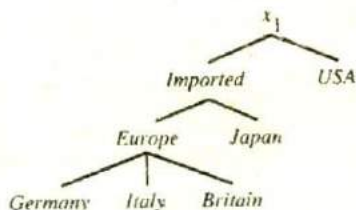
A learning machine is the dream system of AI. As we have seen in previous chapters, the key to intelligent behavior is having a lot of knowledge. Getting all of that knowledge into a computer is a staggering task. One hope of sidestepping the task is to let computers acquire knowledge independently, as people do. We do not yet have programs that can extend themselves indefinitely. But we have discovered some of the reasons for our failure to create such systems. If we look at actual learning programs, we find that the more knowledge a program starts with, the more it can learn. This finding is satisfying, in the sense that it corroborates our other discoveries about the power of knowledge. But it is also unpleasant, because it seems that fully self-extending systems are, for the present, still out of reach.

Research in machine learning has gone through several cycles of popularity. Timing is always an important consideration. A learning program needs to acquire new knowledge and new problem-solving abilities, but knowledge and problem-solving are topics still under intensive study. If we do not understand the nature of the thing we want to learn, learning is difficult. Not surprisingly, the most successful learning programs operate in fairly well-understood areas (like planning), and not in less well-understood areas (like natural language understanding).

17.12 Exercises

1. Would it be reasonable to apply Samuel's rote-learning procedure to chess? Why (not)?

- Implement the candidate elimination algorithm for version spaces. Choose a concept space with several features (for example, the space of books, computers, animals, etc.) Pick a concept and demonstrate learning by presenting positive and negative examples of the concept.
- In Section 17.5.2, the concept "Japanese economy car" was learned through the presentation of five positive and negative examples. Give a sequence of *four* examples that accomplishes the same goal. In general, what properties of a positive example make it most useful? What makes a negative example most useful?
- Recall the problem of learning disjunctive concepts in version spaces. We discussed learning a concept like "European car," where a European car was defined as a car whose *origin* was either *Germany*, *Italy*, or *Britain*. Suppose we expand the number of discrete values the slot *origin* might take to include the values *Europe* and *Imported*. Suppose further that we have the following *isa* hierarchy at our disposal:



The diagram reflects facts such as "Japanese cars are a subset of imported cars" and "Italian cars are a subset of European cars." How could we modify the candidate elimination algorithm to take advantage of this knowledge? Propose new methods of updating the sets G and S that would allow us to learn the concept "European car" in one pass through a set of adequate training examples.

- AM exploited a set of 250 heuristics designed to guide AM's behavior toward interesting mathematical concepts. A classic work by Polya [1957] describes a set of heuristics for solving mathematical problems. Unfortunately, Polya's heuristics are not specified in enough detail to make them implementable in a program. In particular, they lack precise descriptions of the situations in which they are appropriate (i.e., the left sides if they are viewed as productions). Examine some of Polya's rules and refine them so that they could be implemented in a problem-solving program with a structure similar to AM's.
- Consider the problem of building a program to learn a grammar for a language such as English. Assume that such a program would be provided, as input, with a set of pairs, each consisting of a sentence and a representation of the meaning of the sentence. This is analogous to the experience of a child who hears a sentence and sees something at the same time. How could such a program be built using the techniques discussed in this chapter?

Chapter 18

Connectionist Models

In our quest to build intelligent machines, we have but one naturally occurring model: the human brain. One obvious idea for AI, then, is to simulate the functioning of the brain directly on a computer. Indeed, the idea of building an intelligent machine out of artificial neurons has been around for quite some time. Some early results on brainlike mechanisms were achieved by McCulloch and Pitts [1943], and other researchers pursued this notion through the next two decades, e.g., Ashby [1952], Minsky [1954], Minsky and Selfridge [1961], Block [1962], and Rosenblatt [1962]. Research in neural networks came to virtual halt in the 1970s, however, when the networks under study were shown to be very weak computationally. Recently, there has been a resurgence of interest in neural networks. There are several reasons for this, including the appearance of faster digital computers on which to simulate larger networks, the interest in building massively parallel computers, and, most important, the discovery of new neural network architectures and powerful learning algorithms.

The new neural network architectures have been dubbed "connectionist" architectures. For the most part, these architectures are not meant to duplicate the operation of the human brain, but rather to receive inspiration from known facts about how the brain works. They are characterized by having:

- A large number of very simple neuronlike processing elements.
- A large number of weighted connections between the elements. The weights on the connections encode the knowledge of a network.
- Highly parallel, distributed control.
- An emphasis on learning internal representations automatically.

Connectionist researchers conjecture that thinking about computation in terms of the "brain metaphor" rather than the "digital computer metaphor" will lead to insights into the nature of intelligent behavior.

Computers are capable of amazing feats. They can effortlessly store vast quantities of information. Their circuits operate in nanoseconds. They can perform extensive arithmetic calculations without error. Humans cannot approach these capabilities. On the other hand, humans routinely perform "simple" tasks such as walking, talking, and

commonsense reasoning. Current AI systems cannot do any of these things better than humans can. Why not? Perhaps the structure of the brain is somehow suited to these tasks and not suited to tasks such as high-speed arithmetic calculation. Working under constraints similar to those of the brain may make traditional computation more difficult, but it may lead to solutions to AI problems that would otherwise be overlooked.

What constraints, then, does the brain offer us? First of all, individual neurons are extremely slow devices when compared to their counterparts in digital computers. Neurons operate in the millisecond range, an eternity to a VLSI designer. Yet, humans can perform extremely complex tasks, such as interpreting a visual scene or understanding a sentence, in just a tenth of a second. In other words, we do in about a hundred steps what current computers cannot do in 10 million steps. How can this be possible? Unlike a conventional computer, the brain contains a huge number of processing elements that act in parallel. This suggests that in our search for solutions, we should look for massively parallel algorithms that require no more than 100 time steps [Feldman and Ballard, 1985].

Also, neurons are failure-prone devices. They are constantly dying (you have certainly lost a few since you began reading this chapter), and their firing patterns are irregular. Components in digital computers, on the other hand, must operate perfectly. Why? Such components store bits of information that are available nowhere else in the computer: the failure of one component means a loss of information. Suppose that we built AI programs that were not sensitive to the failure of a few components, perhaps by using redundancy and distributing information across a wide range of components? This would open up the possibility of very large-scale implementations. With current technology, it is far easier to build a *billion-component* integrated circuit in which 95 percent of the components work correctly than it is to build a *million-component* machine that functions perfectly [Fahlman and Hinton, 1987].

Another thing people seem to be able to do better than computers is handle fuzzy situations. We have very large memories of visual, auditory, and problem-solving episodes, and one key operation in solving new problems is finding closest matches to old situations. Approximate matching is something brain-style models seem to be good at, because of the diffuse and fluid way in which knowledge is represented.

The idea behind connectionism, then, is that we may see significant advances in AI if we approach problems from the point of view of brain-style computation. Connectionist AI is quite different from the symbolic approach covered in the other chapters of this book. At the end of this chapter, we discuss the relationship between the two approaches.

18.1 Introduction: Hopfield Networks

The history of AI is curious. The first problems attacked by AI researchers were problems such as chess and theorem proving, because they were thought to require the essence of intelligence. Vision and language understanding—processes easily mastered by five-year olds—were not thought to be difficult. These days, we have expert chess programs and expert medical diagnosis programs, but no programs that can match the basic perceptual skills of a child. Neural network researchers contend that there is a basic mismatch between standard computer information processing technology and the technology used by the brain.

In addition to these perceptual tasks, AI is just starting to grapple with the fundamental problems of memory and commonsense reasoning. Computers are notorious for their lack of common sense. Many people believe that common sense derives from our massive store of knowledge and, more important, our ability to access relevant knowledge quickly, effortlessly, and at the right time.

When we read the description "gray, large, mammal," we automatically think of elephants and their associated features. We access our memories *by content*. In traditional implementations, access by content involves expensive searching and matching procedures. Massively parallel networks suggest a more efficient method.

Hopfield [1982] introduced a neural network that he proposed as a theory of memory. A Hopfield network has the following interesting features:

- **Distributed Representation**—A memory is stored as a pattern of activation across a set of processing elements. Furthermore, memories can be superimposed on one another; different memories are represented by different patterns over the *same* set of processing elements.
- **Distributed, Asynchronous Control**—Each processing element makes decisions based only on its own local situation. All these local actions add up to a global solution.
- **Content-Addressable Memory**—A number of patterns can be stored in a network. To retrieve a pattern, we need only specify a portion of it. The network automatically finds the closest match.
- **Fault Tolerance**—If a few processing elements misbehave or fail completely, the network will still function properly.

How are these features achieved? A simple Hopfield net is shown in Figure 18.1. Processing elements, or *units*, are always in one of two states, active or inactive. In the figure, units colored black are active and units colored white are inactive. Units are connected to each other with weighted, symmetric connections. A positively weighted connection indicates that the two units tend to activate each other. A negative connection allows an active unit to deactivate a neighboring unit.

The network operates as follows. A random unit is chosen. If any of its neighbors are active, the unit computes the sum of the weights on the connections to those active neighbors. If the sum is positive, the unit becomes active, otherwise it becomes inactive. Another random unit is chosen, and the process repeats until the network reaches a stable state, i.e., until no more units can change state. This process is called *parallel relaxation*. If the network starts in the state shown in Figure 18.1, the unit in the lower left corner will tend to activate the unit above it. This unit, in turn, will attempt to activate the unit above it, but the inhibitory connection from the upper-right unit will foil this attempt, and so on.

This network has only four distinct stable states, which are shown in Figure 18.2. Given any initial state, the network will necessarily settle into one of these four configurations.¹ The network can be thought of as "storing" the patterns in Figure 18.2. Hopfield's major contribution was to show that given any set of weights and any initial

¹The stable state in which all units are inactive can only be reached if it is also the initial state.

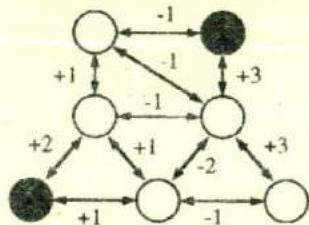


Figure 18.1: A Simple Hopfield Network

state, his parallel relaxation algorithm would eventually steer the network into a stable state. There can be no divergence or oscillation.

The network can be used as a content-addressable memory by setting the activities of the units to correspond to a partial pattern. To retrieve a pattern, we need only supply a portion of it. The network will then settle into the stable state that best matches the partial pattern. An example is shown in Figure 18.3.

Parallel relaxation is nothing more than search, albeit of a different style than the search described in the early chapters of this book. It is useful to think of the various states of a network as forming a search space, as in Figure 18.4. A randomly chosen state will transform itself ultimately into one of the *local minima*, namely the nearest stable state. This is how we get the content-addressable behavior.² We also get error-correcting behavior. Suppose we read the description, "gray, large, fish, eats plankton." We imagine a whale, even though we know that a whale is a mammal, not a fish. Even if the initial state contains inconsistencies, a Hopfield network will settle into the solution that violates the fewest constraints offered by the inputs. Traditional match-and-retrieve procedures are less forgiving.

Now, suppose a unit occasionally fails, say, by becoming active or inactive when it should not. This causes no major problem: surrounding units will quickly set it straight again. It would take the unlikely concerted effort of many errant units to push the network into the wrong stable state. In networks of thousands of more highly interconnected units, such fault tolerance is even more apparent—units and connections can disappear completely without adversely affecting the overall behavior of the network.

So parallel networks of simple elements can compute interesting things. The next important question is: What is the relationship between the weights on the network's connections and the local minima it settles into? In other words, if the weights encode the knowledge of a particular network, then how is that knowledge acquired? In Chapter 17 we saw several ways to acquire symbolic structures and descriptions. Such acquisition was quite difficult. One feature of connectionist architectures is that their method of representation (namely, real-valued connection weights) lends itself very nicely to

²In Figure 18.4, state B is depicted as being lower than state A because fewer constraints are violated. A constraint is violated, for example, when two active units are connected by a negatively weighted connection.

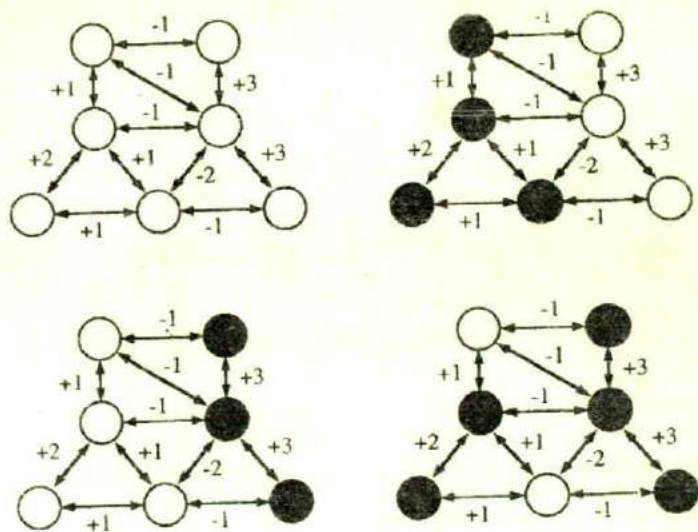


Figure 18.2: The Four Stable States of a Particular Hopfield Net

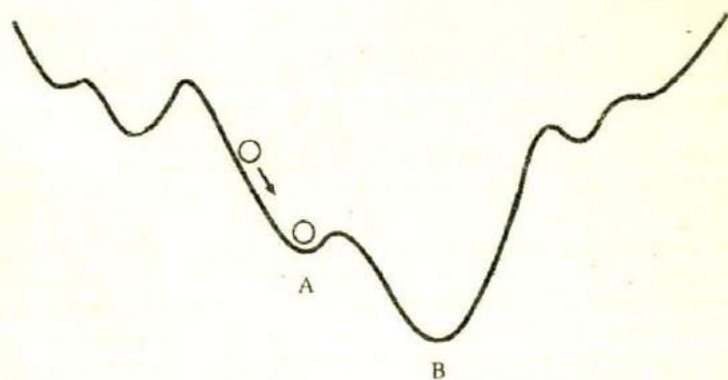


Figure 18.3: A Hopfield Net as a Model of Content-Addressable Memory

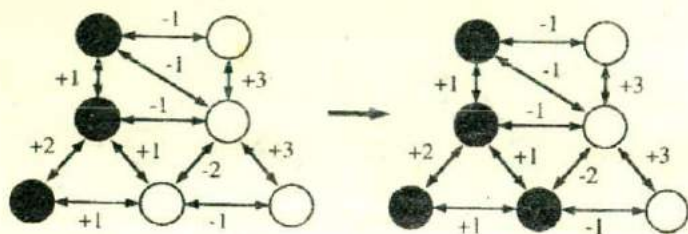


Figure 18.4: A Simplified View of What a Hopfield Net Computes

automatic learning.

In the next section, we look closely at learning in several neural network models, including perceptrons, backpropagation networks, and Boltzmann machines, a variation of Hopfield networks. After this, we investigate some applications of connectionism. Then we see how networks with feedback can deal with temporal processes and how distributed representations can be made efficient.

18.2 Learning in Neural Networks

18.2.1 Perceptrons

The *perceptron*, an invention of Rosenblatt [1962], was one of the earliest neural network models. A perceptron models a neuron by taking a weighted sum of its inputs and sending the output 1 if the sum is greater than some adjustable threshold value (otherwise it sends 0). Figure 18.5 shows the device. Notice that in a perceptron, unlike a Hopfield network, connections are unidirectional.

The inputs (x_1, x_2, \dots, x_n) and connection weights (w_1, w_2, \dots, w_n) in the figure are typically real values, both positive and negative. If the presence of some feature x_i tends to cause the perceptron to fire, the weight w_i will be positive; if the feature x_i inhibits the perceptron, the weight w_i will be negative. The perceptron itself consists of the weights, the summation processor, and the adjustable threshold processor. Learning is a process of modifying the values of the weights and the threshold. It is convenient to implement the threshold as just another weight w_0 , as in Figure 18.6. This weight can be thought of as the propensity of the perceptron to fire irrespective of its inputs. The perceptron of Figure 18.6 fires if the weighted sum is greater than zero.

A perceptron computes a binary function of its input. Several perceptrons can be combined to compute more complex functions, as shown in Figure 18.7.

Such a group of perceptrons can be trained on sample input-output pairs until it learns to compute the correct function. The amazing property of perceptron learning

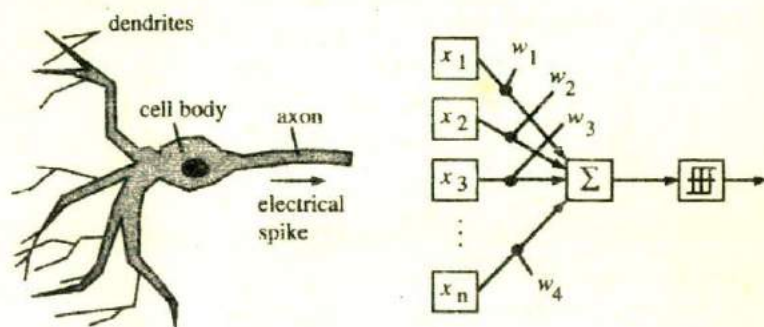


Figure 18.5: A Neuron and a Perceptron

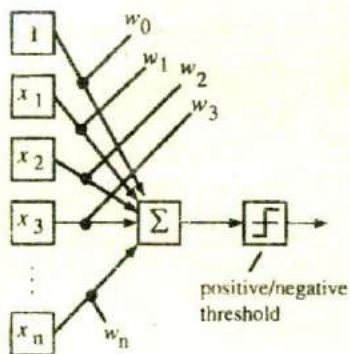


Figure 18.6: Perceptron with Adjustable Threshold Implemented as Additional Weight:

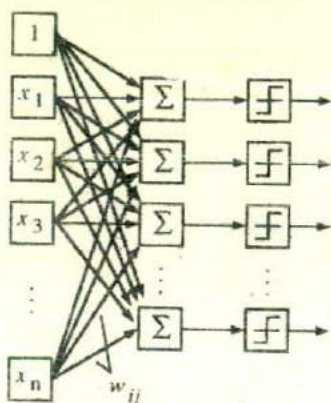


Figure 18.7: A Perceptron with Many Inputs and Many Outputs

is this: Whatever a perceptron can compute, it can *learn* to compute! We demonstrate this in a moment. At the time perceptrons were invented, many people speculated that intelligent systems could be constructed out of perceptrons (see Figure 18.8).

Since the perceptrons of Figure 18.7 are independent of one another, they can be separately trained. So let us concentrate on what a single perceptron can learn to do. Consider the pattern classification problem shown in Figure 18.9. This problem is *linearly separable*, because we can draw a line that separates one class from another. Given values for x_1 and x_2 , we want to train a perceptron to output 1 if it thinks the input belongs to the class of white dots and 0 if it thinks the input belongs to the class of black dots. Pattern classification is very similar to *concept learning*, which was discussed in Chapter 17. We have no explicit rule to guide us; we must induce a rule from a set of training instances. We now see how perceptrons can learn to solve such problems.

First, it is necessary to take a close look at what the perceptron computes. Let \bar{x} be an input vector (x_1, x_2, \dots, x_n) . Notice that the weighted summation function $g(x)$ and the output function $o(x)$ can be defined as:

$$g(x) = \sum_{i=0}^n w_i x_i$$

$$o(x) = \begin{cases} 1 & \text{if } g(x) > 0 \\ 0 & \text{if } g(x) < 0 \end{cases}$$

Consider the case where we have only two inputs (as in Figure 18.9). Then

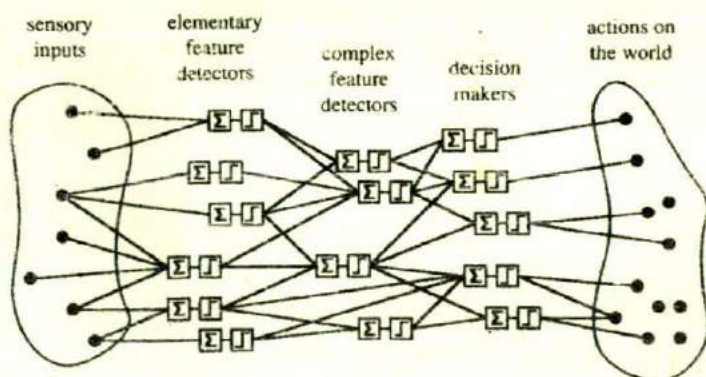


Figure 18.8: An Early Notion of an Intelligent System Built from Trainable Perceptrons

If $g(x)$ is exactly zero, the perceptron cannot decide whether to fire. A slight change in inputs could cause the device to go either way. If we solve the equation $g(x) = 0$, we get the equation for a line:

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2}$$

The location of the line is completely determined by the weights w_0 , w_1 , and w_2 . If an input vector lies on one side of the line, the perceptron will output 1; if it lies on the other side, the perceptron will output 0. A line that correctly separates the training instances corresponds to a perfectly functioning perceptron. Such a line is called a *decision surface*. In perceptrons with many inputs, the decision surface will be a hyperplane through the multidimensional space of possible input vectors. The problem of *learning* is one of locating an appropriate decision surface.

We present a formal learning algorithm later. For now, consider the informal rule:

If the perceptron fires when it should not fire, make each w_i smaller by an amount proportional to x_i . If the perceptron fails to fire when it should fire, make each w_i larger by a similar amount.

Suppose we want to train a three-input perceptron to fire only when its first input is on. If the perceptron fails to fire in the presence of an active x_1 , we will increase w_1 (and we may increase other weights). If the perceptron fires incorrectly, we will end up decreasing weights that are not w_1 . (We will never decrease w_1 because undesired firings only occur when x_1 is 0, which forces the proportional change in w_1 also to be 0.) In addition, w_0 will find a value based on the total number of incorrect firings versus incorrect misfirings. Soon, w_1 will become large enough to overpower w_0 , while w_2 and w_3 will not be powerful enough to fire the perceptron, even in the presence of both x_2 and x_3 .

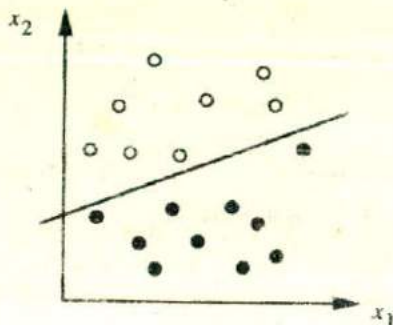


Figure 18.9: A Linearly Separable Pattern Classification Problem

Now let us return to the functions $g(x)$ and $o(x)$. While the sign of $g(x)$ is critical to determining whether the perceptron will fire, the magnitude is also important. The absolute value of $g(x)$ tells how far a given input vector \vec{x} lies from the decision surface. This gives us a way of characterizing how good a set of weights is. Let \vec{w} be the weight vector (w_0, w_1, \dots, w_n) , and let X be the subset of training instances misclassified by the current set of weights. Then define the *perceptron criterion function*, $J(\vec{w})$, to be the sum of the distances of the misclassified input vectors from the decision surface:

$$J(\vec{w}) = \sum_{\vec{x} \in X} \left| \sum_{i=0}^n w_i x_i \right| = \sum_{\vec{x} \in X} |\vec{w} \cdot \vec{x}|$$

To create a better set of weights than the current set, we would like to reduce $J(\vec{w})$. Ultimately, if all inputs are classified correctly, $J(\vec{w}) = 0$.

How do we go about minimizing $J(\vec{w})$? We can use a form of local-search hill climbing known as *gradient descent*. We have already seen in Chapter 3 how we can use hill-climbing strategies in symbolic AI systems. For our current purposes, think of $J(\vec{w})$ as defining a surface in the space of all possible weights. Such a surface might look like the one in Figure 18.10.

In the figure, weight w_0 should be part of the weight space but is omitted here because it is easier to visualize J in only three dimensions. Now, some of the weight vectors constitute solutions, in that a perceptron with such a weight vector will classify all its inputs correctly. Note that there are an infinite number of solution vectors. For any solution vector \vec{w}_s , we know that $J(\vec{w}_s) = 0$. Suppose we begin with a random weight vector \vec{w} that is not a solution vector. We want to slide down the J surface. There is a mathematical method for doing this—we compute the gradient of the function $J(\vec{w})$. Before we derive the gradient function, we reformulate the perceptron criterion function to remove the absolute value sign:

$$J(\vec{w}) = \sum_{\vec{x} \in X} \vec{w} \cdot \begin{cases} \vec{x} & \text{if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} & \text{if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

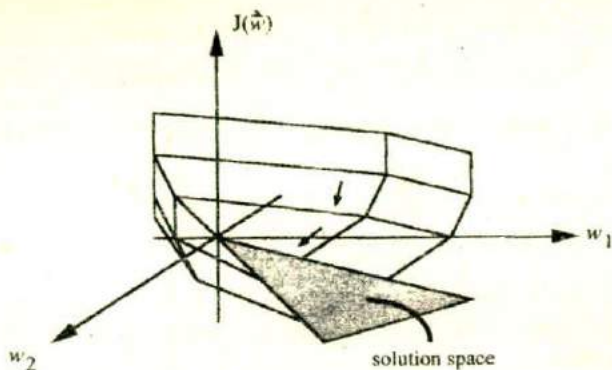


Figure 18.10: Adjusting the Weights by Gradient Descent, Minimizing $J(\vec{w})$

Recall that X is the set of misclassified input vectors.

Now, here is ∇J , the gradient of $J(\vec{w})$ with respect to the weight space:

$$\nabla J(\vec{w}) = \sum_{\vec{x} \in X} \begin{cases} \vec{x} & \text{if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} & \text{if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

The gradient is a vector that tells us the direction to move in the weight space in order to reduce $J(\vec{w})$. In order to find a solution weight vector, we simply change the weights in the direction of the gradient, recompute $J(\vec{w})$, recompute the new gradient, and iterate until $J(\vec{w}) = 0$. The rule for updating the weights at time $t + 1$ is:

$$\vec{w}_{t+1} = \vec{w}_t + \eta \nabla J$$

Or in expanded form:

$$\vec{w}_{t+1} = \vec{w}_t + \eta \sum_{\vec{x} \in X} \begin{cases} \vec{x} & \text{if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} & \text{if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

η is a scale factor that tells us how far to move in the direction of the gradient. A small η will lead to slower learning, but a large η may cause a move through weight space that "overshoots" the solution vector. Taking η to be a constant gives us what is usually called the "fixed-increment perceptron learning algorithm":

Algorithm: Fixed-Increment Perceptron Learning

Given: A classification problem with n input features (x_1, x_2, \dots, x_n) and two output classes.

Compute: A set of weights $(w_0, w_1, w_2, \dots, w_n)$ that will cause a perceptron to fire whenever the input falls into the first output class.

1. Create a perceptron with $n + 1$ inputs and $n + 1$ weights, where the extra input x_0 is always set to 1.
2. Initialize the weights (w_0, w_1, \dots, w_n) to random real values.
3. Iterate through the training set, collecting all examples *misclassified* by the current set of weights.
4. If all examples are classified correctly, output the weights and quit.
5. Otherwise, compute the vector sum S of the misclassified input vectors, where each vector has the form (x_0, x_1, \dots, x_n) . In creating the sum, add to S a vector \vec{x} if \vec{x} is an input for which the perceptron incorrectly *fails to fire*, but add vector $-\vec{x}$ if \vec{x} is an input for which the perceptron incorrectly *fires*. Multiply the sum by a scale factor η .
6. Modify the weights (w_0, w_1, \dots, w_n) by adding the elements of the vector S to them. Go to step 3.

The perceptron learning algorithm is a search algorithm. It begins in a random initial state and finds a solution state. The search space is simply all possible assignments of real values to the weights of the perceptron, and the search strategy is gradient descent. Gradient descent is identical to the hill-climbing strategy described in Chapter 3, except that we view good as "down" rather than "up."

So far, we have seen two search methods employed by neural networks, *gradient descent* in perceptrons and *parallel relaxation* in Hopfield networks. It is important to understand the relation between the two. Parallel relaxation is a problem-solving strategy, analogous to state space search in symbolic AI. Gradient descent is a learning strategy, analogous to techniques such as version spaces. In both symbolic and connectionist AI, learning is viewed as a type of problem solving, and this is why search is useful in learning. But the ultimate goal of learning is to get a system into a position where it can solve problems better. Do not confuse learning algorithms with others.

The *perceptron convergence theorem*, due to Rosenblatt [1962], guarantees that the perceptron will find a solution state, i.e., it will learn to classify any linearly separable set of inputs. In other words, the theorem shows that in the weight space, there are no local minima that do not correspond to the global minimum. Figure 18.11 shows a perceptron learning to classify the instances of Figure 18.9. Remember that every set of weights specifies some decision surface, in this case some two-dimensional line. In the figure, k is the number of passes through the training data, i.e., the number of iterations of steps 3 through 6 of the fixed-increment perceptron learning algorithm.

The introduction of perceptrons in the late 1950s created a great deal of excitement. Here was a device that strongly resembled a neuron and for which well-defined learning algorithms were available. There was much speculation about how intelligent systems could be constructed from perceptron building blocks. In their book *Perceptrons*, Minsky and Papert [1969] put an end to such speculation by analyzing the computational capabilities of the devices. They noticed that while the convergence theorem guaranteed correct classification of linearly separable data, most problems do not supply such nice data. Indeed, the perceptron is incapable of learning to solve some very simple problems. One example given by Minsky and Papert is the exclusive-or (XOR) problem. Given

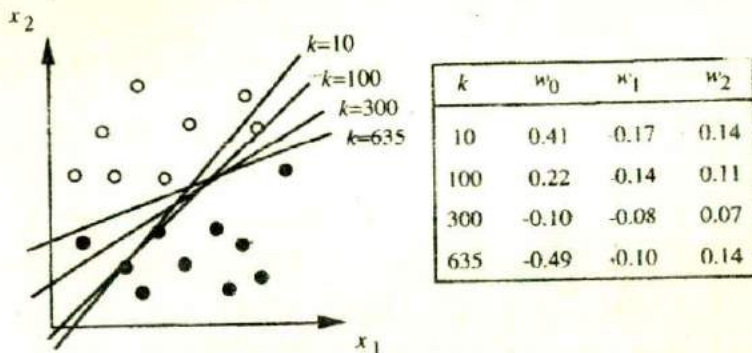


Figure 18.11: A Perceptron Learning to Solve a Classification Problem

two binary inputs, output 1 if *exactly* one of the inputs is on and output 0 otherwise. We can view XOR as a pattern classification problem in which there are four patterns and two possible outputs (see Figure 18.12).

The perceptron cannot learn a linear decision surface to separate these different outputs, *because no such decision surface exists*. No single line can separate the 1 outputs from the 0 outputs. Minsky and Papert gave a number of problems with this property including telling whether a line drawing is connected, and separating figure from ground in a picture. Notice that the deficiency here is not in the perceptron learning algorithm, but in the way the perceptron represents knowledge.

If we could draw an elliptical decision surface, we could encircle the two "1" outputs in the XOR space. However, perceptrons are incapable of modeling such surfaces. Another idea is to employ two separate line-drawing stages. We could draw one line to isolate the point $(x_1 = 1, x_2 = 1)$ and then another line to divide the remaining three points into two categories. Using this idea, we can construct a "multilayer" perceptron (a series of perceptrons) to solve the problem. Such a device is shown in Figure 18.13.

Note how the output of the first perceptron serves as one of the inputs to the second perceptron, with a large, negatively weighted connection. If the first perceptron sees the input $(x_1 = 1, x_2 = 1)$, it will send a massive inhibitory pulse to the second perceptron, causing that unit to output 0 regardless of its other inputs. If either of the inputs is 0, the second perceptron gets no inhibition from the first perceptron, and it outputs 1 if either of the inputs is 1.

The use of multilayer perceptrons, then, solves our knowledge representation problem. However, it introduces a serious learning problem: The convergence theorem does not extend to multilayer perceptrons. The perceptron learning algorithm can correctly adjust weights between inputs and outputs, but it cannot adjust weights between perceptrons. In Figure 18.13, the inhibitory weight "-9.0" was hand-coded, not learned. At the time *Perceptrons* was published, no one knew how multilayer perceptrons could be made to learn. In fact, Minsky and Papert speculated:

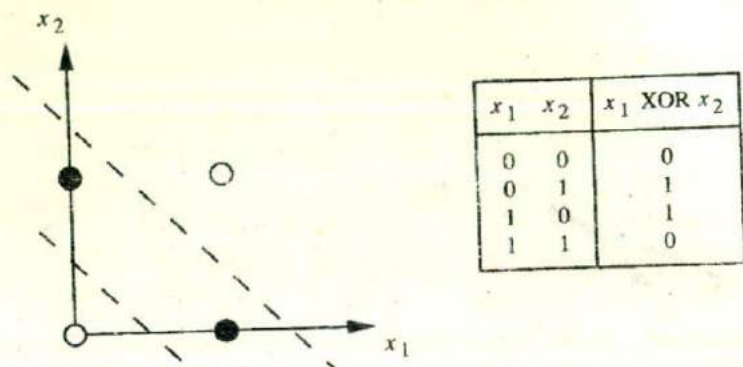


Figure 18.12: A Classification Problem. XOR, That Is Not Linearly Separable

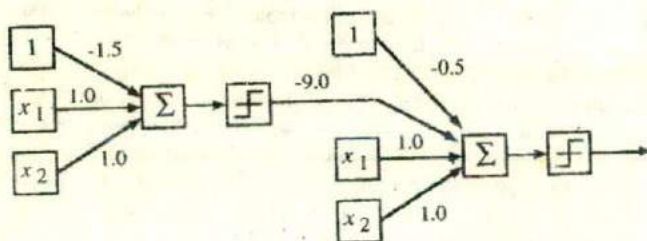


Figure 18.13: A Multilayer Perceptron That Solves the XOR Problem

The perceptron ... has many features that attract attention: its linearity, its intriguing learning theorem ... there is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile.

Despite the identification of this "important research problem," actual research in perceptron learning came to a halt in the 1970s. The field saw little interest until the 1980s, when several learning procedures for multilayer perceptrons—also called multilayer networks—were proposed. The next few sections are devoted to such learning procedures.

18.2.2 Backpropagation Networks

As suggested by Figure 18.8 and the *Perceptrons* critique, the ability to train multilayer networks is an important step in the direction of building intelligent machines from neuronlike components. Let's reflect for a moment on why this is so. Our goal is to

take a relatively amorphous mass of neuronlike elements and teach it to perform useful tasks. We would like it to be fast and resistant to damage. We would like it to generalize from the inputs it sees. We would like to build these neural masses on a very large scale, and we would like them to be able to learn efficiently. Perceptrons got us part of the way there, but we saw that they were too weak computationally. So we turn to more complex, multilayer networks.

What can a multilayer network compute? The simple answer is: *anything!* Given a set of inputs, we can use summation-threshold units as simple AND, OR, and NOT gates by appropriately setting the threshold and connection weights. We know that we can build any arbitrary combinational circuit out of those basic logical units. In fact, if we are allowed to use feedback loops, we can build a general-purpose computer with them.

The major problem is learning. The knowledge representation system employed by neural nets is quite opaque: the nets *must* learn their own representations because programming them by hand is impossible. Perceptrons had the nice property that whatever they could compute, they could learn to compute. Does this property extend to multilayer networks? The answer is yes, sort of. Backpropagation is a step in that direction.

It will be useful to deal first with a subclass of multilayer networks, namely *fully connected, layered, feedforward* networks. A sample of such a network is shown in Figure 18.14. In this figure, x_i , h_i , and o_i represent unit activation levels of input, hidden, and output units. Weights on connections between the input and hidden layers are denoted here by $w1_{ij}$, while weights on connections between the hidden and output layers are denoted by $w2_{ij}$. This network has three layers, although it is possible and sometimes useful to have more. Each unit in one layer is connected in the forward direction to every unit in the next layer. Activations flow from the input layer through the hidden layer, then on to the output layer. As usual, the knowledge of the network is encoded in the weights on connections between units. In contrast to the parallel relaxation method used by Hopfield nets, backpropagation networks perform a simpler computation. Because activations flow in only one direction, there is no need for an iterative relaxation process. The activation levels of the units in the output layer determine the output of the network.

The existence of hidden units allows the network to develop complex feature detectors, or internal representations. Figure 18.15 shows the application of a three layer network to the problem of recognizing digits. The two-dimensional grid containing the numeral "7" forms the input layer. A single hidden unit might be strongly activated by a horizontal line in the input, or perhaps a diagonal. The important thing to note is that the behavior of these hidden units is automatically learned, not preprogrammed. In Figure 18.15, the input grid appears to be laid out in two dimensions, but the fully connected network is unaware of this 2-D structure. Because this structure can be important, many networks permit their hidden units to maintain only local connections to the input layer (e.g., a different 4 by 4 subgrid for each hidden unit).

The hope in attacking problems like handwritten character recognition is that the neural network will not only learn to classify the inputs it is trained on but that it will *generalize* and be able to classify inputs that it has not yet seen. We return to generalization in the next section.

A reasonable question at this point is: "All neural nets seem to be able to do

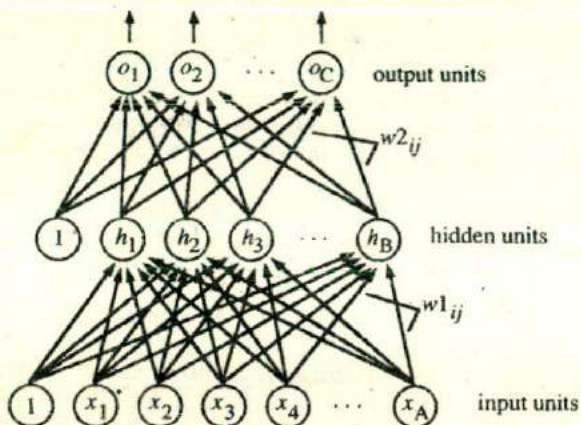


Figure 18.14: A Multilayer Network

is classification. Hard AI problems such as planning, natural language parsing, and theorem proving are not simply classification tasks, so how do connectionist models address these problems?" Most of the problems we see in this chapter are indeed classification problems, because these are the problems that neural networks are best suited to handle at present. A major limitation of current network formalisms is how they deal with phenomena that involve time. This limitation is lifted to some degree in work on recurrent networks (see Section 18.4), but the problems are still severe. Hence, we concentrate on classification problems for now.

Let's now return to backpropagation networks. The unit in a backpropagation network requires a slightly different activation function from the perceptron. Both functions are shown in Figure 18.16. A backpropagation unit still sums up its weighted inputs, but unlike the perceptron, it produces a real value between 0 and 1 as output, based on a sigmoid (or S-shaped) function, which is continuous and differentiable, as required by the backpropagation algorithm. Let *sum* be the weighted sum of the inputs to a unit. The equation for the unit's output is given by:

$$\text{output} = \frac{1}{1 + e^{-\text{sum}}}$$

Notice that if the sum is 0, the output is 0.5 (in contrast to the perceptron, where it must be either 0 or 1). As the sum gets larger, the output approaches 1. As the sum gets smaller, on the other hand, the output approaches 0.

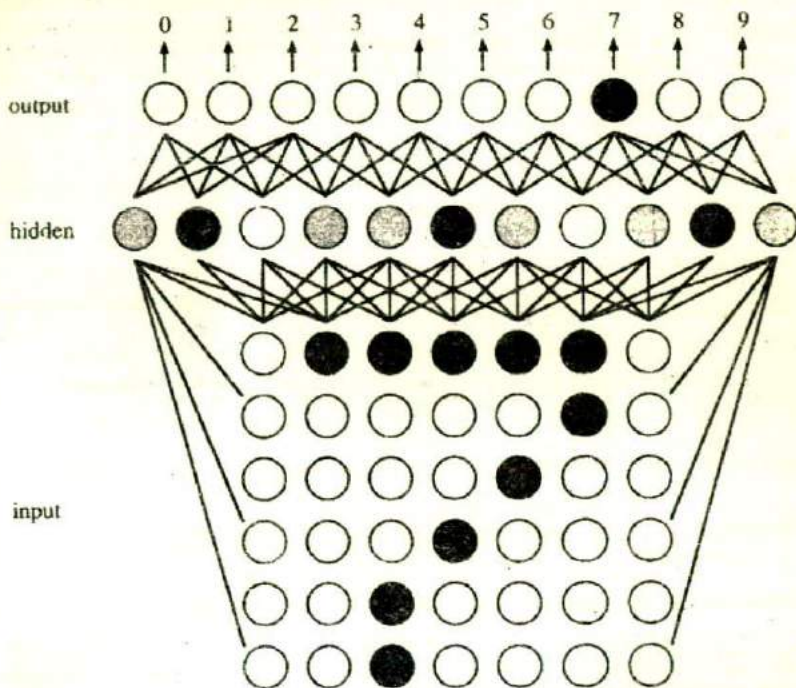


Figure 18.15: Using a Multilayer Network to Learn to Classify Handwritten Digits

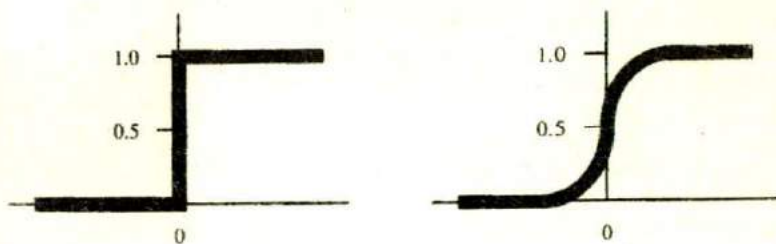


Figure 18.16: The Stepwise Activation Function of the Perceptron (*left*), and the Sigmoid Activation Function of the Backpropagation Unit (*right*)

Like a perceptron, a backpropagation network typically starts out with a random set of weights. The network adjusts its weights each time it sees an input-output pair. Each pair requires two stages: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activations flow until they reach the output layer. During the backward pass, the network's actual output (from the forward pass) is compared with the target output and error estimates are computed for the output units. The weights connected to the output units can be adjusted in order to reduce those errors. We can then use the error estimates of the output units to derive error estimates for the units in the hidden layers. Finally, errors are propagated back to the connections stemming from the input units.

Unlike the perceptron learning algorithm of the last section, the backpropagation algorithm usually updates its weights incrementally, after seeing each input-output pair. After it has seen all the input-output pairs (and adjusted its weights that many times), we say that one *epoch* has been completed. Training a backpropagation network usually requires many epochs.

Refer back to Figure 18.14 for the basic structure on which the following algorithm is based.

Algorithm: Backpropagation

Given: A set of input-output vector pairs.

Compute: A set of weights for a three-layer network that maps inputs onto corresponding outputs.

1. Let A be the number of units in the input layer, as determined by the length of the training input vectors. Let C be the number of units in the output layer. Now choose B , the number of units in the hidden layer.³ As shown in Figure 18.14, the input and hidden layers each have an extra unit used for thresholding; therefore, the units in these layers will sometimes be indexed by the ranges $(0, \dots, A)$ and $(0, \dots, B)$. We denote the activation levels of the units in the input layer by x_j , in the hidden layer by h_j , and in the output layer by o_j . Weights connecting the input layer to the hidden layer are denoted by $w1_{ij}$, where the subscript i indexes the input units and j indexes the hidden units. Likewise, weights connecting the hidden layer to the output layer are denoted by $w2_{ij}$, with i indexing to hidden units and j indexing to output units.
2. Initialize the weights in the network. Each weight should be set randomly to a number between -0.1 and 0.1 .

$$w1_{ij} = \text{random}(-0.1, 0.1) \quad \text{for all } i = 0, \dots, A, j = 1, \dots, B$$

$$w2_{ij} = \text{random}(-0.1, 0.1) \quad \text{for all } i = 0, \dots, B, j = 1, \dots, C$$

3. Initialize the activations of the thresholding units. The values of these thresholding units should never change.

³Successful large-scale networks have used topologies like 203-80-26 [Sejnowski and Rosenberg, 1987], 960-2-45 [Pomerleau, 1989], and 459-24-24-1 [Teodoro and Sejnowski, 1989]. A larger hidden layer results in a more powerful network, but too much power may be undesirable (see Section 18.2.3).

$$x_0 = 1.0$$

$$h_0 = 1.0$$

4. Choose an input-output pair. Suppose the input vector is x_i and the target output vector is y_i . Assign activation levels to the input units.
5. Propagate the activations from the units in the input layer to the units in the hidden layer using the activation function of Figure 18.16:

$$h_j = \frac{1}{1 + e^{-\sum_{i=0}^A w_{1ij} x_i}} \quad \text{for all } j = 1, \dots, B$$

Note that i ranges from 0 to A . w_{10j} is the thresholding weight for hidden unit j (its propensity to fire irrespective of its inputs). x_0 is always 1.0.

6. Propagate the activations from the units in the hidden layer to the units in the output layer.

$$o_j = \frac{1}{1 + e^{-\sum_{i=0}^B w_{2ij} h_i}} \quad \text{for all } j = 1, \dots, C$$

Again, the thresholding weight w_{20j} for output unit j plays a role in the weighted summation. h_0 is always 1.0.

7. Compute the errors⁴ of the units in the output layer, denoted $\delta 2_j$. Errors are based on the network's actual output (o_j) and the target output (y_j).

$$\delta 2_j = o_j(1 - o_j)(y_j - o_j) \quad \text{for all } j = 1, \dots, C$$

8. Compute the errors of the units in the hidden layer, denoted $\delta 1_j$.

$$\delta 1_j = h_j(1 - h_j) \sum_{i=1}^C \delta 2_i \cdot w_{2ji} \quad \text{for all } j = 1, \dots, B$$

9. Adjust the weights between the hidden layer and output layer.⁵ The learning rate is denoted η ; its function is the same as in perceptron learning. A reasonable value of η is 0.35.

⁴The error formula is related to the derivative of the activation function. The mathematical derivation behind the backpropagation learning algorithm is beyond the scope of this book.

⁵Again, we omit the details of the derivation. The basic idea is that each hidden unit tries to minimize the errors of output units to which it connects.

$$\Delta w_{2ij} = \eta \cdot \delta 2_j \cdot h_i \quad \text{for all } i = 0, \dots, B, j = 1, \dots, C$$

10. Adjust the weights between the input layer and the hidden layer⁶

$$\Delta w_{1ij} = \eta \cdot \delta 1_j \cdot x_i \quad \text{for all } i = 0, \dots, A, j = 1, \dots, B$$

11. Go to step 4 and repeat. When all the input-output pairs have been presented to the network, one epoch has been completed. Repeat steps 4 to 10 for as many epochs as desired.

The algorithm generalizes straightforwardly to networks of more than three layers.⁶ For each extra hidden layer, insert a forward propagation step between steps 6 and 7, an error computation step between steps 8 and 9, and a weight adjustment step between steps 10 and 11. Error computation for hidden units should use the equation in step 8, but with i ranging over the units in the next layer, not necessarily the output layer.

The speed of learning can be increased by modifying the weight modification steps 9 and 10 to include a momentum term α . The weight update formulas become:

$$\Delta w_{2ij}(t+1) = \eta \cdot \delta 2_j \cdot h_i + \alpha \Delta w_{2ij}(t)$$

$$\Delta w_{1ij}(t+1) = \eta \cdot \delta 1_j \cdot x_i + \alpha \Delta w_{1ij}(t)$$

where h_i , x_i , $\delta 1_j$ and $\delta 2_j$ are measured at time $t+1$. $\Delta w_{ij}(t)$ is the change the weight experienced during the previous forward-backward pass. If α is set to 0.9 or so, learning speed is improved.⁷

Recall that the activation function has a sigmoid shape. Since infinite weights would be required for the actual outputs of the network to reach 0.0 and 1.0, binary target outputs (the y_j 's of steps 4 and 7 above) are usually given as 0.1 and 0.9 instead. The sigmoid is required by backpropagation because the derivation of the weight update rule requires that the activation function be continuous and differentiable.

The derivation of the weight update rule is more complex than the derivation of the fixed-increment update rule for perceptrons, but the idea is much the same. There is an error function that defines a surface over weight space, and the weights are modified in the direction of the gradient of the surface. See Rumelhart *et al.* [1986] for details. Interestingly, the error surface for multilayer nets is more complex than the error surface for perceptrons. One notable difference is the existence of local minima. Recall the bowl-shaped space we used to explain perceptron learning (Figure 18.10). As we

⁶A network with one hidden layer can compute any function that a network with many hidden layers can compute: with an exponential number of hidden units, one unit could be assigned to every possible input pattern. However, learning is sometimes faster with multiple hidden layers, especially if the input is highly nonlinear, i.e., hard to separate with a series of straight lines.

⁷Empirically, best results have come from letting α be zero for the first few training passes, then increasing it to 0.9 for the rest of training. This process first gives the algorithm some time to find a good general direction, and then moves it in that direction with some extra speed.

modified weights, we moved in the direction of the bottom of the bowl; eventually, we reached it. A backpropagation network, however, may slide down the error surface into a set of weights that does not solve the problem it is being trained on. If that set of weights is at a local minimum, the network will never reach the optimal set of weights. Thus, we have no analogue of the perceptron convergence theorem for backpropagation networks.

There are several methods of overcoming the problem of local minima. The momentum factor α , which tends to keep the weight changes moving in the same direction, allows the algorithm to skip over small minima. *Simulated annealing*, discussed later in Section 18.2.4, is also useful. Finally, adjusting the shape of a unit's activation function can have an effect on the network's susceptibility to local minima.

Fortunately, backpropagation networks rarely slip into local minima. It turns out that, especially in larger networks, the high-dimensional weight space provides plenty of degrees of freedom for the algorithm. The lack of a convergence theorem is not a problem in practice. However, this pleasant feature of backpropagation was not discovered until recently, when digital computers became fast enough to support large-scale simulations of neural networks. The backpropagation algorithm was actually derived independently by a number of researchers in the past, but it was discarded as many times because of the potential problems with local minima. In the days before fast digital computers, researchers could only judge their ideas by proving theorems about them, and they had no idea that local minima would turn out to be rare in practice. The modern form of backpropagation is often credited to Werbos [1974], LeCun [1985], Parker [1985], and Rumelhart *et al.* [1986].

Backpropagation networks are not without real problems, however, with the most serious being the slow speed of learning. Even simple tasks require extensive training periods. The XOR problem, for example, involves only five units and nine weights, but it can require many, many passes through the four training cases before the weights converge, especially if the learning parameters are not carefully tuned. Also, simple backpropagation does not scale up very well. The number of training examples required is superlinear in the size of the network.

Since backpropagation is inherently a parallel, distributed algorithm, the idea of improving speed by building special-purpose backpropagation hardware is attractive. However, fast new variations of backpropagation and other learning algorithms appear frequently in the literature, e.g., Fahlman [1988]. By the time an algorithm is transformed into hardware and embedded in a computer system, the algorithm is likely to be obsolete.

18.2.3 Generalization

If all possible inputs and outputs are shown to a backpropagation network, the network will (probably, eventually) find a set of weights that maps the inputs onto the outputs. For many AI problems, however, it is impossible to give all possible inputs. Consider face recognition and character recognition. There are an infinite number of orientations and expressions to a face, and an infinite number of fonts and sizes for a character, yet humans learn to classify these objects easily from only a few examples. We would hope that our networks would do the same. And, in fact, backpropagation shows promise as a generalization mechanism. If we work in a domain (such as the classification domains just discussed) where similar inputs get mapped onto similar outputs, backpropagation

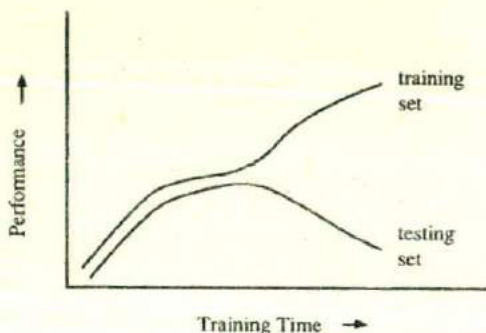


Figure 18.17: A Common Generalization Effect in Neural Network Learning

will interpolate when given inputs it has never seen before. For example, after learning to distinguish a few different sized As from a few different sized Bs, a network will usually be able to distinguish *any* sized A from *any* sized B. Also, generalization will help overcome any undesirable noise in the inputs.

There are some pitfalls, however. Figure 18.17 shows the common generalization effect during a long training period. During the first part of the training, performance on the training set improves as the network adjusts its weights through backpropagation. Performance on the test set (examples that the network is *not* allowed to learn on) also improves, although it is never quite as good as the training set. After a while, network performance reaches a plateau as the weights shift around, looking for a path to further improvement. Ultimately, such a path is found, and performance on the training set improves again. But performance on the test set gets worse. Why? The network has begun to memorize the individual input-output pairs rather than settling for weights that generally describe the mapping for all cases. With thousands of real-valued weights at its disposal, backpropagation is theoretically capable of storing entire training sets; with enough hidden units, the algorithm could learn to assign a hidden unit to every distinct input pattern in the training set. It is a testament to the power of backpropagation that this actually happens in practice.

Of course, that much power is undesirable. There are several ways to prevent backpropagation from resorting to a table-lookup scheme. One way is to stop training when a plateau has been reached, on the assumption that any other improvement will come through "cheating." Another way is to add deliberately small amounts of noise to the training inputs. The noise should be enough to prevent memorization, but it should not be so much that it confuses the classifier. A third way to help generalization is to reduce the number of hidden units in the network, creating a bottleneck between the input and output layers. Confronted with a bottleneck, the network will be forced to come up with compact internal representations of its inputs.

Finally, there is the issue of exceptions. In many domains, there are general rules,

but there are also exceptions to the rules. For example, we can generally make the past tense of an English verb by adding "-ed" to it, but this is not true of verbs like "sing," "think," and "eat." When we show many present and past tense pairs to a network, we would like it to generalize in spite of the exceptions—but not to generalize so far that the exceptions are lost. Backpropagation performs fairly well in this regard, as do simple perceptrons, as reported in Rumelhart and McClelland [1986a].

18.2.4 Boltzmann Machines

A Boltzmann machine is a variation on the idea of a Hopfield network. Recall that pairs of units in a Hopfield net are connected by symmetric weights. Units update their states asynchronously by looking at their local connections to other units.

In addition to serving as content-addressable memories, Hopfield networks can solve a wide variety of constraint satisfaction problems. The idea is to view each unit as a "hypothesis," and to place positive weights on connections between units representing compatible or mutually supporting hypotheses, and negative weights on connections between units representing incompatible hypotheses. As the Hopfield net settles into a stable state, it attempts to assign truth and falsity to the various hypotheses while violating as few constraints as possible. We see examples of how neural networks attack real-world constraint satisfaction problems in Section 18.3.

The main problem with Hopfield networks is that they settle into local minima. Having many local minima is good for building content-addressable memories, but for constraint satisfaction tasks, we need to find the *globally* optimal state of the network. This state corresponds to an interpretation that satisfies as many interacting constraints as possible. Unfortunately, Hopfield networks cannot find global solutions because they settle into stable states via a completely distributed algorithm. If a network reaches a stable state like state *A* in Figure 18.4, then no single unit is willing to change its state in order to move uphill, so the network will never reach globally optimal state *B*. If several units decided to change state simultaneously, the network might be able to scale the hill and slip into state *B*. We need a way to push networks into globally optimal states while maintaining our distributed approach.

At about the same time that Hopfield networks were developed, a new search technique, called *simulated annealing*, appeared in the literature. Simulated annealing, described in Chapter 3, is a technique for finding globally optimal solutions to combinatorial problems. Hinton and Sejnowski [1986] combined Hopfield networks and simulated annealing to produce networks called *Boltzmann machines*.

To understand how annealing applies, go back to Figure 18.4 and imagine it as a black box. Imagine further a ball rolling around in the box. If we could not see into the black box, how could we coax the ball into the deepest valley? By shaking the box, of course. Now, if we shake too violently, the ball will bounce from valley to valley at random. That is, if the ball were in valley *A*, it might jump to valley *B*; but if the ball were in valley *B*, it might jump to valley *A*. If we shake too *softly*, however, the ball might find itself in valley *A*, unable to jump out. The answer suggested by annealing is to shake the box violently at first, then gradually slow down. At some point, the probability of the ball jumping from *A* to *B* will be larger than the probability of jumping from *B* to *A*. The ball will very likely find its way to valley *B*, and as the shaking becomes softer, it will be unable to escape. This is what we want.

How is this idea implemented in a neural network? Units in Boltzmann machines update their individual binary states by a *stochastic* rather than deterministic rule. The probability that any given unit will be active is given by p :

$$p = \frac{1}{1 + e^{\Delta E/T}}$$

where ΔE is the sum of the unit's active input lines and T is the "temperature" of the network. Stochastic updating of units is very similar to updating in Hopfield nets, except for the temperature factor. At high temperatures, units display random behavior, while at very low temperatures, units behave as in Hopfield nets. Annealing is the process of gradually moving from a high temperature down to a low temperature. The randomness added by the temperature helps the network escape from local minima.

There is a learning procedure for Boltzmann machines, i.e., a procedure that assigns weights to connections between units given a training set of initial states and final states. We do not go into the algorithm here; interested readers should see Hinton and Sejnowski [1986]. Boltzmann learning is more time-consuming than backpropagation,⁸ because of the complex annealing process, but it has some advantages. For one thing, it is easier to use Boltzmann machines to solve constraint satisfaction problems. Unlike backpropagation networks, Boltzmann machines do not make a clear division between "input" and "output." For example, a Boltzmann machine might have three important sets of units, any two of which could have their values "clamped," or fixed, like the input layer of a backpropagation net—activations in the third set of units would be determined by parallel relaxation.

If the annealing is carried out properly, Boltzmann machines can avoid local minima and learn to compute any computable function of fixed-sized inputs and outputs.

18.2.5 Reinforcement Learning

What if we train our networks not with sample outputs but with punishment and reward instead? This process is certainly sufficient to train animals to perform relatively interesting tasks. Barto [1985] describes a network which learns as follows: (1) the network is presented with a sample input from the training set, (2) the network computes what it thinks should be the sample output, (3) the network is supplied with a real-valued judgment by the teacher, (4) the network adjusts its weights, and the process repeats. A positive value in step 3 indicates good performance, while a negative value indicates bad performance. The network seeks a set of weights that will prevent negative reinforcement in the future, much as an experimental rat seeks behaviors that will prevent electric shocks.

18.2.6 Unsupervised Learning

What if a neural network is given *no* feedback for its outputs, not even a real-valued reinforcement? Can the network learn anything useful? The unintuitive answer is yes.

⁸One deterministic variation of Boltzmann learning [Peterson and Anderson, 1987] promises to be more efficient.

	has-hair?	has-scales?	has-feathers?	flies?	lives in water?	lays eggs?
Dog	1	0	0	0	0	0
Cat	1	0	0	0	0	0
Bat	1	0	0	1	0	0
Whale	1	0	0	0	1	0
Canary	0	0	1	1	0	1
Robin	0	0	1	1	0	1
Ostrich	0	0	1	1	0	1
Snake	0	1	0	0	0	1
Lizard	0	1	0	0	0	1
Alligator	0	1	0	0	1	1

Figure 18.18: Data for Unsupervised Learning

This form of learning is called *unsupervised learning* because no teacher is required.⁹ Given a set of input data, the network is allowed to play with it to try to discover regularities and relationships between the different parts of the input.

Learning is often made possible through some notion of which features in the input set are important. But often we do not know in advance which features are important, and asking a learning system to deal with raw input data can be computationally expensive. Unsupervised learning can be used as a "feature discovery" module that precedes supervised learning.

Consider the data in Figure 18.18. The group of ten animals, each described by its own set of features, breaks down naturally into three groups: mammals, reptiles, and birds. We would like to build a network that can learn which group a particular animal belongs to, and to generalize so that it can identify animals it has not yet seen. We can easily accomplish this with a six-input, three-output backpropagation network. We simply present the network with an input, observe its output, and update its weights based on the errors it makes. Without a teacher, however, the error cannot be computed, so we must seek other methods.

Our first problem is to ensure that only one of the three output units becomes active for any given input. One solution to this problem is to let the network settle, find the output unit with the highest level of activation, and set that unit to 1 and all other output units to 0. In other words, the output unit with the highest activation is the only one we consider to be active. A more neural-like solution is to have the output units fight among themselves for control of an input vector. The scheme is shown in Figure 18.19. The input units are directly connected to the output units, as in the perceptron, but the output units are also connected to each other via prewired negative, or inhibitory, connections. The output unit with the most activation along its input lines initially will most strongly

⁹One analogue of unsupervised learning in symbolic AI is *discovery* (Section 17.7).

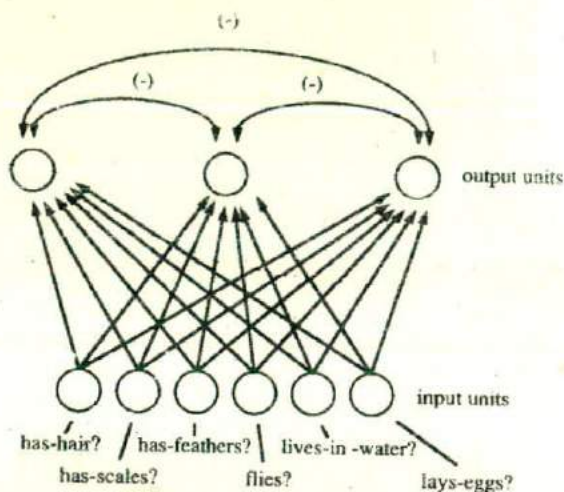


Figure 18.19: A Competitive Learning Network

dampen its competitors. As a result, the competitors will become weaker, losing their power of inhibition over the stronger output unit. The stronger unit then becomes even stronger, and its inhibiting effect on the other output units becomes overwhelming. Soon, the other output units are all completely inactive. This type of mutual inhibition is called *winner-take-all* behavior. One popular unsupervised learning scheme based on this behavior is known as *competitive learning*.

In competitive learning, output units fight for control over portions of the input space. A simple competitive learning algorithm is the following:

1. Present an input vector.
2. Calculate the initial activation for each output unit.
3. Let the output units fight until only one is active.
4. Increase the weights on connections between the active output unit and active input units. This makes it more likely that the output unit will be active next time the pattern is repeated.

One problem with this algorithm is that one output unit may learn to be active all the time—it may claim all the space of inputs for itself. For example, if all the weights on a unit's input lines are large, it will tend to bully the other output units into submission. Learning will only further increase those weights.

The solution, originally due to Rosenblatt (and described in Rumelhart and Zipser [1986]), is to ration the weights. The sum of the weights on a unit's input lines is limited to 1. Increasing the weight of one connection requires that we decrease the weight of some other connection. Here is the learning algorithm.

Algorithm: Competitive Learning

Given: A network consisting of n binary-valued input units directly connected to any number of output units.

Produce: A set of weights such that the output units become active according to some natural division of the inputs.

1. Present an input vector, denoted (x_1, x_2, \dots, x_n) .
2. Calculate the initial activation for each output unit by computing a weighted sum of its inputs.¹⁰
3. Let the output units fight until only one is active.¹¹
4. Adjust the weights on the input lines that lead to the single active output unit.

$$\Delta w_j = \eta \frac{x_j}{m} - \eta w_j \quad \text{for all } j = 1, \dots, n$$

where w_j is the weight on the connection from input unit j to the active output unit, x_j is the value of the j th input bit, m is the number of input units that are active in the input vector that was chosen in step 1, and η is the learning rate (some small constant). It is easy to show that if the weights on the connections feeding into an output unit sum to 1 before the weight change, then they will still sum to 1 afterward.

5. Repeat steps 1 to 4 for all input patterns for many epochs.

The weight update rule in step 4 makes the output unit more prone to fire when it sees the same input again. If the same input is presented over and over, the output unit will eventually adjust its weights for maximum activation on that input. Because input vectors arrive in a mixed fashion, however, output units never settle on a perfect set of weights. The hope is that each will find a natural group of input vectors and gravitate toward it, that is, toward high activations when presented with those inputs. The algorithm halts when the weight changes become very small.

The competitive learning algorithm works well in many cases, but it has some problems. Sometimes, one output unit will always win, despite the existence of more than one cluster of input vectors. If two clusters are close together, one output unit may learn weights that give it a high level of activation when presented with an input from either cluster. In other words, it may oscillate between the two clusters. Normally, another output unit will win occasionally and move to claim one of the two clusters. However, if the other output units are completely unexcitable by the input vectors, they may never win the competition. One solution, called "leaky learning," is to change

¹⁰There is no reason to pass the weighted sum through a sigmoid function, as we did with backpropagation, because we only calculate activation levels for the purpose of singling out the most highly activated output unit.

¹¹As mentioned earlier, any method for determining the most highly activated output unit is sufficient. Simulators written in a serial programming language may dispense with the neural circuitry and simply compare activations levels to find the maximum.

the weights belonging to relatively inactive output units as well as the most active one. The weight update rule for losing output units is the same as in the algorithm above, except that they move their weights with a much smaller η (learning rate). An alternative solution is to adjust the sensitivity of an output unit through the use of a bias, or adjustable threshold. Recall that this bias mechanism was used in perceptrons and corresponded to the propensity of a unit to fire irrespective of its inputs. Output units that seldom win in the competitive learning process can be given larger biases. In effect, they are given control over a larger portion of the input space. In this way, units that consistently lose are eventually given a chance to win and adjust their weights in the direction of a particular cluster.

18.3 Applications of Neural Networks

Connectionist models can be divided [Touretzky, 1989b] into the following categories based on the complexity of the problem and the network's behavior:

- Pattern recognizers and associative memories
- Pattern transformers
- Dynamic inferencers

Most of the examples we have seen so far fall into the first category. In this section, we also see networks that fall into the second category. General inferencing in connectionist networks is still at a primitive stage.

18.3.1 Connectionist Speech

Speech recognition is a difficult perceptual task (as we see in Chapter 21). Connectionist networks have been applied to a number of problems in speech recognition; for a survey, see Lippmann [1989]. Figure 18.20 shows how a three-layer backpropagation network can be trained to discriminate between different vowel sounds. The network is trained to output one of ten vowels, given a pair of frequencies taken from the speech waveform. Note the nonlinear decision surfaces created by backpropagation learning.

Speech production—the problem of translating text into speech rather than vice versa—has also been attacked with neural networks. Speech production is easier than speech recognition, and high performance programs are available. NETalk [Sejnowski and Rosenberg, 1987], a network that learns to pronounce English text, was one of the first systems to demonstrate that connectionist methods could be applied to real-world tasks.

Linguists have long studied the rules governing the translation of text into speech units called *phonemes*. For example, the letter "x" is usually pronounced with a "ks" sound, as in "box" and "axe." A traditional approach to the problem would be to write all these rules down and use a production system to apply them. Unfortunately, most of the rules have exceptions—consider "xylophone"—and these exceptions must also be programmed in. Also, the rules may interact with one another in unpleasant, unforeseen ways. A connectionist approach is simply to present a network with words and their

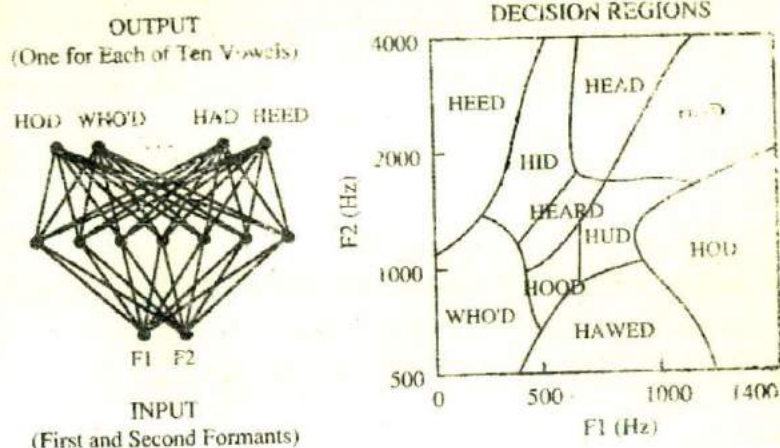


Figure 18.20: A Network That Learns to Distinguish Vowel Sounds

pronunciations, and hope that the network will discover the regularities and remember the exceptions. NETalk succeeds fairly well at this task with a backpropagation network of the type described in Section 18.2.2.

We can think of NETalk as an exercise in "extensional programming" [Courlet *et al.*, 1987]. There exists some complex relationship between text and speech, and we program that relationship into the computer by showing it examples from the real world. Contrast this with traditional, "intensional programming," in which we write rules or specialized algorithms without reference to any particular examples. In the former case, we hope that the network generalizes to translate new words correctly; in the latter case, we hope that the algorithm is general enough to handle whatever words it receives. Extensional programming is a powerful technique because it drastically cuts down on knowledge acquisition time, a major bottleneck in the construction of AI systems. However, current learning methods are not adequate for the extensional programming of very complex tasks, such as the translation of English sentences into Japanese.

18.3.2 Connectionist Vision

Humans achieve significant visual prowess with limited visual hardware. Only the center of the retina maintains good spatial resolution; as a result, we must constantly shift our attention among various points of interest. Each snapshot lasts only about two hundred milliseconds. Since individual neural firing rates usually lie in the millisecond range, each scene must be interpreted in about a hundred computational steps. To compound the problem, each interpretation must be rapidly integrated with previous interpretations to enable the construction of a stable three-dimensional model of the

world. These severe timing constraints strongly suggest that human vision is highly parallel. Connectionism offers many methods for studying both the engineering and biological aspects of massively parallel vision.

Parallel relaxation plays an important role in connectionist vision systems [Ballard *et al.*, 1983; Ballard, 1984]. Recall our discussion of parallel relaxation search in Hopfield networks and Boltzmann machines. In a typical system, some neural units receive their initial activation levels from a video camera and then these activations are iteratively modified based on the influences of nearby units. One use for relaxation is detecting edges. If many units think they are located on an edge border, they can override any dissenters. The relaxation process settles on the most likely set of edges in the scene. While traditional vision programs running on serial computing engines must reason about which regions of a scene require edge detection processing, the connectionist approach simply assumes massively parallel machinery [Fahlman and Hinton, 1987].

Visual interpretation also requires the integration of many constraint sources. For example, if two adjacent areas in the scene have the same color and texture, then they are probably part of the same object. If these constraints can be encoded in a network structure, then parallel relaxation is an attractive technique for combining them. Because relaxation treats constraints as "soft"—i.e., it will violate one constraint if necessary to satisfy the others—it achieves a global best-fit interpretation even in the presence of local ambiguity or noise.

18.3.3 Combinatorial Problems

Parallel relaxation can also be used to solve many other constraint satisfaction problems. Hopfield and Tank [1985] show how a Hopfield network can be programmed to come up with approximate solutions to the traveling salesman problem. The system employs n^2 neural units, where n is the number of cities to be toured. Figure 18.21 shows how tours themselves are represented. Each row stands for one city. The tour proceeds horizontally across the columns. The starting city is marked by the active unit in column 1, the next city by column 2, etc. The tour shown in Figure 18.21 goes through cities D, B, E, H, G, F, C, A, and back to D.

Like all Hopfield networks, this n by n array contains a number of weighted connections. The connection weights are initialized to reflect exactly the constraints of a particular problem instance.¹² First of all, every unit is connected with a negative weight to every other unit in its *column*, because only one city at a time can be visited. Second, every unit inhibits every other unit in its *row*, because each city can only be visited once. Third, units in adjacent columns inhibit each other in proportion to the distances between cities represented by their rows. For example, if city D is far from city G, then the fourth unit in column 3 will strongly inhibit the seventh units in columns 2 and 4. There is some global excitation, so in the absence of strong inhibition, individual units will prefer to be active.

Notice that each unit represents some hypothesis about the position of a particular city in a short tour. To find that tour, we start out by giving our units random activation values. Once all the weights are set, the units update themselves asynchronously.

¹²Note that these connection weights are hand-coded, not learned.

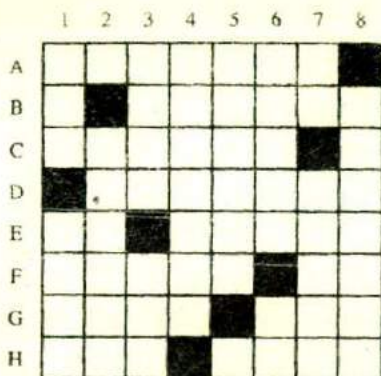


Figure 18.21: The Representation of a Traveling Salesman Tour in a Hopfield Network

according to the rule described in Section 18.1.¹³ This updating continues until a stable state is reached. Stable states of the network correspond to short tours because conflicts between constraints are minimal. Hopfield and Tank [1985] have used these networks to come up with quick, approximate solutions to traveling salesman problems (but see Wilson and Pawley [1988] for a critique of their results). Many other combinatorial problems, such as graph-coloring, can be cast as constraint satisfaction problems and solved with parallel relaxation networks.

18.3.4 Other Applications

Other tasks successfully tackled by neural networks include learning to play backgammon [Tesauro and Sejnowski, 1989], to classify sonar signals [Gorman and Sejnowski, 1988], to compress images [Cottrell *et al.*, 1987], and to drive a vehicle along a road [Pomerleau, 1989]. While there are other techniques for attacking all these problems, learning-based connectionist systems can often be built more quickly and with less expertise than their traditional counterparts.

18.4 Recurrent Networks

One clear deficiency of neural network models compared to symbolic models is the difficulty in getting neural network models to deal with temporal AI tasks such as planning and natural language parsing. Recurrent networks, or networks with loops, are an attempt to remedy this situation.

Consider trying to teach a network how to shoot a basketball through a hoop. We can present the network with an input situation (distance and height of hoop, initial position

¹³Actually, the units used by Hopfield and Tank [1985] take on real activation values (determined by a sigmoid curve) not binary values. By changing the shape of the sigmoid during processing, the network achieves some of the same results as does simulated annealing.

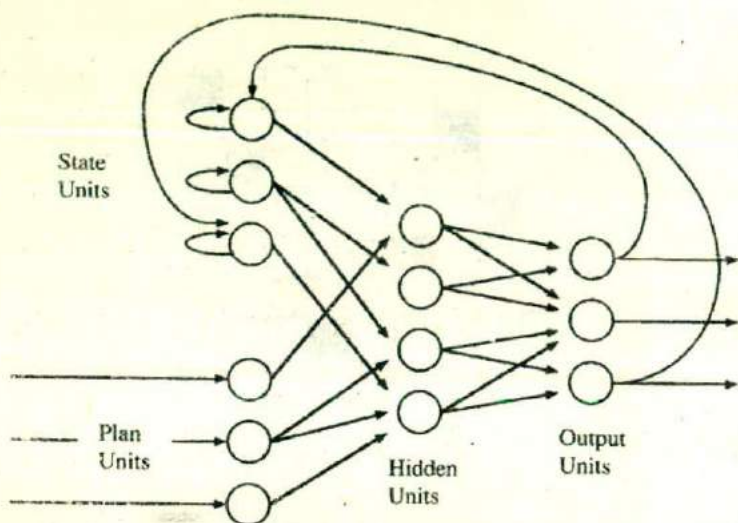


Figure 18.22: A Jordan Network

of muscles), but we need more than a single output vector. We need a series of output vectors: first move the muscles this way, then this way, then this way, etc. Jordan [1986] has invented a network that can do something like this. It is shown in Figure 18.22. The network's *plan units* stay constant. They correspond to an instruction like "shoot a basket." The *state units* encode the current state of the network. The *output units* simultaneously give commands (e.g., move arm x to position y) and update the state units. The network never settles into a stable state; instead it changes at each time step.

Recurrent networks can be trained with the backpropagation algorithm. At each step, we compare the activations of the output units with the desired activations and propagate errors backward through the network. When training is completed, the network will be capable of performing a sequence of actions. Features of backpropagation, such as automatic generalization, also hold for recurrent networks. A few modifications are useful, however. First of all, we would like the state units to change smoothly. For example, we would not like to move from a crouched position to a jumping position instantaneously. Smoothness can be implemented as a change in the weight update rule; essentially, the "error" of an output becomes a combination of real error and the magnitude of the change in the state units. Enforcing the smoothness constraint turns out to be very important in fast learning, as it removes many of the weight-manipulation options available to backpropagation.

A major problem in supervised learning systems lies in correcting the network's behavior. If enough training data can be amassed, then target outputs can be provided for many input vectors. Recurrent networks have special training problems, however, because it is difficult to specify completely a series of target outputs. In shooting basketballs, for example, the feedback comes from the external world (i.e., where the

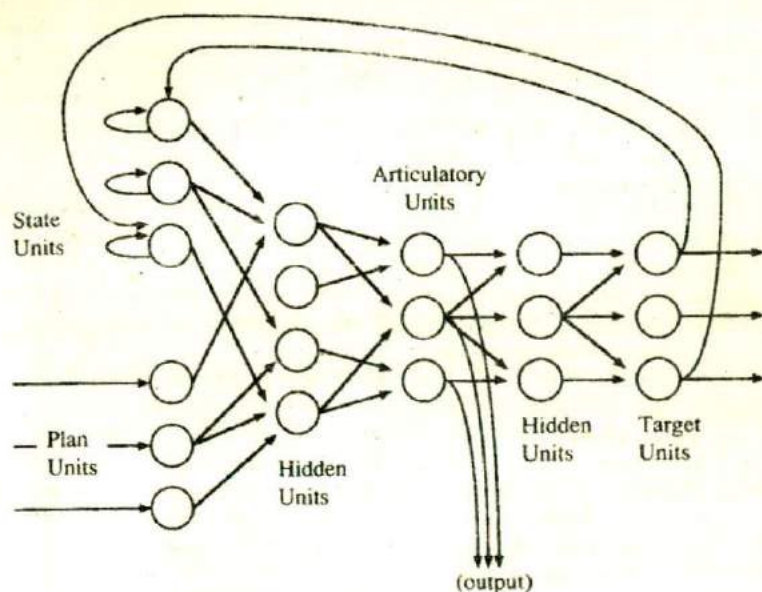


Figure 18.23: A Recurrent Network with a Mental Model

basketball lands), not from a teacher showing how to move each muscle. To get around this difficulty, we can learn a *mental model*, a mapping that relates the network's outputs to events in the world. Once such a model is known, the system can learn sequential tasks by backpropagating the errors it sees in the real world. So it is necessary to learn two different things: the relationship between the plan and the network's output, and the relationship between the network's output and the real world.

Networks of this type are described by Jordan [1988]. Figure 18.23 shows such a network, which is essentially the same as a Jordan net except for the addition of two more layers: another hidden layer and a layer representing results as seen in the world. First, the latter portion of the network is trained (using backpropagation) on various pairs of outputs and targets until the network gets a good feel for how its outputs affect the real world. After these rough weights are established, the whole network is trained using real-world feedback until it is able to perform accurately.

Another type of recurrent network is described in Elman [1990]. In this model, activation levels are explicitly copied from hidden units to state units. Networks of this kind have been used in a number of applications, including natural language parsing.

18.5 Distributed Representations

As we have seen, the long-term knowledge of a connectionist network is stored as a set of weights on connections between units. This general scheme admits many kinds of representations, just as the basic slot-and-filler structure left room for all the representations discussed in Chapters 9 and 10. Connectionist networks can be divided roughly into two classes: those that use *localist* representations and those that use *distributed* representations.

NETL [Fahlman, 1979] is a highly parallel system that employs a localist representation. Each node in a NETL network stands for one concept in a semantic network. For example, there is a node for "elephant," a node for "gray," etc. When the network is considering an elephant, the elephant unit becomes active. This unit then activates neighboring units, such as units for gray, large, and mammal. The reverse process works nicely as a content-addressable memory.

Distributed representations [Hinton *et al.*, 1986], on the other hand, do not use individual units to represent concepts; they use patterns of activations over many units. We have already seen one example of how this works: A Hopfield network provides a distributed representation for a content-addressable memory, in which each structure is stored as a collection of active units. One might be tempted to say that digital computers also use distributed representations. After all, a small integer is stored in a distributed fashion, as a pattern of activation over eight storage locations, each of which represents one bit of data. An extreme localist approach, on the other hand, would be to use 256 bits per integer, only one of which could be active at any given time. However, besides storing objects as patterns across many units, distributed representations have another important property, namely that stored objects may be superimposed on one another: One set of units can thus store many different objects. It is clearly impossible to store two 8-bit integers in one 8-bit place-holder, so we do not view such an encoding as a truly distributed representation.

Distributed representations have several advantages over localist ones. For one thing, they are more resistant to damage. If NETL loses its "elephant" unit somehow, then it immediately loses all ability to reason or remember about elephants. This fragility is undesirable if our goal is to build very large systems from unreliable parts. Also, it does not conform to what we know about human and animal memory. Lashley [1929] performed a number of classic experiments concerning memories in rats. Lashley wanted to find out in which part of its brain a rat stores its knowledge of how to run a particular maze. In the experiments, rats' brains were lesioned in many different places. Performance degraded in all rats in proportion to the size of the lesion, but the location of the lesion had no special effect on performance. Lashley concluded that the memory of how to run the maze was somehow stored in a distributed fashion across the entire rat cortex. Such a memory organization has been described using a hologram metaphor, in reference to the holographic storage medium, which allows the reconstruction of the entire image from just a portion of the recording (although the reconstructed image may be of poorer quality than the original). Work on distributed representations brings this metaphor down to an implementational level.

In addition to being more robust than localist representations, distributed representations can also be more efficient. Consider the problem of describing the locations of objects on a two-dimensional 8 by 8 grid [Hinton *et al.*, 1986]. In a symbolic implemen-

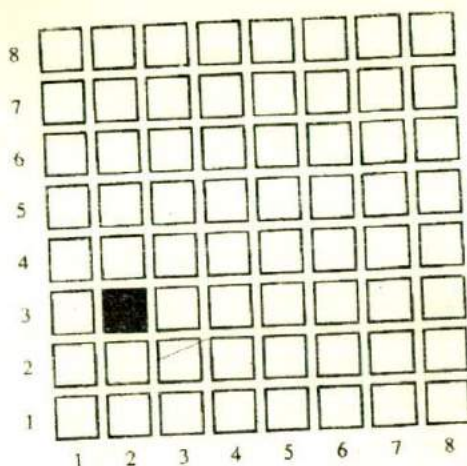


Figure 18.24: A Localist Representation of Location (2 3) on an 8 by 8 Grid

tation, this task is easy: A location can be stored simply as a list of two numbers, e.g., (2 3). Multiple object locations can be stored easily in this notation, as a list of lists: ((2 3) (6 5) (7 1)). How can we accomplish the same task with neuronlike units? The localist approach is to maintain an array of sixty-four units, one unit for every possible location (see Figure 18.24). A more efficient approach would be to use a group of eight units for the *x*-axis and another group of eight units for the *y*-axis, as in Figure 18.25. To represent the location (2 3), we activate two units: the second unit of the *x*-axis group and the third unit of the *y*-axis group. The other 14 units remain inactive. This method is not very damage resistant, however, and it will not support the representation of multiple object locations. To represent both (2 3) and (6 5) would require turning on two *x*-axis units and two *y*-axis units. But then we get the following *binding problem*: it is impossible to tell which of the four *x-y* pairs (2 3), (2 5), (6 3), and (6 5) correspond to actual object locations.

There is a distributed representation for solving this problem—it is called *coarse coding*. In coarse coding, we divide the space of possible object locations into a number of large, overlapping, circular zones. See, for example, Figure 18.26, in which units are depicted as small dots and their receptive fields as large circles. A unit becomes active if any object is located within its receptive field. There is a unit associated with each zone—the zone is called the unit's *receptive field*.¹⁴ Whenever an object is located in a unit's receptive field, the unit becomes active. By looking at a single active unit, we cannot tell with any accuracy where an object is located, but by looking at the pattern of activity across all the units, we can actually be quite precise. Consider that the intersection of several circular zones associated with a group of units may be a

¹⁴The term *receptive field* comes from the study of vision. A receptive field of a retinal cell is an area of the retina that the cell is responsible for. The cell is triggered by light in that area.

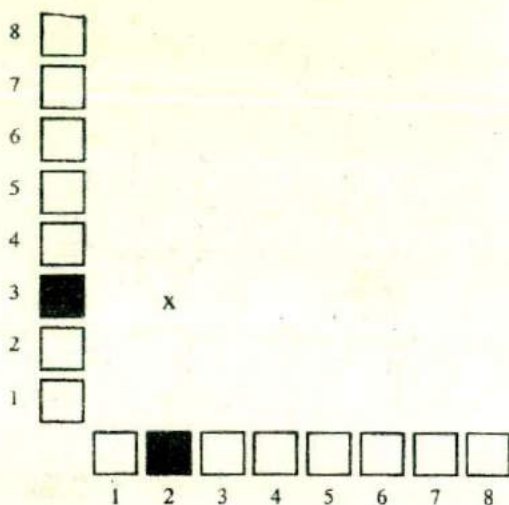


Figure 18.25: A More Efficient Representation, Requiring Only 16 Units, but Unable to Store Multiple Locations

very small area—if only those units are active, we can be fairly precise about where the object is located. In fact, as the receptive fields become larger, i.e., as the individual units become *less* discriminating about object locations, the *whole* representation becomes *more* accurate, because the regions of intersection become smaller. In the end, we can represent multiple objects with some precision without paying the price of the localist representation scheme.

One drawback to distributed representations is that they cannot store many densely packed objects. A localist or symbolic system could easily represent the three distinct objects at (4,4), (4,5), and (5,4), but a distributed scheme would be confounded by the loss of information caused by the effect of many objects on a single unit's receptive field. On the other hand, psychological experiments have shown that a similar *interference effect* is very likely a cause of forgetting in human memory [Gleitman, 1981]. A more serious deficiency concerning distributed representations lies in the difficulty of interpreting, acquiring, and modifying them by hand. Thus, they are usually used in conjunction with automatic learning mechanisms of the type discussed in Section 18.2.

18.6 Connectionist AI and Symbolic AI

The connectionist approach to AI is quite different from the traditional symbolic approach. Both approaches are certainly joined at the problem; both try to address difficult issues in search, knowledge representation, and learning. Let's list some of the methods used by both:

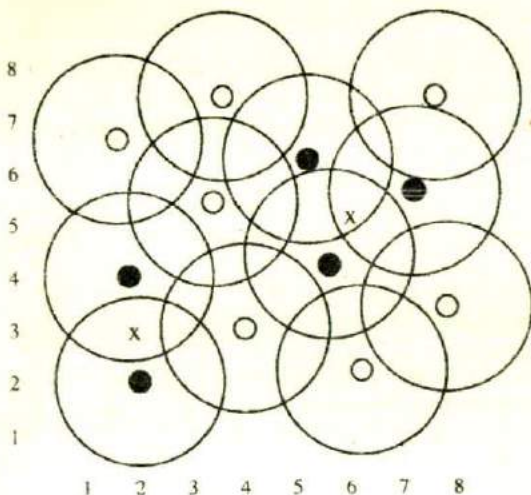


Figure 18.26: Distributed Representation Using Coarse Coding

1. Connectionist

- Search—Parallel relaxation.
- Knowledge Representation—Very large number of real-valued connection strengths. Structures often stored as distributed patterns of activation.
- Learning—Backpropagation, Boltzmann machines, reinforcement learning, unsupervised learning.

2. Symbolic

- Search—State space traversal.
- Knowledge Representation—Predicate logic, semantic networks, frames, scripts.
- Learning—Macro-operators, version spaces, explanation-based learning, discovery.

The approaches have different strengths and weaknesses. One major allure of connectionist systems is that they employ knowledge representations that seem to be more learnable than their symbolic counterparts. Nearly all connectionist systems have a strong learning component. However, neural network learning algorithms usually involve a large number of training examples and long training periods compared to their symbolic cousins. Also, after a network has learned to perform a difficult task, its knowledge is usually quite opaque, an impenetrable mass of connection weights. Getting the network to explain its reasoning, then, is difficult. Of course, this may not

be a bad thing. Humans, for example, appear to have little access to the procedures they use for many tasks such as speech recognition and vision. It is no accident that the most promising uses for neural networks are in these areas of low-level perception.

Connectionist knowledge representation offers other advantages besides learnability. Touretzky and Geva [1987] discuss the fluidity and richness of connectionist representations. In connectionist models, concepts are represented as feature vectors, sets of activation values over groups of units. Similar concepts are given similar feature vector representations. In symbolic models, on the other hand, concepts are usually given atomic labels that bear no surface relation to each other, such as *Car* and *Porsche*. Links (like *isa*) are used to describe relationships between concepts. When the relationships become more fuzzy than *isa*, however, symbolic systems have difficulty doing matching. For example, consider the phrases "mouth of a bird" and "nose of a bird." People have no trouble mapping these phrases onto the concept *Beak*. A connectionist system could perform this fuzzy match by considering that *Nose*, *Mouth*, and *Beak* have similar feature value representations. Moreover, symbolic systems do not handle multiple, related shades of meaning very well. Consider the sentence, "The newspaper changed its format." Usually, the word "newspaper" is interpreted either as (1) something made of black and white paper or (2) a group of people in charge of producing a daily periodical. In the sentence above, however, it is impossible to choose between the two readings. In symbolic systems, different word senses are represented as independent atomic objects. Connectionist models offer several ways of maintaining multiple meanings: the simultaneous activations of different units (localist), the superposition of activity patterns (distributed), and the choice of intermediate feature vectors. The third method involves choosing a representation that shares some features of one meaning and some feature of another, but the intermediate representation itself has no single, corresponding symbolic concept.

A major part of this book has been devoted to the study of search in symbolic systems. It is difficult to see how connectionist systems will tackle difficult problems that state-space search addresses (e.g., chess, theorem-proving, and planning). Parallel relaxation search, however, does have some advantages over symbolic search. First of all, it maps naturally onto highly parallel hardware. When such hardware becomes widely available, parallel relaxation methods will be extremely efficient. More importantly, parallel relaxation search may prove even more efficient because it makes use of states that have no analogues in symbolic search. We saw this phenomenon briefly in Section 18.3.3 when we considered a Hopfield network that comes up with short traveling salesman tours. In the process of settling into a solution state, the network enters and exits many "impossible" states, such as ones in which a city is visited twice, or ones in which the traveler is in two places at the same time. Eventually, a valid solution state falls out of the relaxation process. In contrast, a symbolic system can only expand new search nodes that correspond to valid, possible states of the world.

A good deal of connectionist research concerns itself with modeling human mental processes. Neural networks seem to display many psychologically and biologically plausible features such as: content-addressable memory, fault tolerance, distributed representations, and automatic generalization. Can we integrate these desirable properties into symbolic AI systems? Certainly, high-level theories of cognition can incorporate such features as new psychological primitives. Practically speaking, we may want to use connectionist architectures for low-level tasks such as vision, speech recognition, and

memory, feeding results from these modules into symbolic AI programs. Another idea is to take a symbolic notion and implement it in a connectionist framework. Touretzky and Hinton [1988] describe a connectionist production system, and Derthick [1988] describes a connectionist semantic network.

A third idea is to program a symbolic system with the basic principles that are necessary to perform a task and then use the symbolic system to guide the performance of a neural network, which refines its behavior as it acquires experience. An example of this approach is described by Handelman *et al.* [1989], who describe a robot arm that can throw a ball at a target. Initially, a symbolic system guides the behavior of the arm. Each throw produces a training case, which is fed to a neural network. The symbolic system monitors the progress of the network, which is acquiring the fine motor control that the symbolic system lacks. When the network's behavior exceeds a set criterion, control of the arm is turned over to it.

Ultimately, connectionists would like to see symbolic structures "emerge" naturally from complex interactions among simple units, in the same way that "wetness" emerges from the combination of hydrogen and oxygen, although it is an intrinsic property of neither.

Most of the promising advantages of connectionist systems described in this section are just that: promising. A great deal of work remains to be done to turn these promises into results. Only time will tell how influential connectionist models will be in the evolution of AI research. In any case, connectionists can at least point to the brain as an existence proof that neural networks, in some form, are capable of exhibiting intelligent behavior.

18.7 Exercises

1. Consider a Hopfield net with the symmetric, weighted connections of Figure 18.1. If all the units are initially active, which of the four states in Figure 18.2 will the network settle into?
2. Implement the fixed-increment perceptron learning algorithm. Invent a three-feature linearly separable classification problem on which to test your program.
3. Implement the backpropagation learning algorithm for a fully connected three-layer network. Be sure to include parameters for layer sizes, learning rate (η), and number of training epochs. Test your implementation first on the OR problem:

Input Vector	Target Output Vector
(0.0, 0.0)	(0.1)
(0.0, 1.0)	(0.9)
(1.0, 0.0)	(0.9)
(1.0, 1.0)	(0.9)

Then on the XOR problem:

Input Vector	Target Output Vector
(0.0, 0.0)	(0.1)
(0.0, 1.0)	(0.9)
(1.0, 0.0)	(0.9)
(1.0, 1.0)	(0.1)

Initially, use two hidden units, set $\eta = 0.35$, and run for 6000 training epochs. (Each epoch consists of forward and backward propagation of each of the four training examples.) Modify your program to use the momentum factor $\alpha = 0.9$. Did adding momentum significantly decrease the number of training epochs required for learning?

4. Here is a toy problem for testing generalization in networks. Suppose that there are eight political issues on which every political party must decide, and suppose further that those decisions are binary (for example, to legalize gambling or not, to increase military spending or not, etc.). We can then represent the platform of a political party as a vector of eight ones and zeros. Individuals who belong to political parties may have beliefs that differ slightly from their party's platform. Your job is to train a backpropagation network to compute the political platform of the party that most closely matches a given individual's beliefs.

Generate four random 8-bit vectors to represent the platforms of four political parties. For each party, generate nine individuals who belong to that party. The beliefs of an individual, like those of a party, are represented as an 8-bit vector. One of the nine individuals should agree entirely with the party platform, and the other eight should differ on exactly one issue (1 bit). Now generate 36 input-output pairs, by juxtaposing individuals with the platforms of their respective political parties. Each input is 8 bits, and each output is 8 bits.

Input Vector	Target Output Vector
individual ₁	party ₁
individual ₂	party ₁
...	...
individual ₉	party ₁
individual ₁₀	party ₂
individual ₁₁	party ₂
...	...
individual ₁₈	party ₂
...	...
...	...
individual ₃₆	party ₄

Next, remove five of the input-output pairs. These five will make up the "testing set"; the other 31 will make up the "training set." Create a backpropagation network with eight input units, eight hidden units, and eight output units. Train the network on the 31 vectors in the training set until performance is very high. Now test the network on the five input-output pairs it has never seen before. How well does it perform? Experiment with the different sizes of testing and training

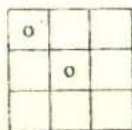
sets, as well as hidden layers of different sizes. Finally, how does the network perform when given individuals whose beliefs are not so close to one of the four parties?

5. Many people consider connectionism to be irrelevant to AI, because it studies intelligence at such a low level. They argue that intelligence should be modeled at a higher, more abstract level. They often relate connectionism to symbolic AI with a software metaphor that runs: "If you want to study the behavior of a complex LISP program, then you should inspect its input and outputs, its functions, its data, its general flow of control, but you should not be concerned about the particular hardware the program happens to be running on. The same goes for the study of intelligence." Read both Broadbent [1985] and Rumelhart and McClelland [1986b], and comment on this line of reasoning.
6. In contrast to those who view connectionism merely as an implementational theory, others believe that connectionist models are too abstract and that we should look more closely at the organization of the brain for clues about how to organize artificial networks. Consider the following facts about the brain [Crick and Asanuma, 1986; Rosenzweig and Leiman, 1982], and comment on how they might affect current connectionist models of memory, learning, and problem solving:
 - Neurons excite and inhibit one another, but an individual neuron is either purely excitatory or purely inhibitory. Neuron A cannot excite neuron B while inhibiting neuron C.
 - Neurons communicate through their firing rates, which range from a few spikes per second to perhaps 500 spikes per second. Neuron firing is asynchronous; there appears to be no global clock. There are two types of neural summation: (1) spatial summation, in which the effects of various connecting neurons are added together and (2) temporal summation, in which asynchronously arriving impulses are likely to cause a neuron to fire when they all arrive closely together in time. As a corollary of (2), one neuron can have a very great effect on another by firing very rapidly.
 - Some behavioral functions, such as vision and language, appear to be localized in the brain. Destruction of small portions of the brain can result in the complete inability to perform certain cognitive tasks.
 - The human brain has at least 150 billion neurons and probably 1000 to 10,000 connections per neuron. The brain of a rat is 700 times smaller. The proportion of the brain taken up by the cortex is much larger in humans than in rats.
 - Neurons are not connected symmetrically: if the axon of neuron A leads to neuron B, then the axon of neuron B probably does not lead back to neuron A.
 - There are many different types of neurons. Some types do not project out of their local area of the brain, while the axons of other types travel long distances.

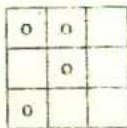
7. *Life* is a one-player game invented by John Conway. The game is played on an infinite two-dimensional grid of square cells. At any given time, a cell is either living or dead. Patterns of cells transform themselves according to a simple set of rules:

- If a cell is living, it continues to live if surrounded by exactly two or three living cells. If it is surrounded by more than three living cells, it dies of overcrowding; if less than two of its neighbors is alive, it dies of loneliness.
- If a cell is dead, it becomes living only if it is surrounded by exactly three living cells. Otherwise, it remains dead.

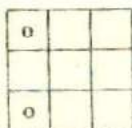
For example:



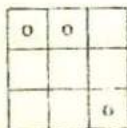
→ cell becomes dead



→ cell stays living



→ cell stays dead



→ cell becomes living

- Create input-output pairs for every possible configuration of a cell and its eight neighbors. There will be 512 (2^9) different input vectors. Associated with each input vector will be one output bit: 0 if the next state of the cell is dead, 1 if living. Use the rules above to compute the proper output for each input vector.
- Train a three-layer backpropagation network to learn the behavior of a Life cell. Use two hidden units.
- Print out the set of weights and biases learned by the network. Now derive a set of (symbolic) rules that concisely describes how the network is actually computing its output. Focus on the behaviors of the two hidden units—how do they respond to their inputs, and what effects do they have on the eventual output?
- Compare the rules you derived in part (c) with the rules you used to create the data in part (a).

Chapter 19

Common Sense

Computers have an entirely deserved reputation for lacking common sense. Anyone who has ever received a bill for \$0.00 from an accounting program can attest to this fact. An AI program may possess more knowledge than an accounting program, but it still computes using primitives that it knows nothing about. For example, consider the following interaction between a medical diagnosis system and a human (adapted from Lenat and Guha [1990]):

System: How old is the patient?

Human (looking at his 1957 Chevrolet): 33.

System: Are there any spots on the patient's body?

Human (noticing rust spots): Yes.

System: What color are the spots?

Human: Reddish-brown.

System: The patient has measles (probability 0.9).

Obviously, the system does not really know what measles are, what spots are, or what the difference between cars and people is. Even within its specialty, the system is unaware of fundamental facts, for example, that humans have two arms. Clearly, what the system lacks is knowledge. So far in this book, we have seen a number of techniques that can be used to enable an AI program to represent and reason with commonsense knowledge. For example, in predicate logic, one can state facts such as "if you die, you are dead at all later times." Frames can describe everyday objects, and scripts can describe the typical sequences of events. Nonmonotonic logics can support default reasoning, an important aspect of common sense.

As of yet, however, no program can match the commonsense reasoning powers of a five-year-old child. This is due, in part, to the large amount of knowledge required for common sense. In Section 10.3, we discussed the CYC program, one attempt to codify this information in a large knowledge base. In this chapter, we look more closely at the kinds of knowledge such a system must possess. In particular, we investigate how to understand and predict physical processes, how to model the behavior of materials, and how to reason about time and space. Memory is another key aspect to common sense. We look at how a memory can organize experiences, generalize them, and use them to solve new problems.

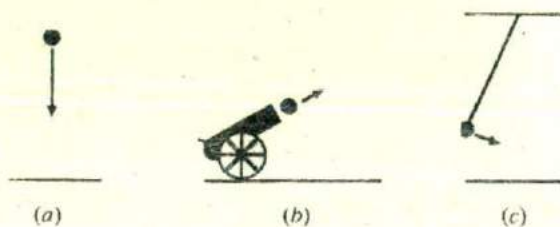


Figure 19.1: Three Physical Situations

19.1 Qualitative Physics

People know a great deal about how the physical world works. Consider the three situations shown in Figure 19.1.

Anyone can predict what will happen in these scenarios. In situation (a), the ball will probably bounce on the ground several times, then come to rest. In situation (b), the ball will travel upward and to the right, then downward. In situation (c), the ball will swing repeatedly from left to right, finally coming to rest in the middle. Now, how can we build a computer program to do this kind of reasoning?

The obvious answer is to program in the equations governing the physical motion of objects. These equations date back to classical physics and appear in every introductory physics textbook. For example, if the initial velocity of the ball in Figure 19.1(b) is v_0 , and the angle of its departure from the ground is θ , then the ball's position t seconds after being launched is given by:

$$\begin{aligned} \text{height} &= v_0 \cdot t \cdot \sin(\theta) - \frac{1}{2}gt^2 \\ \text{distance} &= v_0 \cdot t \cdot \cos(\theta) \end{aligned}$$

We can do the same thing for Figures 19.1(a) and (c). For Figure 19.1(a), we need to know the coefficient of elasticity, and for Figure 19.1(c), we need to know the length of the string, the initial velocity of the ball, and its original horizontal displacement.

There are two problems with this approach. First, most people do not know these equations, yet they are perfectly capable of predicting what will happen in physical situations. Also, unlike equations, people do not need exact, numerical measures. They need only *qualitative* descriptions, such as the ones given at the beginning of this section. People seem to reason more abstractly than the equations would indicate. The goal of qualitative physics is to understand how to build and reason with abstract, numberless representations.

One might object to qualitative physics on the grounds that computers are actually well-suited to model physical processes with numerical equations. After all, a computer's ability to solve simultaneous equations far outstrips that of a human. However, we cannot escape common sense so easily. Equations themselves say nothing about when they should be used; this is usually left up to a human physicist. The common-sense knowledge employed by the physicist is part of what we must model. While some sort of qualitative physics seems necessary for automating the solution of physics

problems, it is not sufficient by itself. The goal of qualitative physics is not to replace traditional physics but rather to provide a foundation for programs that can reason about the physical world. One such program might be a physics expert system.

As a further illustration of the need for qualitative models, consider a scene in which a glass of water leans precariously against a book on top of a cluttered desk. When the book is moved, the glass begins to tip over. At present, no set of differential equations can accurately model exactly how the spilling water will flow across the desk. Even if such a model existed, it would be impossible to measure the initial conditions accurately enough to make an accurate prediction. Yet anyone in this situation can immediately visualize what is likely to happen and take rapid action to prevent it.

19.1.1 Representing Qualitative Information

Qualitative physics seeks to understand physical processes by building models of them. A model is an abstract representation that eliminates irrelevant details. For example, if we want to predict the motion of a ball, we may want to consider its mass and velocity, but probably not its color. Traditional physical models are built up from real-valued variables, rates of change, expressions, equations, and states. Qualitative physics provides similar building blocks, ones which are more abstract and nonnumeric.

- **Variables**—In traditional physics, real-valued variables are used to represent features of objects, such as position, velocity, angle, and temperature. Qualitative physics retains this notion, but restricts each variable to a small finite set of possible values. For example, the amount of water in a pot might be represented as one of {*empty, between, full*}, and its temperature as {*frozen, between, boiling*}.
- **Quantity Spaces**—A small set of discrete values for a variable is called a *quantity space*. The elements of a quantity space are usually ordered with respect to each other so that one value can be said to be smaller than another.¹
- **Rates of Change**—Variables take on different values at different times. A real-valued rate of change (dx/dt) can be modeled qualitatively with the quantity space {*decreasing, steady, increasing*}.
- **Expressions**—Variables can be combined to form expressions. Consider representing the volume of water in a glass as {*empty, between, full*}. If we pour the contents of one glass into another, how much water will the second glass contain? We can add two qualitative values with the following chart:

empty + *empty* = *empty*
empty + *between* = *between*
empty + *full* = *full*
between + *between* = {*between, full*}
between + *full* = *full* + *overflow*
full + *full* = *full* + *overflow*

Notice that qualitative addition differs from its quantitative counterpart, in part because the result of qualitative addition may be ambiguous. For example, if both

¹In some variations [Raiman, 1986], it is possible to state that one value is much larger than another, or that two values are unequal but very close to one another.

glasses are between empty and full, it is impossible to know whether combining them will result in a full glass or not.

- **Equations**—Expressions and variables can be linked to one another via equations. The simplest equation states that variable x increases as variable y increases. This gives us an abstract representation of the actual function relating x and y (it may be linear, quadratic, or logarithmic, for example).
- **States**—Traditional physics models a process as a set of variables whose values evolve over time. A state is a single snapshot in which each variable possesses one value. Within qualitative physics, there are several different ways of formulating state information. One idea [de Kleer, 1979] is to combine qualitative state variables with symbolic descriptions. For example, the state of Figure 19.1(a) might be represented as (BALL-1, IN-AIR, DOWN). In order to predict the behavior of devices, de Kleer and Brown [1984] represent a state as a network of connected components. Forbus [1984] presents a state organization centered on processes and their influences.

19.1.2 Reasoning with Qualitative Information

No matter how states are represented, we need some way to reason about the information contained in them. A common reasoning method in qualitative physics is called *qualitative simulation* [Koipers, 1986]. The idea is to construct a sequence of discrete "episodes" that occur as qualitative variables change values. States are linked to other states by qualitative rules. Some rules are very general. For example, one simulation rule states that variables reach closer values before reaching further ones, and another rule states that changing from one value to another consumes some finite amount of time. Other rules, such as the rules governing the motion of objects through the air, are more specific.

In systems that contain more than one object, rules must apply to all objects simultaneously. For example, consider an electrical device with many components. Because the components are connected, they influence one another. The constraint satisfaction technique (Chapters 3 and 14) is one efficient way of propagating a change in one component to other nearby components.

Since combining qualitative values can lead to ambiguity, a qualitative simulation must sometimes split into two or more possible paths. A network of all possible states and transitions for a qualitative system is called an *envisionment*. Figure 19.2 shows an envisionment of the bouncing ball system of Figure 19.1(a). This network allows a computer to reason about the behavior of the ball without recourse to numerical simulation. There are often many paths through an envisionment. Each path is called a *history*.

Envisionments are useful in a number of applications. Most importantly, envisionments provide explanations for physical systems, and those explanations can be used to predict future behavior. In addition, if a system is an artificial one, such as a mechanical device, envisionments can be used to diagnose problems that occur when components fail to behave correctly. Envisionments can also be used to represent and/or repair inaccurate mental models that people may have. For more information about envisionments and qualitative simulation, see Weld and de Kleer [1988].

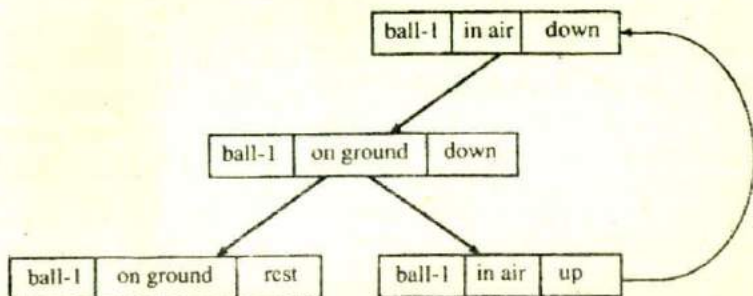


Figure 19.2: An Envisionment

In order to write programs that automatically construct envisionments, we must represent qualitative knowledge about the behavior of particular kinds of processes, substances, spaces, devices, and so on. In the next section, we look at how to codify some of this knowledge.

19.2 Commonsense Ontologies

A computer program that interacts with the real world must be able to reason about things like time, space, and materials. As fundamental and commonsensical as these concepts may be, modeling them turns out to present some thorny problems.

19.2.1 Time

While physicists and philosophers still debate the true nature of time, we all manage to get by on a few basic commonsense notions. These notions help us to decide when to initiate actions, how to reason about others' actions, and how to determine relationships between events. For instance, if we know that the Franco-Prussian War preceded World War I and that the Battle of Verdun occurred during World War I, then we can easily infer that the Battle of Verdun must have occurred sometime after the Franco-Prussian War. A commonsense theory of time must account for reasoning of this kind.

The most basic notion of time is that it is occupied by events. These events occur during intervals, continuous spaces of time. What kinds of things might we want to say about an interval? An interval has a starting point and an ending point, and a duration defined by these points. Intervals can be related to other intervals, as we saw in the last paragraph. It turns out that there are exactly thirteen ways in which two non-empty time intervals can relate to one another. Figure 19.3 shows these relationships. As is clear from the figure, there are actually only seven distinct relationships: the relationship of equality plus six other relationships that have their own inverses.

Now we can state rules for drawing inferences about time intervals. For example, common sense tells us that the IS-BEFORE relation is transitive. That is, if event a occurred before event b and if event b occurred before event c , then event a must have

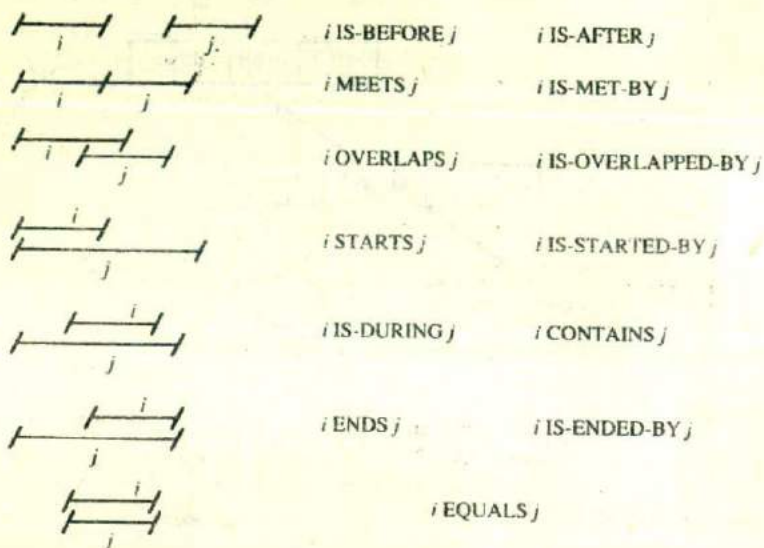


Figure 19.3: Thirteen Possible Relationships between Two Time Intervals

occurred before event *c*. How many such axioms will we need before we capture all of our basic commonsense notions of time? We can greatly simplify matters if we define some interval relationships in terms of other more basic ones. In fact, we can reduce all the relations in Figure 19.3 to the single relation MEETS. Here is the definition of the relation IS-BEFORE:

$$i \text{ IS-BEFORE } j \equiv \exists k : (i \text{ MEETS } k) \wedge (k \text{ MEETS } j)$$

In other words, if *i* IS-BEFORE *j*, then there must be some *k* in between that MEETS both *i* and *j*. When the rest of the relations are defined similarly, MEETS becomes the only primitive relation, and we can write all our commonsense axioms in terms of it. Our first axiom states that points where intervals MEET are unique:

$$\forall i, j : (\exists k : (i \text{ MEETS } k) \wedge (j \text{ MEETS } k)) \rightarrow \\ (\forall l : (i \text{ MEETS } l) \leftrightarrow (j \text{ MEETS } l))$$

In other words, *i* and *j* cannot MEET *k* at different points in time, so every event has a unique starting time. We can write a similar axiom to state that every event has a unique ending time. Next, we state that given two places where intervals meet, exactly one of the following three conditions must hold: the places are the same, the first place precedes the second, or the second precedes the first.²

²In this formula \oplus should be read as "exclusive-or." $p \oplus q \oplus r$ is logical shorthand for $(p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r)$.

$$\begin{aligned} \forall i, j, k, l: & (i \text{ MEETS } j) \wedge (k \text{ MEETS } l) \rightarrow \\ & (i \text{ MEETS } l) \oplus \\ & \exists m : (i \text{ MEETS } m) \wedge (m \text{ MEETS } l) \oplus \\ & \exists m : (k \text{ MEETS } m) \wedge (m \text{ MEETS } j) \end{aligned}$$

There are two more axioms. One states that there are always intervals surrounding any given interval. This axiom turns out to be useful, although it prohibits any reasoning about infinite time intervals.

$$\forall i : \exists j, k : (j \text{ MEETS } i) \wedge (i \text{ MEETS } k)$$

Finally, we can state that for any two intervals that MEET, there exists a continuous interval that is the union of the two:

$$\begin{aligned} \forall i, j: & (i \text{ MEETS } j) \rightarrow \\ & \exists a, b, (i + j) : \\ & (a \text{ MEETS } i) \wedge (j \text{ MEETS } b) \wedge \\ & (a \text{ MEETS } (i + j)) \wedge ((i + j) \text{ MEETS } b) \end{aligned}$$

These axioms encode a rich commonsense theory of time. They allow us to derive many facts, such as the transitivity of the IS-BEFORE relation. Suppose we know that a IS-BEFORE b and that b IS-BEFORE c . By the definition of IS-BEFORE, there must be some interval d that lies between a and b , i.e., a MEETS d and d MEETS b . By the union axiom, we can deduce the existence of an interval $(d + b)$ such that there is an x that MEETS $(d + b)$ and a y that IS-MET-BY $(d + b)$. By the uniqueness of starting points, we can conclude that a also meets $(d + b)$. Since b IS-BEFORE c , there must be an e between them. We can now construct another union interval $(d + b + e)$, which we can prove MEETS c and IS-MET-BY a . Therefore a IS-BEFORE c .

This may seem like a roundabout way of doing things, and it is. There is nothing in the axioms themselves that dictates how they should be used in real programs. In fact, efficient implementations represent all thirteen temporal relations explicitly, making use of precompiled tables that record how the relations can interact. Constraint satisfaction is a useful technique for making inferences about these relations [Kautz, 1986]. The logical statements above are just a concise way of writing down one particular commonsense theory of time.

19.2.2 Space

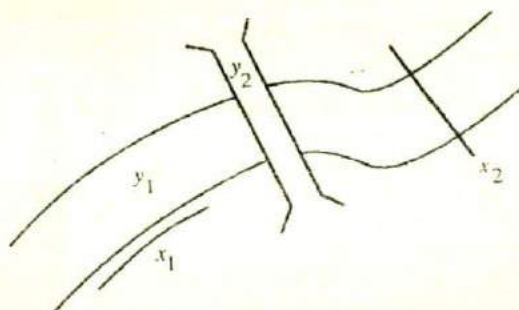
In this book, we have often used examples from the blocks world. Primitives in this world include block names, actions like PICKUP and STACK, and predicates like ON(x, y). These primitives constitute a useful abstraction, but eventually we must break them down. If we want a real robot to achieve ON(x, y), then that robot had better know what ON really means, where x and y are located, how big they are, how they are shaped, how to align x on top of y so that x won't fall off, and so forth. These requirements become more apparent if we want to issue commands like "place block x near block y " or "lean block x up against block y ". Commonsense notions of space are critical for living in the real world.

Objects have spatial extent, while events have temporal extent. We might therefore try to expand our commonsense theory of time into a commonsense theory of space. Because space is three-dimensional, there are far more than thirteen possible spatial relationships between two objects. For instance, consider one block perfectly aligned on top of another. The objects are EQUAL in the length and width dimensions, while they MEET in the height dimension. If the top block is smaller than the bottom one but still centered on top of it, then they still MEET in the height dimension, but we must use the spatial equivalent of IS-DURING to describe the length and width relationships. The main problem with this approach is that it generates a vast number of relations (namely $13^3 = 2197$), many of which are not very commonsensical. Moreover, a number of interesting spatial relations, such as "x curves around y," are not included. So we must consider another approach.

In our discussion of qualitative physics, we saw how to build abstract models by transforming real-valued variables into discrete quantity spaces. We can also view objects and spaces at various levels of abstraction. For instance, we can view a three-dimensional piece of paper as a two-dimensional sheet; similarly, we can view a three-dimensional highway as a one-dimensional curve. Hobbs [1985] proposed one very general mechanism for creating and manipulating abstract models. With this mechanism, we start out with a full-blown theory of the world, and then we construct a simpler, more abstract model by extracting a set of *relevant properties*. We then group objects into classes whose members are indistinguishable from each other as far as their relevant properties go. For example, as we drive along a highway, our major relevant property might be DISTANCE-TO-GOAL. This property effectively reduces the bits of concrete in the three-dimensional highway into a one-dimensional curve, where each point on the curve has a unique DISTANCE-TO-GOAL value. In a similar fashion, we can map real time intervals onto discrete time steps, spatial coordinates onto a two-dimensional grid, and so on. Choosing a set of relevant properties amounts to viewing the world at a particular level of *granularity*. Since different granularities are systematically related to each other, we can reason in a simplified model with relative assurance that our actions will be implementable in the real world.

The idea of granularity can be used to build a commonsense model of space [Kautz, 1985]. The basic idea is to define relations over spaces. The first relation is *INSIDE*(x, y, g), where x and y are spaces occupied by particular objects and g is the level of granularity at which those objects are viewed. For example, water is *INSIDE* a glass if the three-dimensional space taken up by the water is completely contained within the three-dimensional space taken up by the glass. If we view a highway as a three-dimensional slab of concrete, then a car driving along the highway would be considered *ADJACENT* to the highway, but not *INSIDE* of it. However, if some granularity g views the highway as a one-dimensional curve, then the relation *INSIDE*(*Car*, *Highway*, g) holds for as long as the car stays on the road. This is because the car and its position on the road are indistinguishable at that level of granularity.

We can now define a number of useful properties for curves, lines, surfaces, planes, and volumes. For example, here is the definition of a terminal point p of a curve c :

Figure 19.4: Two Ribbons (y_1 and y_2) and Two Curves (x_1 and x_2)

$$\begin{aligned} \text{TERMINAL}(p, c) \equiv & \\ & \text{INSIDE}(p, c) \wedge \\ & \forall c_1, c_2 : \text{INSIDE}(c_1, c) \wedge \text{INSIDE}(c_2, c) \\ & \quad \wedge \text{INSIDE}(p, c_1) \wedge \text{INSIDE}(p, c_2) \\ & \quad \rightarrow \text{INSIDE}(c_1, c_2) \vee \text{INSIDE}(c_2, c_1) \end{aligned}$$

In other words, p is a **TERMINAL** of c if, whenever two subcurves of c both include p , one must be a subcurve of the other. We can similarly define curve segments, adjoining curves, loops, and forks. Another useful class to define is that of a **RIBBON**:

$$\text{RIBBON}(\text{object}, \text{side}_1, \text{side}_2)$$

A ribbon is essentially a curve viewed at a coarser level of granularity, resulting in a two dimensional ribbonlike shape. Our world contains many objects that are usefully viewed as ribbons, e.g., rivers and bridges (Figure 19.4). We can define several properties of curves as they relate to ribbons. For example,

$$\begin{aligned} \text{ALONG}(x, y) & \equiv \text{CURVE}(x) \wedge \text{RIBBON}(y, s_1, s_2) \wedge \\ & \quad \forall z : \text{INSIDE}(z, x) \rightarrow \text{ADJACENT}(z, y) \\ \\ \text{ACROSS}(x, y) & \equiv \text{CURVE}(x) \wedge \text{RIBBON}(y, s_1, s_2) \wedge \\ & \quad \text{PERPENDICULAR}(x, \text{AXIS}(y, s_1, s_2)) \wedge \\ & \quad \text{ADJACENT}(x, y) \wedge \text{ADJACENT}(x, s_1) \wedge \\ & \quad \text{ADJACENT}(x, s_2) \end{aligned}$$

These definitions assume that we have defined the terms **PERPENDICULAR**, **AXIS**, and **ADJACENT**, and that we have supplied the commonsense axiom that an object x is **ADJACENT** to an object y if any part of x is **ADJACENT** to y .

A robot could use the **ALONG** relation to plot a course down the river's edge. It could similarly use the **ACROSS** relation to navigate to the other side of the river. Unfortunately, the **ACROSS** relation is not enough, as the robot might try to cross the river without using the bridge. The robot is still missing one fact: you can't walk on water. That's common sense

19.2.3 Materials

Why can't you walk on water? What happens if you turn a glass of water upside down? What happens when you pour water into the soil of a potted plant?

Liquids present a particularly interesting and challenging domain to formalize. Hayes [1985] presented one attempt to describe them. Before we can write down any properties of liquids, we must decide what kinds of objects those properties will describe. In the last section, we defined spatial relations in terms of the spaces occupied by objects, not in terms of the objects themselves. It is particularly useful to take this point of view with liquids, since liquid "objects" can be split and merged so easily. For example, if we consider a river to be a piece of liquid, then what happens to the river when the liquid flows out into the ocean? Instead of continually changing our characterization of the river, it is more convenient to view the river as a fixed space occupied by water.

Containers play an important role in the world of liquids. Since we do not want to refer to liquid objects, we must have another way of stating how much liquid is in a container. We can define a CAPACITY function to bound the amount of liquid l that a space s can hold. The space is FULL when the AMOUNT equals the CAPACITY.

$$\begin{aligned} \text{CAPACITY}(s) &\geq \text{AMOUNT}(l, s) > \text{none} \\ \text{FULL}(s) &\equiv \text{AMOUNT}(l, s) = \text{CAPACITY}(s) \end{aligned}$$

We can also define an AMOUNT function:

$$\text{AMOUNT}(\text{Water}, \text{Glass}) > \text{none}$$

This statement means, "There is water in the glass." Here, *Water* refers to the generic concept of water and *Glass* refers to the space enclosed by a particular glass.

Spaces have a number of other properties besides CAPACITY and FULL. Recall that spaces can be linked to one another by the INSIDE relation. In addition, a space can be free or not. A space is free if it is not wholly contained inside a solid object. In addition, every space is bounded on all sides by a set of two-dimensional regions, called faces. If a free face (one not part of a solid object) separates two free spaces, it is called a portal. Liquids can flow from one free space to another via a portal. Two objects are said to be joined if they share a common face. To summarize:

$$\begin{aligned} \text{FREE}(s) &\equiv \neg \exists o : \text{SOLID}(o) \wedge \text{INSIDE}(s, o) \\ \text{FACE}(f, s) &\equiv f \text{ is some 2-D bounding region of } s \\ \text{PORTAL}(f) &\equiv \exists s_1, s_2 : \text{FACE}(f, s_1) \wedge \text{FACE}(f, s_2) \wedge \\ &\quad \text{FREE}(s_1) \wedge \text{FREE}(s_2) \wedge \text{FREE}(f) \\ \text{JOINED}(o_1, o_2, f) &\equiv \text{FACE}(f, o_1) \wedge \text{FACE}(f, o_2) \end{aligned}$$

We can now define a closed container as a hollow object with no portals:

$$\begin{aligned} \text{CLOSED-CONTAINER}(c) &\equiv \\ &(\exists s : \text{INSIDE}(s, c) \wedge \text{FREE}(s)) \wedge \\ &(\neg \exists s, f : \text{INSIDE}(s, c) \wedge \text{JOINED}(s, c, f) \wedge \neg \text{PORTAL}(f)) \end{aligned}$$

An open container has (at least) one portal at the top:

OPEN-CONTAINER(c) \equiv

$(\exists s : \text{INSIDE}(s, c) \wedge \text{FREE}(s)) \wedge$

$(\forall s, f : \text{INSIDE}(s, c) \wedge \text{JOINED}(s, c, f) \rightarrow$

$f = \text{TOP}(c) \wedge \neg \text{PORTAL}(f))$

Liquids make things wet. To model wetness, we will find it useful to imagine a solid object as being surrounded by a very thin free space. This space is broken up into a set of thin *outer* spaces corresponding to the various faces of the object. Objects that touch share these outer spaces.

SURROUND(o) \equiv thin space surrounding object o

$\forall o : \text{FREE}(\text{SURROUND}(o))$

OUTER(d, o) $\equiv \exists f : \text{FACE}(f, o)$ and d is the thin free space just outside f

TOUCHING(o_1, o_2) $\equiv \exists d : \text{OUTER}(d, o_1) \wedge \text{OUTER}(d, o_2)$

$\forall d, o : \text{OUTER}(d, o) \rightarrow \text{INSIDE}(d, \text{SURROUND}(o))$

$\forall s, o : \text{FREE}(s) \wedge \text{INSIDE}(o, s) \rightarrow \text{INSIDE}(\text{SURROUND}(o), s)$

The last two facts state that SURROUND(o) contains all its outer spaces, and that any larger, free space containing object o also contains SURROUND(o). Now we can define wetness as a relation between an outer space d and some generic liquid l :

WET-BY(d, l) $\equiv \text{CAPACITY}(d) \geq \text{AMOUNT}(l, d) > \text{none}$

IS-WET(o) $\equiv \exists d, l : \text{OUTER}(d, o) \wedge \text{WET-BY}(d, l)$

IS-WET-ALL-OVER(o) $\equiv \forall d : \text{OUTER}(d, o) \rightarrow \exists l : \text{WET-BY}(d, l)$

Suppose our robot encounters a room with six inches of water on the floor. What will happen if the robot touches the floor? By the definition of TOUCHING, we have:

$\exists d_1 : \text{OUTER}(d_1, \text{Robot}) \wedge \text{OUTER}(d_1, \text{Floor})$

Since the floor only has one face, d , we can conclude:

$\text{OUTER}(d, \text{Robot}) \wedge \text{OUTER}(d, \text{Floor})$

Combining the first clause with the fact WET-BY(d, Water) gives us IS-WET(Robot). In other words, the robot will get wet. Recall that at the end of the last section, our robot was about to try crossing a river without using a bridge. It might find this fact useful:

$\text{INSIDE}(s_1, s_2) \wedge \text{FREE}(s_1) \wedge \text{FULL}(s_2, l) \rightarrow \text{FULL}(s_1, l)$

It is straightforward to show that if the robot is submerged, it will be wet all over. Predicting that the robot will become submerged in the first place requires some envisionment. We need a rule that says one dense solid object must be supported by another solid object, or else it will tend to move downward. One property of liquids is that they do not support dense solid objects.

We also need general rules describing how liquids themselves behave over time. Consider all the possible forms that a liquid may take at a given instant. Hayes (1985)

distinguishes between "lazy, still" liquids, "lazy, moving" liquids and "energetic, moving" liquids. Energetic, moving liquids are liquids being propelled by some active force, for example, oil being pumped through a pipeline. Lazy liquids are liquids in their natural state. Sometimes they are moving, as in river water and rain, and sometimes they are still. Liquids in any of these forms can also be either bulk or divided. Most of the time we deal with bulk liquid, but sometimes we encounter mist, dew, or rain. Finally, liquids can be either unsupported, on a surface, or in a container.

What happens to these types of liquids? Figure 19.5 shows five envisionments for lazy, bulk liquids. A containment event can become a falling event if the container tips. The falling event becomes a wetting event and then a spreading one. Depending on where the spreading takes place, further falling or flowing events may ensue. When all the liquid has left the container, the spreading will stop, and sometime afterward, a drying event will begin.

Other materials behave differently. Solids can be rigid or flexible. A string can be used to pull an object but not to push it. Solids can also be particulate (like sand), in which case they share many of the same behaviors as liquids. Gases are also similar to liquids. Also, some solids soak up liquid (sponges, dirt), while others are watertight.

We can see that commonsense knowledge representation has a strongly taxonomic flavor. A lot of work has been done in these and other areas, but much more also remains to be worked out.

19.3 Memory Organization

Memory is central to commonsense behavior. Human memory contains an immense amount of knowledge about the world. So far, we have only discussed a tiny fraction of that knowledge. Memory is also the basis for learning. A system that cannot learn cannot, in practice, possess common sense.

A complete theory of human memory has not yet been discovered, but we do have a number of facts at our disposal. Some of these facts come from neurobiology (e.g., [Kandel and Schwartz, 1985]), while others are psychological in nature. Computer models of neural memory (such as the Hopfield network of Chapter 18) are interesting, but they do not serve as theories about how memory is used in everyday, commonsense reasoning. Psychology and AI seek to address these issues.

Psychological studies suggest several distinctions in human memory. One distinction is between short-term memory (STM) and long-term memory (LTM). We know that a person can only hold a few items at a time in STM, but the capacity of LTM is very large. LTM storage is also fairly permanent. The production system is one computer model of the STM-LTM structure. Perceptual information is stored directly in STM, also called working memory. Production rules, stored in LTM, match themselves against items in STM. Productions fire, modify STM, and repeat.

LTM is often divided into *episodic memory* and *semantic memory*. Episodic memory contains information about past personal experiences, usually stored from an autobiographical point of view. For example, a college graduation, a wedding, or a concert may all form episodic memories. Semantic memory, on the other hand, contains facts like "Birds fly." These facts are no longer connected with personal experiences.

Work on modeling semantic memory began with Quillian [1969]. This model soon

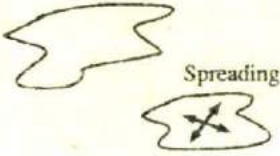
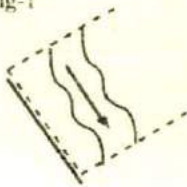



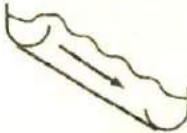
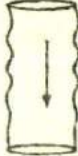
Still liquids	Moving liquids	
<p>Wetting</p>  <p>Spreading</p>	<p>Flowing-1</p> 	Supported
<p>Containment</p>  <p>Emptying</p>  <p>Filling</p> 	<p>Flowing-2</p> 	Contained
<p>(Impossible)</p>	<p>Falling</p> 	Unsupported

Figure 19.5: Five Environments for Lazy, Bulk Liquids

developed into the idea of semantic networks and from there into the other slot-and-filler structures we saw in Chapters 9 and 10. Semantic memory is especially useful in programs that understand natural language.

Models for episodic memory grew out of research on scripts. Recall that a script is a stereotyped sequence of events, such as those involved in going to the dentist. One obvious question to ask is: How are scripts acquired? Surely they are acquired through personal experience. But a particular experience often includes details that we do not want to include in a script. For example, just because we once saw *The New Yorker* magazine in a dentist's waiting room, that doesn't mean that *The New Yorker* should be part of the dentist script. The problem is that if a script contains too many details, it will not be matched and retrieved correctly when new, similar situations arise.

In general, it is difficult to know which script to retrieve (as we discussed in Section 4.3.5). One reason for this is that scripts are too monolithic. It is hard to do any kind of partial matching. It is also hard to modify a script. More recent work reduces scripts to individual *scenes*, which can be shared across multiple structures. Stereotypical sequences of scenes are strung together into memory organization packets (MOPs) [Schank, 1977]. Usually, three distinct MOPs encode knowledge about an event sequence. One MOP represents the physical sequence of events, such as entering a dentist's office, sitting in the waiting room, reading a magazine, sitting in the dentist's chair, etc. Another MOP represents the set of social events that take place. These are events that involve personal interactions. A third MOP revolves around the goals of the person in the particular episode. Any of these MOPs may be important for understanding new situations.

MOPs organize scenes, and they themselves are further organized into higher-level MOPs. For example, the MOP for visiting the office of a professional may contain a sequence of abstract general scenes, such as talking to an assistant, waiting, and meeting. High-level MOPs contain no actual memories, so where do they come from?

New MOPs are created upon the failure of expectations. When we use scripts for story understanding, we are able to locate interesting parts of the story by noticing places where events do not conform to the script's expectations. In a MOP-based system, if an expectation is repeatedly violated, then the MOP is generalized or split. Eventually, episodic memories can fade away, leaving only a set of generalized MOPs. These MOPs look something like scripts, except that they share scenes with one another.

Let's look at an example. The first time you go to the dentist, you must determine how things work from scratch since you have no prior experience. In doing so, you store detailed accounts of each scene and string them together into a MOP. The next time you visit the dentist, that MOP provides certain expectations, which are mostly met. You are able to deal with the situation easily and make inferences that you could not make the first time. If any expectation fails, this provides grounds for modifying the MOP. Now, suppose you later visit a doctor's office. As you begin to store episodic scenes, you notice similarities between these scenes and scenes from the dentist MOP. Such similarities provide a basis for using the dentist MOP to generate expectations. Multiple trips to the doctor will result in a doctor MOP that is slightly different from the dentist MOP. Later experiences with visiting lawyers and government officials will result in other MOPs. Ultimately, the structures shared by all of these MOPs will cause a generalized MOP to appear. Whenever you visit a professional's office in the future, you can use the generalized MOP to provide expectations.

With MOPs, memory is both a constructive and reconstructive process. It is constructive because new experiences create new memory structures. It is reconstructive because even if the details of a particular episode are lost, the MOP provides information about what was likely to have happened. The ability to do this kind of reconstruction is an important feature of human memory.

There are several MOP-based computer programs. CYRUS [Kolodner, 1984] is a program that contains episodes taken from the life of a particular individual. CYRUS can answer questions that require significant amounts of memory reconstruction. The IPP program [Lebowitz, 1983] accepts stories about terrorist attacks and stores them in an episodic memory. As it notices similarities in the stories, it creates general memory structures. These structures improve its ability to understand. MOPTRANS [Lytinen, 1984] uses a MOP-based memory to understand sentences in one language and translate them into another.

19.4 Case-Based Reasoning

We now turn to the role of memory in general problem solving. Most AI programs solve problems by reasoning from first principles. They can explain their reasoning by reporting the string of deductions that led from the input data to the conclusion. With human experts, however, we often observe a different type of explanation. An expert encountering a new problem is usually reminded of similar cases seen in the past, remembering the results of those cases and perhaps the reasoning behind those results. New problems are solved by analogy with old ones and the explanations are often couched in terms of prior experiences. Medical expertise, for example, seems to follow this pattern, and legal education is also case-oriented.

Computer systems that solve new problems by analogy with old ones are often called *case-based reasoning* (CBR) systems. A CBR system draws its power from a large case library, rather than from a set of first principles. In order to be successful, CBR systems must answer the following questions:

1. How are cases organized in memory?
2. How are relevant cases retrieved from memory?
3. How can previous cases be adapted to new problems?
4. How are cases originally acquired?

The memory structures we discussed in the previous section are clearly relevant to CBR. Those structures were used primarily in text understanding applications, however. Now we look at general memory-based problem solving.

To use a memory effectively, we must have a rich indexing mechanism. When we are presented with a problem, we should be reminded of relevant past experiences, but not be inundated with a lot of useless memories. The obvious idea is to index past episodes by the features present in them. For example, any experience having to do with a car would be filed under *Car*, as well as under other indices. But we must have some scheme for distinguishing important indices from unimportant ones. Otherwise, everything will remind us of everything else, and we will be unable to focus on memories

that will best help us to solve our current problem. But important features are not always the most obvious ones. Here is an example from Schank [1977], called the "steak and haircut" story:

X described how his wife would never cook his steak as rare as he liked it. When X told this to Y, Y was reminded of a time, 30 years earlier, when he tried to get his hair cut in England and the barber just wouldn't cut it as short as he wanted it.

Clearly, the indices *Steak*, *Wife*, and *Rare* are insufficient to remind Y of the barbershop episode. We need more general indices, such as *Provide-Service*, *Refusal*, and *Extreme*. Dyer [1983] also takes up this theme, embodied in a program that deduces adages and morals from narratives.

Some features are only important in certain contexts. For example, suppose it is cloudy. If your problem is to plan a picnic, you might want to retrieve other episodes involving cloudy days. But if your problem is to write a computer program, then the fact that it is cloudy is probably incidental. Because important features vary from domain to domain, a general CBR system must be able to learn a proper set of indices from experience. Both the inductive and explanation-based learning techniques described in Chapter 17 have been used for this task.

Recall that in our discussion of production systems, we talked about how rules and states could be organized into a RETE network for efficient matching. We also discussed matching frames and scripts in Section 4.3.5. Something similar is required for CBR, since the number of cases can be very large. The data structure for the case itself is also important. A case is usually stored as a monolithic structure, although in some variations, cases can be stored piecemeal. The former strategy is efficient when it is possible to obtain almost-perfect matches; the latter strategy is better in complex problem-solving domains.

The result of the retrieval process is usually a set of cases. The next step is to take the best case and adapt it to the current situation. One method for choosing the best case is the use of *preference heuristics* [Kolodner, 1989]. Here are some examples:

- **Goal-Directed Preference**—Prefer cases that involve the same goal as the current situation.
- **Salient-Feature Preference**—Prefer cases that match the most important features, or those that match the largest number of important features.
- **Specificity Preference**—Prefer cases that match features exactly over those that match features generally.
- **Frequency Preference**—Prefer frequently matched cases.
- **Recency Preference**—Prefer recently matched cases.
- **Ease-of-Adaptation Preference**—Prefer cases with features that are easily adapted to new situations.

Since even the best case will not match the current situation exactly, it will have to be adapted. At the simplest level, this involves mapping new objects onto old ones (e.g., *Steak* onto *Hair*, and *Rare* onto *Short*). When old cases represent entire problem-solving episodes, adaptation can be quite complex. CHEF [Hammond, 1986] is an example of a case-based planner, a program whose cases are actually complete plans for solving problems in the domain of cooking. CHEF's case library is augmented with a plan-modification library indexed by plan types and change types. CHEF first looks at the retrieved plan and sees if it satisfies the current goals. If any goal is unsatisfied, then the plan-modification library is consulted. The library may suggest a list of steps to be added to the plan, deleted from the plan, or substituted for existing steps. This modification process is not guaranteed to succeed, however, and so CHEF includes a plan repair module that uses domain knowledge to explain why the new plan fails, if it does. Once a complete, working plan is created, it is executed and then stored in the case library for future reference.

We have said nothing yet about how cases are acquired originally. In fact, most CBR systems draw on a small library of cases that are entered by hand. Of course, we will eventually be able to transform large bodies of on-line texts, such as legal cases, into large case libraries. Another approach is to bootstrap gradually from rule-based search into CBR. The idea is to start solving problems with a heuristic search engine. Each time a problem is solved, it is automatically stored in a case library. As the library grows, it becomes possible to solve some new problems by analogy with old ones. This idea is very similar to some of the learning techniques we saw in Section 17.4—the acquisition of search control rules, for example. This brings up the issue of whether it is better to store whole cases in memory or to store smaller bits of control knowledge instead. There are a number of trade-offs involved. First is the ease of modification. Central to case-based reasoning is the idea that stored cases can be adapted and modified. Search control rules are more procedural. Once learned, they are hard to modify. If a search control rule starts to perform badly, it is usually deleted in toto. Another trade-off involves indexing. Search control rules are fully indexed: they apply in exactly the situations to which they are relevant. Cases, on the other hand, are usually indexed heuristically, as we saw above. Finally, search control rules are explicitly generalized at storage time. In CBR, generalization occurs over time as a by-product of the retrieval and adaptation process. Aggressive generalization makes it easy to solve new problems quickly, but in less complete domains, where proper generalizations are unknown, an aggressive strategy can be inefficient and even incorrect.

19.5 Exercises

1. Consider a toy balloon hooked up to a bottle of compressed air. As the air is released, the balloon expands. Using qualitative measures, list the quantity spaces of variables and rates of change in this system. Construct an environment for the system, and write down one possible history.
2. Express all the temporal relations in Figure 19.3 in terms of the single relation MEETS.
3. Suppose you know the following facts:

- The Franco-Prussian War took place before World War I.
- The Battle of Verdun took place during World War I.

Convert these facts into logical statements in terms of the MEETS relation. Use the commonsense axioms of time given in Section 19.2.1 to show that the Franco-Prussian War must have occurred before the Battle of Verdun.

4. Using the axioms in Section 19.2.2, show that a robot submerged under water will be wet all over.
5. Case-based reasoning shares many of the same ideas of learning by analogy (Section 17.8). Briefly discuss how transformational and derivational analogy could apply in case-based reasoning systems.
6. Forgetting is one aspect of human memory that is not usually modeled in computer systems. Under what circumstances might a case-based reasoning system benefit from the ability to forget?

Chapter 20

Expert Systems

Expert systems solve problems (such as the ones in Figure 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, 1984] (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: POOL 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. R1, 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 Figure 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.

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

There is strongly suggestive evidence (.8) that the category of ORGANISM-1 is enterobacteriaceae

{ALSO: there is weakly suggestive evidence (.1) that the identity of ORGANISM-1 is pseudomonas-aeruginosa}

{RULE037}

** HOW [2.3]

[i.e., HOW was it established that the aerobicity of ORGANISM-1 is facultative?]

The following were used:

[3.1] RULE027 indicated there is strongly suggestive evidence (.8) that the aerobicity of ORGANISM-1 is facultative.

[3.2] RULE033 indicated there is weakly suggestive evidence (.2) that the aerobicity of ORGANISM-1 is not facultative.

Since this gave a cumulative CF of (.6) for facultative, the evidence indicates that the aerobicity of ORGANISM-1 is facultative.

Figure 20.1: A Portion of a Dialogue with TEIRESIAS

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 Figure 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].)

When TEIRESIAS provides the answer to the first of these questions, the user may be satisfied or may want to follow the reasoning process back even further. The user can do that by asking additional "WHY" questions.

When TEIRESIAS provides the answer to the second of these questions and tells the user what it already believes, the user may want to know the basis for those beliefs. The user can ask this with a "HOW" question, which TEIRESIAS will interpret as "How did you know that?" This question also can be answered by looking at the goal tree and chaining backward from the stated fact to the evidence that allowed a rule that determined the fact to fire. Thus we see that by reasoning backward from its top-level goal and by keeping track of the entire tree that it traverses in the process, TEIRESIAS-MYCIN can do a fairly good job of justifying its reasoning to a human user. For more details of this process, as well as a discussion of some of its limitations, see Davis [1982].

The production system model is very general, and without some restrictions, it is hard to support all the kinds of explanations that a human might want. If we focus on a particular type of problem solving, we can ask more probing questions. For example, SALT [Marcus and McDermott, 1989] is a knowledge acquisition program used to build expert systems that design artifacts through a *propose-and-revise* strategy. SALT is capable of answering questions like WHY-NOT ("why didn't you assign value *x* to this parameter?") and WHAT IF ("what would happen if you did?"). A human might ask these questions in order to locate incorrect or missing knowledge in the system as a precursor to correcting it. We now turn to ways in which a program such as SALT can support the process of building and refining knowledge.

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.

Statistical techniques, such as multivariate analysis, provide an alternative approach to building expert-level systems. Unfortunately, statistical methods do not produce concise rules that humans can understand. Therefore it is difficult for them to explain their decisions.

For highly structured problems that require deep causal chains of reasoning, learning techniques are presently inadequate. There is, however, a great deal of research activity

in this area, as we saw in Chapter 17.

20.5 Summary

Since the mid-1960s, when work began on the earliest of what are now called expert systems, much progress has been made in the construction of such programs. Experience gained in these efforts suggests the following conclusions:

- These systems derive their power from a great deal of domain-specific knowledge, rather than from a single powerful technique.
- In successful systems, the required knowledge is about a particular area and is well defined. This contrasts with the kind of broad, hard-to-define knowledge that we call common sense. It is easier to build expert systems than ones with common sense.
- An expert system is usually built with the aid of one or more experts, who must be willing to spend a great deal of effort transferring their expertise to the system.
- Transfer of knowledge takes place gradually through many interactions between the expert and the system. The expert will never get the knowledge right or complete the first time.
- The amount of knowledge that is required depends on the task. It may range from forty rules to thousands.
- The choice of control structure for a particular system depends on specific characteristics of the system.
- It is possible to extract the nondomain-specific parts from existing expert systems and use them as tools for building new systems in new domains.

Four major problems facing current expert systems are:

- **Brittleness**—Because expert systems only have access to highly specific domain knowledge, they cannot fall back on more general knowledge when the need arises. For example, suppose that we make a mistake in entering data for a medical expert system, and we describe a patient who is 130 years old and weighs 40 pounds. Most systems would not be able to guess that we may have reversed the two fields since the values aren't very plausible. The CYC system, which we discussed in Section 10.3, represents one attempt to remedy this problem by providing a substrate of commonsense knowledge on which specific expert systems can be built.
- **Lack of Meta-Knowledge**—Expert systems do not have very sophisticated knowledge about their own operation. They typically cannot reason about their own scope and limitations, making it even more difficult to deal with the brittleness problem.

- **Knowledge Acquisition**—Despite the development of the tools that we described in Section 20.4, acquisition still remains a major bottleneck in applying expert systems technology to new domains.
- **Validation**—Measuring the performance of an expert system is difficult because we do not know how to quantify the use of knowledge. Certainly it is impossible to present formal proofs of correctness for expert systems. One thing we can do is pit these systems against human experts on real-world problems. For example, MYCIN participated in a panel of experts in evaluating ten selected meningitis cases, scoring higher than any of its human competitors [Buchanan, 1982].

There are many issues in the design and implementation of expert systems that we have not covered. For example, there has been a substantial amount of work done in the area of real-time expert systems [Laffey *et al.*, 1988]. For more information on the whole area of expert systems and to get a better feel for the kinds of applications that exist, look at Weiss and Kulikowski [1984], Harmon and King [1985], Rauch-Hindin [1986], Waterman [1986], and Prerau [1990].

20.6 Exercises

1. Rule-based systems often contain rules with several conditions in their left sides.
 - (a) Why is this true in MYCIN?
 - (b) Why is this true in R1?
2. Contrast expert systems and neural networks (Chapter 18) in terms of knowledge representation, knowledge acquisition, and explanation. Give one domain in which the expert system approach would be more promising and one domain in which the neural network approach would be more promising.

Chapter 21

Perception and Action

In the first chapter of this book, we proposed a definition of AI based on the nature of the problems it tackles, namely those for which humans currently outperform computers. So far, we have discussed primarily cognitive tasks, but there are many other tasks that also fall within this realm. In basic perceptual and motor skills, even lower animals possess phenomenal capabilities compared to computers.

Perception involves interpreting sights, sounds, smells, and touch. Action includes the ability to navigate through the world and manipulate objects. In order to build robots that live in the world, we must come to understand these processes. Figure 21.1 shows a design for a complete autonomous robot. Most of AI is concerned only with cognition, the idea being that when intelligent programs are developed, we will simply add sensors and effectors to them. But problems in perception and action are substantial in their own right and are being tackled by researchers in the field of robotics.

In the past, robotics and AI have been largely independent endeavors, and they have developed different techniques to solve different problems. We attempt to characterize the field of robotics at the end of this chapter, but for now, we should note one key difference between AI programs and robots: While AI programs usually operate in computer-simulated worlds, robots must operate in the physical world. As an example, consider making a move in chess. An AI program can search millions of nodes in a game tree without ever having to sense or touch anything in the real world. A complete chess-playing robot, on the other hand, must be capable of grasping pieces, visually interpreting board positions, and carrying on a host of other actions.

The distinction between real and simulated worlds has several implications:

- The input to an AI program is symbolic in form, e.g., an 8-puzzle configuration or a typed English sentence. The input to a robot is typically an analog signal, such as a two-dimensional video image or a speech waveform.
- Robots require special hardware for perceiving and affecting the world, while AI programs require only general-purpose computers.
- Robot sensors are inaccurate, and their effectors are limited in precision. There is always some degree of uncertainty about exactly where the robot is located.

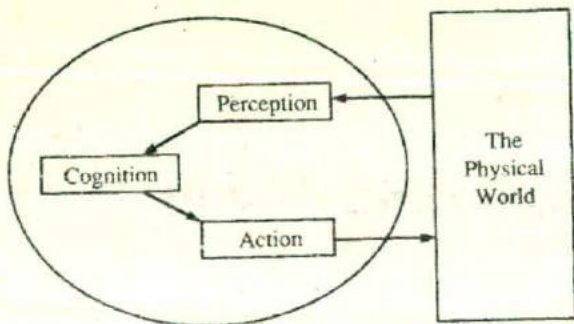


Figure 21.1: A Design for an Autonomous Robot

and where objects and obstacles stand in relation to it. Robot effectors are also limited in precision.

- Many robots must react in real time. A robot fighter plane, for example, cannot afford to search optimally or to stop monitoring the world during a LISP garbage collection.
- The real world is unpredictable, dynamic, and uncertain. A robot cannot hope to maintain a correct and complete description of the world. This means that a robot must consider the trade-off between devising and executing plans. This trade-off has several aspects. For one thing, a robot may not possess enough information about the world for it to do any useful planning. In that case, it must first engage in information-gathering activity. Furthermore, once it begins executing a plan, the robot must continually monitor the results of its actions. If the results are unexpected, then re-planning may be necessary.

Consider the problem of traveling across town. We might decide to take a bus, but without a bus schedule, it is impossible to complete the plan. So we make a plan for acquiring a schedule and execute it in the world. Now we can plan our route. The bus we want to take may be scheduled to arrive at 5:22 p.m., but the probability of it coming at exactly 5:22 p.m. is actually very small. We should stick to our plan and wait, even if the bus is late. After a while, if the bus still has not come, we must make a new plan.

- Because robots must operate in the real world, searching and backtracking can be costly. Consider the problem of moving furniture into a room. Operating in a simulated world with full information, an AI program can come up with an optimal plan by best-first search. Preconditions of operators can be checked quickly, and if an operator fails to apply, another can be tried. Checking preconditions in the real world, however, can be time-consuming if the robot does not have full information. For example, one operator may require that an object weigh less than fifty pounds. Navigating to the object and applying a force to it may take the

robot several minutes. At that rate, it is impossible to traverse and backtrack over a large search space. Worse still, it may be impossible to evaluate a projected arrangement of furniture without actually moving the pieces first.

Recent years have seen efforts to integrate research in robotics and AI. The old idea of simply attaching sensors and effectors to existing AI programs has given way to a serious rethinking of basic AI algorithms in light of the problems involved in dealing with the physical world. Research in robotics is likewise affected by AI techniques, since reasoning about goals and plans is essential for mapping perceptions onto appropriate actions. In this chapter, we explore the interface between robotics and AI. We do not delve too deeply into purely robotic issues, but instead focus on how the AI techniques we have seen in this book can be used and/or modified to handle problems that arise in dealing with the physical world.

At this point, one might ask whether physical robots are necessary for research purposes. Since current AI programs already operate in simulated worlds, why not build more realistic simulations, which better model the real world? Such simulators do exist, for example, Carbonell and Hood [1986] and Langley *et al.* [1981b]. There are several advantages to using a simulated world: Experiments can be conducted very rapidly, conditions can easily be replicated, programs can return to previous states at no cost, and sensory input can be treated in a high level fashion. Furthermore, simulators require no fragile, expensive mechanical parts. The major drawback to simulators is figuring out exactly which factors to build in. Experience with real robots continues to expose tough problems that do not arise even in the most sophisticated simulators. The world turns out—not surprisingly—to be an excellent model of itself, and a readily available one.

21.1 Real-Time Search

We now turn to heuristic search, as exemplified in AI by the A* algorithm. While A* is guaranteed to find an optimal path from the initial state to the goal state, the algorithm has a number of limitations in the real world. For one, the exponential complexity of A* limits the size of problems it can realistically solve, and forces us to consider a limited search horizon. Also, having incomplete information about the world can further limit that search horizon. For example, consider the task of navigating from one room to another in an unfamiliar building. The search horizon is limited to how far one can (literally) see at any given time. It is necessary to take steps in the physical world in order to see beyond the horizon, despite the fact that the steps may be nonoptimal ones. Finally, real-time tasks like driving require continuous monitoring and reacting. Because heuristic search is time-consuming, we cannot afford to work out optimal solutions ahead of time.

There is a variation of A* that addresses these issues. It is called Real-Time-A* (RTA*) [Korf, 1988]. This algorithm commits to a real-world action every k seconds, where k is some constant that depends on the depth of the search horizon. Each time RTA* carries out an action, it restarts the search from that point. Thus, RTA* is able to make progress toward a goal state without having to plan a complete sequence of solution steps in advance. RTA* was inspired to a degree by work on computer games.

As we mentioned in Chapter 12, game-playing programs must commit to irrevocable moves because of time constraints.

Algorithm: Real-Time-A*

1. Set NODE to be the start state.
2. Generate the successors of NODE. If any of the successors is a goal state, then quit.
3. Estimate the value of each successor by performing a fixed depth search starting at that successor. Use depth-first search. Evaluate all leaf nodes using the A* heuristic function $f = g + h'$, where g is the distance to the leaf node and h' is the predicted distance to the goal. Pass heuristic estimates up the search tree in such a way that the f value of each internal node is set to the minimum of the values of its children.¹
4. Set NODE to the successor with the lowest score, and take the corresponding action in the world. Store the old NODE in a table along with the heuristic score of the second-best successor. (With this strategy, we can never enter into a fixed loop, because we never make the same decision at the same node twice.) If this node is ever generated again in step 2, simply look up the heuristic estimate in the table instead of redoing the fixed-depth search of step 3.
5. Go to step 2.

We can adjust the depth to which we search in step 3, depending on how much time we want to spend planning versus executing actions in the world. Provided that every part of the search space is accessible from every other part, RTA* is guaranteed to find a path to a solution state if one exists. The path may not be an optimal one, however. The deeper we search in step 3, the shorter our average solution paths will be. Of course, the task itself may impose limits on how deep we can search, as a result of incomplete information.

RTA* is just one example of a limited-horizon search algorithm. Another algorithm, due to Hansson and Mayer [1989], uses Bayesian inference. Dean and Boddy [1988] define a related notion, the *anytime* algorithm. An anytime algorithm is one that can be interrupted and queried at any point during its computation. The longer the algorithm runs, the more accurate its answer is.

Now we turn to more specific techniques aimed at various perceptual and motor problems. Later, we investigate architectures for integrating perception, action, and cognition. It should be noted that this is only a brief survey of a very large and active field of research. Those interested in investigating these issues more deeply should consult robotics texts such as Brady [1982] and Craig [1985].

¹It is possible to prune the search tree using a technique called *alpha pruning*, a single-agent analogue of alpha-beta pruning. Alpha pruning is a branch-and-bound technique of the type we encountered in Chapter 2.

21.2 Perception

We perceive our environment through many channels: sight, sound, touch, smell, taste. Many animals possess these same perceptual capabilities, and others are able to monitor entirely different channels. Robots, too, can process visual and auditory information, and they can also be equipped with more exotic sensors, such as laser rangefinders, speedometers, and radar.

Two extremely important sensory channels for humans are vision and spoken language. It is through these two faculties that we gather almost all of the knowledge that drives our problem-solving behaviors.

21.2.1 Vision

Accurate machine vision opens up a new realm of computer applications. These applications include mobile robot navigation, complex manufacturing tasks, analysis of satellite images, and medical image processing. In this section, we investigate how we can transform raw camera images into useful information about the world.

A video camera provides a computer with an image represented as a two-dimensional grid of intensity levels. Each grid element, or *pixel*, may store a single bit of information (that is, black/white) or many bits (perhaps a real-valued intensity measure and color information). A visual image is composed of thousands of pixels. What kinds of things might we want to do with such an image? Here are four operations, in order of increasing complexity:

1. **Signal Processing**—Enhancing the image, either for human consumption or as input to another program.
2. **Measurement Analysis**—For images containing a single object, determining the two-dimensional extent of the object depicted.
3. **Pattern Recognition**—For single-object images, classifying the object into a category drawn from a finite set of possibilities.
4. **Image Understanding**—For images containing many objects, locating the objects in the image, classifying them, and building a three-dimensional model of the scene.

See Niblack [1986] for algorithms that perform the first two operations. The third operation, pattern recognition, varies in its difficulty. It is possible to classify two-dimensional (2-D) objects, such as machine parts coming down a conveyor belt, but classifying 3-D objects is harder because of the large number of possible orientations for each object. Image understanding is the most difficult visual task, and it has been the subject of the most study in AI. While some aspects of image understanding reduce to measurement analysis and pattern recognition, the entire problem remains unsolved, because of difficulties that include the following:

- An image is two-dimensional, while the world is three-dimensional. Some information is necessarily lost when an image is created.

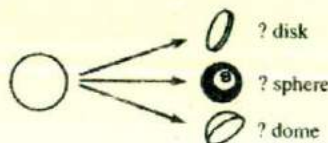


Figure 21.2: An Ambiguous Image

- One image may contain several objects, and some objects may partially occlude others, as we saw earlier in Figure 14.8.
- The value of a single pixel is affected by many different phenomena, including the color of the object, the source of the light, the angle and distance of the camera, the pollution in the air, etc. It is hard to disentangle these effects.

As a result, 2-D images are highly ambiguous. Given a single image, we could construct any number of 3-D worlds that would give rise to the image. For example, consider the ambiguous image of Figure 21.2. It is impossible to decide what 3-D solid it portrays. In order to determine the most likely interpretation of a scene, we have to apply several types of knowledge.

For example, we may invoke knowledge about low-level image features, such as shadows and textures. Figure 21.3 shows how such knowledge can help to disambiguate the image. Having multiple images of the same object can also be useful for recovering 3-D structure. The use of two or more cameras to acquire multiple simultaneous views of an object is called stereo vision. Moving objects (or moving cameras) also supply multiple views. Of course, we must also possess knowledge about how motion affects images that get produced. Still more information can be gathered with a laser rangefinder, a device that returns an array of distance measures much like sonar does. While rangefinders are still somewhat expensive, integration of visual and range data will soon become commonplace. Integrating different sense modalities is called *sensor fusion*. Other image factors we might want to consider include shading, color, and reflectance.

High-level knowledge is also important for interpreting visual data. For example, consider the ambiguous object at the center of Figure 21.4(a). While no low-level image features can tell us what the object is, the object's surroundings provide us with top-down expectations. Expectations are critical for interpreting visual scenes, but resolving expectations can be tricky. Consider the scene shown in Figure 21.4(b). All objects in this scene are ambiguous; the same shapes might be interpreted elsewhere as an amoeba, logs in a fireplace, and a basketball. As a result, there are no clear-cut top-down expectations. But the preferred interpretations of egg, bacon, and plate reinforce each other mutually, providing the necessary expectations.

So how can we bring all of this knowledge to bear in an organized fashion? One possible architecture for vision is shown in Figure 21.5. The very first step is to convert the analog video signal into a digital image. The next step is to extract image features like edges and regions. Edges can be detected by algorithms that look for sets of adjacent pixels with differing values. Since pixel values are affected by many factors, small edges

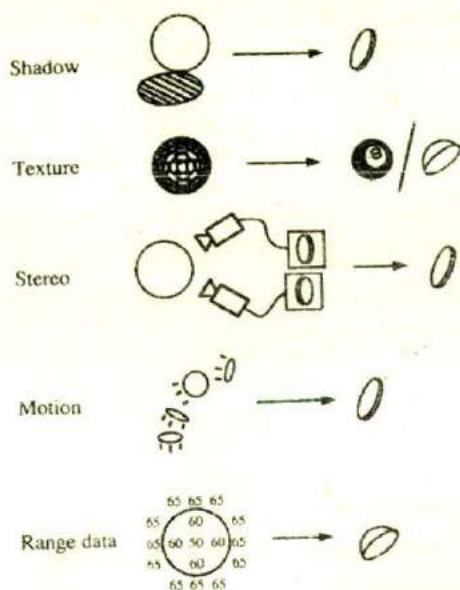


Figure 21.3: Using Low-Level Knowledge to Interpret an Image

12
A|BC
14

(a)



(b)

Figure 21.4: Using High-Level Knowledge to Interpret an Image

with similar orientations must be grouped into larger ones [Ballard and Brown, 1982]. Regions, on the other hand, are found by grouping similar pixels together. Edge and region detection are computationally intensive processes, but ones that can be readily mapped onto parallel hardware. The next step is to infer 3-D orientations for the various regions. Texture, illumination, and range data are all useful for this task. Assumptions about the kinds of objects that are portrayed can also be valuable, as we saw in the Waltz labeling algorithm (Section 14.3). Next, surfaces are collected into 3-D solids. Small solids are combined into larger, composite objects. At this point, the scene is segmented into discrete entities. The final step involves matching these entities against a knowledge base in order to pick the most likely interpretations for them. Organizing such a knowledge base of objects is difficult, though the knowledge-structuring techniques we studied in Part II are useful. As we demonstrated above, it may be impossible to interpret objects in isolation. Therefore, higher-level modules can pass hypotheses back down to lower level modules, which check for predictions made by the hypotheses.

This is only one way of structuring an image understanding program. It highlights the spectrum of low- to high-level knowledge required for 3-D vision. As with other AI tasks, the success of a vision program depends critically on the way it represents and applies knowledge. For more information on computer vision, see Marr [1982], Ballard and Brown [1982], and Horn [1986].

21.2.2 Speech Recognition

Natural language understanding systems usually accept typed input, but for a number of applications this is not acceptable. Spoken language is a more natural form of communication in many human-computer interfaces. Speech recognition systems have been available for some time, but their limitations have prevented widespread use. Below are five major design issues in speech systems. These issues also provide dimensions along which systems can be compared with one another.

- **Speaker Dependence versus Speaker Independence**—A speaker-independent system can listen to any speaker and translate the sounds into written text. Speaker independence is hard to achieve because of the wide variations in pitch and accent. It is easier to build a speaker-dependent system, which can be trained on the voice patterns of a single speaker. The system will only work for that one speaker. It can be retrained on another voice, but then it will no longer work for the original speaker.
- **Continuous versus Isolated-Word Speech**—Interpreting isolated-word speech, in which the speaker pauses between each word, is easier than interpreting continuous speech. This is because boundary effects cause words to be pronounced differently in different contexts. For example, the spoken phrase "could you" contains a *j* sound, and despite the fact it contains two words, there is no empty space between them in the speech wave. The ability to recognize continuous speech is very important, however, since humans have difficulty speaking in isolated words.
- **Real Time versus Offline Processing**—Highly interactive applications require that a sentence be translated into text as it is being spoken, while in other situations,

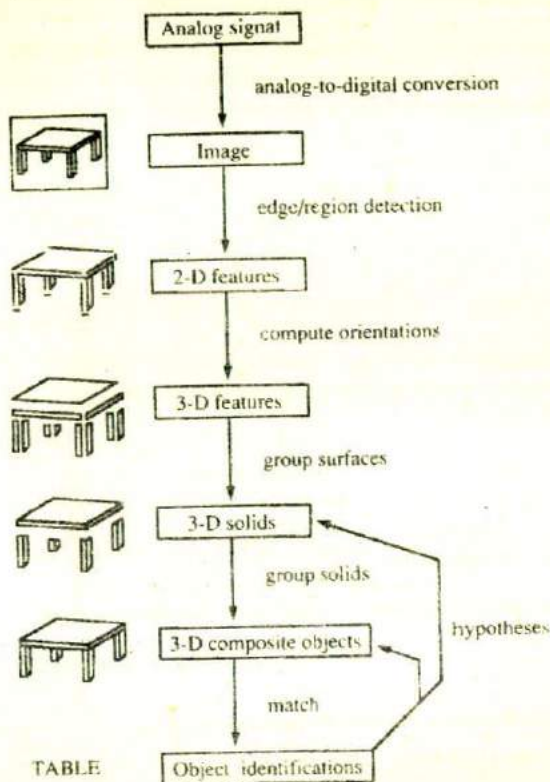


Figure 21.5: One Possible Architecture for Image Understanding

it is permissible to spend minutes in computation. Real-time speeds are hard to achieve, especially when higher-level knowledge is involved.

- **Large versus Small Vocabulary**—Recognizing utterances that are confined to small vocabularies (e.g., 20 words) is easier than working with large vocabularies (e.g., 20,000 words). A small vocabulary helps to limit the number of word candidates for a given speech segment.
- **Broad versus Narrow Grammar**—An example of a narrow grammar is the one for phone numbers: $S \rightarrow XXX-XXXX$, where X is any number between zero and nine. Syntactic and semantic constraints for unrestricted English are much harder to represent, as we saw in Chapter 15. The narrower the grammar is, the smaller the search space for recognition will be.

Existing speech systems make various compromises. Early systems, like DRAGON [Baker, 1975], HEARSAY [Lesser *et al.*, 1975], and HARPY [Lowerre, 1976] dealt with single-user, continuous speech, and vocabularies up to a thousand words. They achieved word accuracy rates of 84 to 97 percent. TANGORA [IBM speech recognition group, 1985] moved to speaker independence and a large, 20,000-word vocabulary, but sacrificed continuous speech. TANGORA is 97 percent accurate. One system built at Bell Labs for recognizing continuous, speaker-independent digit recognition (for phone numbers) has also produced 97 percent accuracy [Rabiner *et al.*, 1988]. SPHINX [Lee and Hon, 1988] is the first system to achieve high accuracy (96 percent) on real-time, speaker independent, continuous speech with a vocabulary of 1000 words.

What techniques do these systems use? HEARSAY used a blackboard architecture, of the kind we discussed in Chapter 16. Using this method, various knowledge sources enter positive and negative evidence for different hypotheses, and the blackboard integrates all the evidence. Low-level phonemic knowledge sources provide information that high-level knowledge sources can use to make hypotheses about what words appear in the input. The high-level knowledge sources can then generate expectations that can be checked by the low-level ones.

The HARPY system also used knowledge to direct its reasoning, but it precompiled all that knowledge into a very large network of phonemes. In the network model, an interpreter tries to find the path through the network that best matches the spoken input. This path can be found with any number of heuristic search techniques, for example, beam search. HARPY was much faster than HEARSAY, but the blackboard architecture that HEARSAY used was more general and easily extensible.

Most modern speech systems are learning systems. In other words, they accept sample inputs and interpretations, and modify themselves appropriately until they are able to transform speech waveforms into written words. So far, statistical learning methods have proven most useful for learning this type of transformation. The statistical method used in the SPHINX system is called *hidden Markov modeling*. A hidden Markov model (HMM) is a collection of states and transitions. Each transition leaving a state is marked with (1) the probability with which that transition is taken, (2) an output symbol, and (3) the probability that the output symbol is emitted when the transition is taken. The problem of decoding a speech waveform turns into the problem of finding the most likely path (set of transitions) through an appropriate HMM. It is possible to tune the probabilities of an HMM automatically so that these paths correspond to correct interpretations of the waveform. The technique for doing this is called the *forward-backward algorithm*.

Connectionist systems also show promise as a learning mechanism for speech recognition. One problem with connectionist models is that they do not deal very well with time-varying data. New types of networks, such as recurrent and time-delay networks [Waibel *et al.*, 1989], are being employed to overcome these difficulties.

In our discussion of vision in Section 21.2.1, we saw that higher-level sources of knowledge can be used to manage uncertainty at lower levels. Speech recognition also has sources of higher-level knowledge. We have already studied some of these in Chapter 15. Syntactic knowledge can be used to identify constituent phrases, semantic knowledge to disambiguate word senses, discourse knowledge to dereference pronouns, and so forth. Early speech recognition systems sought to make use of this higher-level knowledge in order to constrain the interpretation at the lower levels. As we saw in

Chapter 14, a speech system that cannot decide between "the cat's cares are few" and "the cat scares are few" can invoke high-level knowledge to choose one alternative over the other.

However, modern speech systems perform fairly well without any sophisticated syntactic or semantic models of language. Instead, simple statistical models are used. For example, SPHINX uses a *word-pair grammar*, which tells it which words can legally appear adjacent to one another in the input. TANGORA uses a *trigram grammar*, which, given the previous two words in the input, yields the probability that a given word will occur next.

Still, no speech system is 100 percent accurate. There has recently been renewed interest in integrating speech recognition and natural language processing in order to overcome the final hurdle. For example, ATNs and unification-based grammars can be used to constrain the hypotheses made by a speech system. Thus far, integration has proved difficult, because natural language grammars do not offer much in the way of constraints.

In the speech recognition literature, there is a quantitative measure of grammar, called *perplexity*. Perplexity measures the number of words that can legally appear next in the input (on average). The telephone number recognition task has a perplexity of 10, because at any decision point, there are ten alternatives. On a sample 1000-word English task, a word-pair grammar may reduce the perplexity from 1000 down to 60. A bigram grammar may reduce it further, perhaps to 20 [Lee and Hon, 1988].

While natural language grammars accurately predict word categories (such as noun and verb), they say nothing about which words within a category are likely to show up in the input. For example, given the word "the," a grammar might hypothesize that the next word is either an adjective or a noun. But this knowledge does us little good when there are thousands of possible adjectives and nouns to choose from. Thus, it is natural to turn to statistical, or collocational, facts about language. For example, if the word "doctor" is recognized, then one might expect to hear the word "nurse" later in the input, but not "Norse." Collocational data, unlike more complex syntactic and semantic structures, can be extracted automatically from large on-line bodies of text. Ultimately, we want to substitute semantic and discourse information for statistical data. If we know the conversation is about doctors, and if we know that doctors and nurses typically work together, then we should be able to generate the proper expectations. Such a strategy will require large knowledge bases and a deeper understanding of semantics and discourse.

21.3 Action

Mobility and intelligence seem to have evolved together. Immobile creatures have little use for intelligence, while it is intelligence that puts mobility to effective use. In this section, we investigate the nature of mobility in terms of how robots navigate through the world and manipulate objects.

21.3.1 Navigation

Navigation means moving around the world: planning routes, reaching desired destinations without bumping into things, and so forth. Like vision and speech recognition,

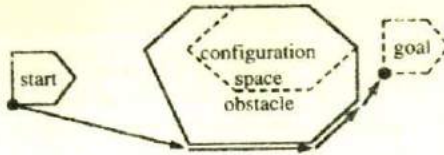


Figure 21.9: Constructing Configuration Space Obstacles

shape. Fortunately, we can reduce this problem to the previous path-planning problem. The algorithm is as follows: First choose a point P on the surface of the robot, then increase the size of the obstacles so that they cover all points that P cannot enter, because of the physical size and shape of the robot. Now, simply construct and search a visibility graph based on P and the vertices of the new obstacles, as in Figure 21.9. The basic idea is to reduce the robot to a point P and do path planning in an artificially constructed space, known as *configuration space*, or *c-space* [Lozano-Pérez et al., 1984].

If we want to allow rotations, we can represent the robot as a combination of point P and some angle of rotation θ . The robot can now be considered as a point moving through three-dimensional space (x, y, θ) . Obstacles can be transformed into three-dimensional *c-space* objects, and a visibility graph can again be created and searched.

An alternative approach to obstacle avoidance is the use of *potential fields* [Khatib, 1986]. With this technique, the direction of a moving robot is continually recomputed as a function of its current position relative to obstacles and its destination. The robot is essentially repelled by obstacles and attracted to the destination point. This approach is especially useful for correcting positioning errors that accumulate during a robot's journey and for dealing with unexpected obstacles. It can be combined with configuration space path planning to enable robust navigation [Krogh and Thorpe, 1986].

Road following is another navigational task that has received a great deal of attention. The object of road following is to steer a moving vehicle so that it stays centered on a road and avoids obstacles. Much of the problem comes in locating the edges of the road despite varying light, weather, and ground conditions. At present, this control task is feasible only for fairly slow-moving vehicles [Shafer and Whittaker, 1989]. Increases in speed demand more reactivity and thus more real-time computation.

21.3.2 Manipulation

Robots have found numerous applications in industrial settings. Robot manipulators are able to perform simple repetitive tasks, such as bolting and fitting automobile parts, but these robots are highly task-specific. It is a long-standing goal in robotics to build robots that can be programmed to carry out a wide variety of tasks.

A manipulator is composed of a series of links and joints, usually terminating in an *end-effector*, which can take the form of a two-pronged gripper, a humanlike hand, or any of a variety of tools. One general manipulation problem is called *pick-and-place*, in which a robot must grasp an object and move it to a specific location. For example, consider Figure 21.10, where the goal is to place a peg in a hole.

There are two main subtasks here. The first is to design a robot motion that ends

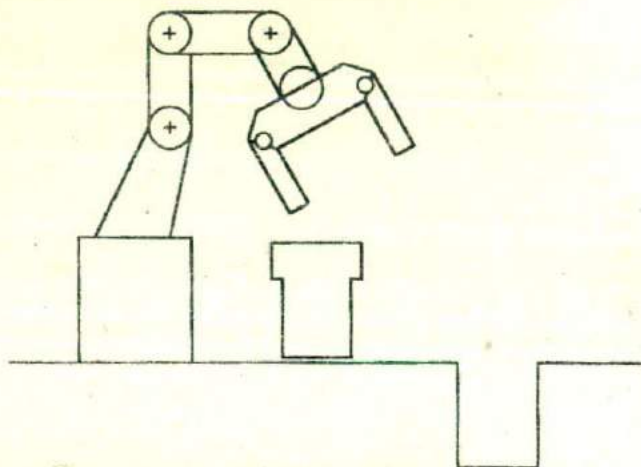


Figure 21.10: A Pick-and-Place Task

with the object stably grasped between the two fingers of the robot. Clearly some form of path planning, as discussed above, can be used to move the arm toward the object, but we need to modify the technique when it comes to the fine motion involved in the grasp itself. Here, uncertainty is a critical problem. Even with the vision techniques of Section 21.2.1, a robot can never be sure of the precise location of the peg or the arm. Therefore it would be a mistake to plan a grasp motion in which the gripper is spread only wide enough to permit the peg to pass, as in Figure 21.11(a). A better strategy is to open the gripper wide, then close gradually as the gripper gets near the peg, as in Figure 21.11(b). That way, if the peg turns out to be located some small distance away from where we thought it was, the grasp will still succeed. Although this strategy depends less on precise vision, it requires some tactile sensitivity in order to terminate the grasp. Unless we take special care in designing grasping motions, uncertainty can lead to disasters. For example, should the left side of the gripper touch the peg one second before the right side does, the peg may fall, thus foiling the grasp. Brost [1988] and Mason *et al.* [1988] give robust algorithms for grasping a wide variety of objects.

After the peg is stably grasped, the robot must place it in the hole. This subtask resembles the path-planning problem, although it is complicated by the fact that moving the peg through 3-D space requires careful orchestration of the arm's joints. Also, we must seriously consider the problems introduced by uncertainty. Figure 21.12(a) shows a naive strategy for placing the peg. Failure will result from even a slight positioning error, because the peg will jam flatly on the outer surface. A better strategy is shown in Figure 21.12(b). We slide the peg along the surface, applying downward pressure so that the peg enters the hole at an angle. After this happens, we straighten the peg gradually and push it down into the hole.

This type of motion, which reacts to forces generated by the world, is called *compli-*

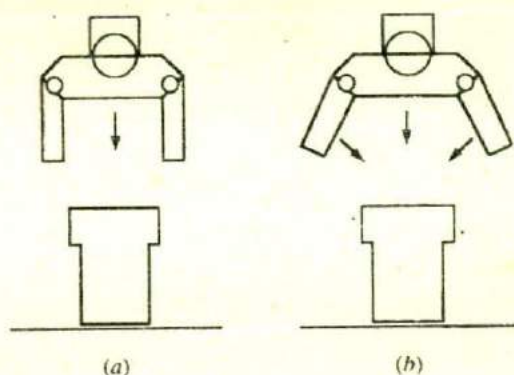


Figure 21.11: Naive and Clever Strategies for Grasping

ant motion. Compliant motion is very robust in the face of uncertainty. Humans employ compliant motion in a wide variety of activities, such as writing on chalkboards.

So given a pick-and-place problem, how can we automatically generate a sequence of compliant motions? One approach [Lozano-Perez *et al.*, 1984] is to use the familiar problem-solving process of backward chaining. Our initial and goal states for the peg-in-hole problem are represented as points in configuration space, as shown in Figure 21.13. First, we compute the set of points in c -space from which we are guaranteed to reach the goal state in a single compliant motion, assuming a certain degree of uncertainty in initial position and direction of movement and certain facts about relative friction. This set of points is called the goal state's *strong pre-image*.² In Figure 21.13, the strong pre-image of the goal state is shown in gray. Now we use backward chaining to design a set of motions that is guaranteed to get us from the initial state to some point in the goal state's strong pre-image. Recursively applying this procedure will eventually yield a set of motions that, while individually uncertain, combine to form a guaranteed plan.

21.4 Robot Architectures

Now let us turn to what happens when we put it all together—perception, cognition, and action. There are many decisions involved in designing an architecture that integrates all these capabilities, among them:

- What range of tasks is supported by the architecture?
- What type of environment (e.g., indoor, outdoor, space) is supported?
- How are complex behaviors turned into sequences of low-level actions?
- Is control centralized or distributed?

²The set of points from which it is possible to reach the state in a single motion is called the state's *weak pre-image*.

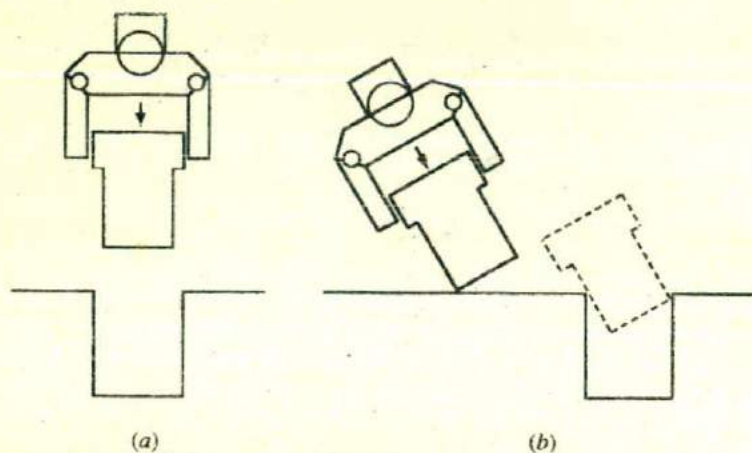


Figure 21.12: Naive and Clever Strategies for Placement

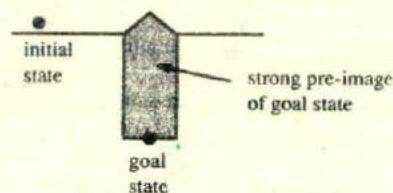


Figure 21.13: Planning with Uncertainty in Configuration Space

- How are numeric and symbolic representations merged?
- How does the architecture represent the state of the world?
- How quickly can the architecture react to changes in the environment?
- How does the architecture decide when to plan and when to act?

With these issues in mind, let's look briefly at a few existing robot architectures.

CODGER [Shafer *et al.*, 1986] is an architecture for controlling vehicles in outdoor road-following tasks. CODGER uses a blackboard structure to organize incoming perceptual data. The system's control is centralized and hierarchical—all numerical data from sensors are fused in order to build up a consistent model of the world. This model is represented symbolically. CODGER has been used to build a system for driving the experimental NAVLAB [Shafer and Whittaker, 1989] vehicle, a commercial van that has been altered for computer control via electric and hydraulic servos. The

NAVLAB is completely self-contained, with room for several on-board computers and researchers.

Brooks [1986] describes the *subsumption* architecture for building autonomous robots for indoor navigation and manipulation. Behaviors are built up from layers of simple, numeric finite-state machines. Bottom layers consist of reactive, instinctual behaviors such as obstacle avoidance and wandering. Upper layers consist of behaviors like object identification and reasoning about object motions. The various behaviors operate in a decentralized fashion, computing independently, and suppressing or informing one another. Such an organization encourages reactivity—for example, high-level navigation behavior is suppressed abruptly when an obstacle moves to block a robot's path. In fact, the subsumption architecture takes reactivity to the extreme. Separate modules monitor only the sensors that affect their behavior, and there are no explicit goals, plans, or world models in these systems. They simply react to the situation at hand. For example, the task of one such robot is to wander the halls, picking up soda cans and depositing them in a bin. When the robot locates a can, several modules steer the robot toward it. Modules governing the robot arm continuously monitor the physical wheels of the robot. When the wheels stop, the arm extends to grasp the can. Notice that all these motions are decentralized and reactive; nowhere in the robot is there any explicit plan for how to pick up the soda can, or how to pick up soda cans in general.

This kind of organization presents a perspective on problem solving similar to the one we described in Section 13.7. Advantages of the subsumption architecture include simplicity and speed, since programs for controlling such robots are simple enough that they can be rendered easily into hardware. Also, modeling the real world is a very difficult task, one that the subsumption architecture avoids. On the other hand, it is not clear that the subsumption architecture will scale up to complex planning problems. Subsumption robots tend to lack the flexibility that traditional problem solvers display in being able to reason about a wide variety of tasks. Also, they lack the ability to reflect on their own actions. For example, if the wheels of the soda can robot should stop turning because of a loose connection, the robot arm will mindlessly extend forward in search of a nonexistent can. While the CODGER architecture emphasizes data fusion, subsumption robots emphasize data fission. A series of subsumption robots have been built, and they demonstrate how reactive systems are capable of much more interesting and varied behavior than was previously thought. It is unknown whether these architectures are capable of achieving tasks that seem to require significant amounts of planning.

TCA [Simmons and Mitchell, 1989] is an architecture that combines the idea of reactive systems with traditional AI planning. TCA is a distributed system with centralized control, designed to control autonomous robots for long periods in unstructured environments, such as the surface of Mars. TCA particularly addresses issues that arise in the context of multiple goals and limited resources. The architecture provides mechanisms for hierarchical task management and allows action based on incomplete plans. Because robots gather new information by moving through the world, TCA permits plans to be terminated early should higher-priority goals arise. Some situations require highly reactive behavior. TCA achieves high-speed response by parallelizing planning and execution whenever possible. For example, in designing walking motions over rough terrain, TCA plans one step, initiates it, and then begins to plan the next step before the leg motion has been completed.

Another program for combining heuristic problem solving with reactivity is called THEO-Agent [Mitchell, 1990]. THEO-Agent contains two subsystems, a reactive engine and a general problem solver (called THEO [Mitchell *et al.*, 1989]). When the reactive subsystem fails to suggest a course of action, the problem solver creates a plan for the robot. As it executes the plan, the robot uses explanation-based learning to create new reactive modules. Thus, the robot becomes increasingly reactive with experience. Robo-SOAR [Laird *et al.*, 1989], an extension of the SOAR problem-solving system, is another learning robot architecture.

PRS [Georgeff and Lansky, 1987] is a symbolic robot planning system that interleaves planning and execution. In PRS, goals represent robot behaviors, not world states. PRS contains procedures for turning goals into subgoals or iterations thereof. A procedure can be invoked by either the presence of a goal or the presence of some sensory input. Thus, the robot is capable of goal-directed behavior but can also react when the world changes or when plans fail. Goals and procedures are represented symbolically, and a central reasoner uses a stack to oversee the invocation of procedures.

21.5 Summary

The field of robotics is often described as the subfield of AI that is concerned with perceptual and motor tasks. As Figure 21.1 suggests, the tables can easily be turned, and AI could well be the subfield of robotics that deals with cognition. Indeed, Brady [1985] has proposed a definition of robotics with this flavor:

Robotics is the intelligent connection of perception to action.

Another definition, suggested by Grossman,³ reads as follows:

A robot is anything that is surprisingly animate.

The word "surprisingly" suggests a moving-target definition. It should be noted that the first automatic dishwashing machines were called robots by their designers. But after a while, it became less surprising that a machine could wash dishes, and the term "robot" fell away. This characterization of robotics is similar to the one we proposed for AI in Chapter 1. There, we characterized AI as the study of problems in which humans currently perform better than computers. As a result, programs that solve calculus problems are no longer considered artificial intelligence.⁴

These moving-target definitions accurately differentiate actual AI work and robotics work. AI tends to focus on uniquely human capabilities, while robotics aims to produce physical, animate behaviors. As we have seen in this chapter, however, many interesting problems lie at the intersection of AI and robotics, and only by combining techniques from both fields will we be able to design intelligent robots that live in the world.

³David Grossman, after-dinner speech delivered at the 7th NSF Grantees Conference, Ithaca, NY, 1979.

⁴We must be careful here. When movable-type printing was first introduced, it was called *artificial writing*, because it seemed to be automating what scribes had been doing for previous centuries. Of course, printing only automates a small portion of the writing process. It is often more enlightening to view AI programs and robots as tools for enhancing human capabilities, rather than as independent, autonomous agents [Hill, 1989].

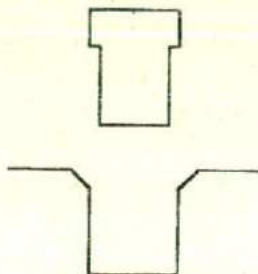
21.6 Exercises

1. Describe scenarios in which the following features are critical:
 - (a) **Reactivity**—The robot must react quickly to a changing environment.
 - (b) **Robustness**—The robot must act appropriately, in spite of incomplete or inexact sensory data.
 - (c) **Recoverability**—When a plan fails to bring about expected results, the robot must find another way to achieve its goal.

Why aren't the planning techniques described in Chapter 13 sufficient to ensure these characteristics?

2. Describe three different ways of combining speech recognition with a natural language understanding system. Compare and contrast them in terms of expected performance and ease of implementation.
3. Say each of the following phrases very slowly, and write down the sounds you use. Then gradually speed up, and continue to write down the sounds. Finally, say them the way you would in ordinary speech. How do the sounds change as you move through each series? What are the implications of these changes for continuous speech recognition?
 - (a) could you
 - (b) boy's school
 - (c) the store, the elevator
 - (d) sharp point
 - (e) stop it
 - (f) want to go
4. Create a search graph, labeled with heuristic estimates, that shows the RTA* algorithm entering the same node twice. Explain what would happen if RTA* did not keep track of previously visited states.
5. In Section 21.1, we said that the RTA* algorithm is guaranteed to find a path to a solution state if such a path exists and if every part of the search space is accessible from every other part. Why is this second qualification necessary? Give an example in which, without it, a solution will not be found.

6. Consider the following variation on the peg-in-hole problem:



Explain, using the concept of a strong pre-image, why this problem is easier than the standard peg-in-hole problem of Figure 21.10.

Chapter 22

Conclusion

22.1 Components of an AI Program

We have now surveyed the major techniques of artificial intelligence. From our discussion of them, it should be clear that there are two important classes of AI techniques:

- Methods for representing and using knowledge
- Methods for conducting heuristic search

These two aspects interact heavily with each other. The choice of a knowledge representation framework determines the kind of problem-solving methods that can be applied. For example, if knowledge is represented as formulas in predicate logic, then resolution can be used to derive new inferences. If, on the other hand, knowledge is represented in semantic nets, then network search routines must be used. Or, if knowledge is represented as a set of weights in a neural network, then some form of network search (e.g., relaxation or forward propagation) must be exploited.

If there is one single message that this book has tried to convey, it is the crucial part that knowledge plays in AI programs. Although much of the book has been devoted to other topics, particularly to search techniques, it is important to keep in mind that the power of those techniques lies in their ability to use knowledge effectively to solve particular problems. Because of the importance of the role of knowledge in problem-solving programs, it is worth reviewing here what that role is.

Knowledge serves two important functions in AI programs. The first is to define what can be done to solve a problem and to specify what it means to have solved the problem. We can call knowledge that does this *essential knowledge*. The second is to provide advice on how best to go about solving a problem efficiently. We can call such knowledge *heuristic knowledge*.

The goal of this book has been to say enough about the use of knowledge in problem-solving programs to enable you to build one. Go do it. And have fun.

22.2 Exercises

1. What do you think is the main result to come out of AI research in the last 20 years? Give a brief justification of your answer.
2. Why are table-driven programs so important in AI?
3. What is the role of matching in AI programs? Give several examples of its use.
4. How do the topics of knowledge representation and problem-solving techniques interact with each other? Give examples.
5. Dreyfus [1972] presents a criticism of AI in which it is argued that AI is not possible. Read through it, and using the material presented in this book, refute the arguments.
6. Using what you have learned in this book, comment briefly on each line below. Feel free to augment your answer with diagrams or illustrations.

Question. How many AI people does it take to change a lightbulb?

Answer. At least 67.

The Problem Space Group (5)

One to define the goal state

One to define the operators

One to describe the universal problem solver

One to hack the production system

One to indicate about how it is a model of human lightbulb-changing behavior

The Logical Formalism Group (12)

One to figure out how to describe lightbulb changing in predicate logic

One to show the adequacy of predicate logic

One to show the inadequacy of predicate logic

One to show that lightbulb logic is nonmonotonic

One to show that it isn't nonmonotonic

One to incorporate nonmonotonicity into predicate logic

One to determine the bindings for the variables

One to show the completeness of the solution

One to show the consistency of the solution

One to hack a theorem prover for lightbulb resolution

One to indicate how it is a description of human lightbulb-changing behavior

One to call the electrician

The Statistical Group (1)

One to point out that, in the real world, a lightbulb is never "on" or "off," but usually somewhere in between.

The Planning Group (4)

- One to define STRIPS-style operators for lightbulb changing
- One to show that linear planning is not adequate
- One to show that nonlinear planning is adequate
- One to show that people don't plan; they simply react to lightbulbs

The Robotics Group (7)

- One to build a vision system to recognize the dead bulb
- One to build a vision system to locate a new bulb
- One to figure out how to grasp the lightbulb without breaking it
- One to figure out the arm solutions that will get the arm to the socket
- One to organize the construction teams
- One to hack the planning system
- One to indicate how the robot mimics human motor behavior in lightbulb changing

The Knowledge Engineering Group (6)

- One to study electricians changing lightbulbs
- One to arrange for the purchase of the Lisp machines
- One to assure the customer that this is a hard problem and that great accomplishments in theory will come from support of this effort
- The same one can negotiate the project budget
- One to study related research
- One to indicate how it is a description of human lightbulb-changing behavior
- One to call the Lisp hackers

The Lisp Hackers (7)

- One to bring up the network
- One to order the Chinese food
- Four to hack on the Lisp debugger, compiler, window system, and microcode
- One to write the lightbulb-changing program

The Connectionist Group (6)

- One to claim that lightbulb changing can only be achieved through massive parallelism
- One to build a backpropagation network to direct the robot arm
- One to assign initial random weights to the connections in the network
- One to train the network by showing it how to change a lightbulb (training shall consist of 500,000 repeated epochs)
- One to tell the media that the network learns "just like a human does"
- One to compare the performance of the resulting system with that of traditional symbolic approaches (optional)

The Natural Language Group (5)

- One to collect sample utterances from the lightbulb domain
- One to build an English understanding program for the lightbulb-changing robot
- One to build a speech recognition system
- One to tell lightbulb jokes to the robot in between bulb-changing tasks
- One to build a language generation component so that the robot can make up its own lightbulb jokes

The Learning Group (4)

- One to collect twenty lightbulbs
- One to collect twenty "near misses"
- One to write a concept learning program that learns to identify lightbulbs
- One to show that the program found a local maximum in the space of lightbulb descriptions

The Game-Playing Group (5)

- One to design a two-player game tree with the robot as one player and the lightbulb as the other
- One to write a minimax search algorithm that assumes optimal play on the part of the lightbulb
- One to build special-purpose hardware to enable 24-ply search
- One to enter the robot in a human lightbulb-changing tournament
- One to state categorically that lightbulb changing is "no longer considered AI"

The Psychological Group (5)

- One to build an apparatus which will time lightbulb-changing performance
- One to gather and run subjects
- One to mathematically model the behavior
- One to call the expert systems group
- One to adjust the resulting system, so that it drops the right number of bulbs