ARTIFICIAL INTELLIGENCE

(DMCA301)

(MCA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

UNIT – I

Problems, Problem Spaces, and Search

Objectives of this lesson are

- To define the Problem as a State Space Search
- To know about the Production Systems
- To analyze the Problem Characteristics
- To explore the Production System Characteristics
- To know the issues in the Design of Search Programs & to solve a few problems

Introduction

Artificial Intelligence is typically concerned with different kinds of problems as well as the techniques it offers to solve those problems. To build a system to solve a particular problem, we need to do four things:

1.Define the problem precisely. This definition must include precise specifications of what the initial situation(s) will be as

well as what final situations constitute acceptable solutions to the problem.

2. Analyze the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.

3. Isolate and represent the task knowledge that is necessary to solve the problem.

4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

In this chapter and the next, we discuss the first two and the last of these issues. Then, in the chapters in Part II, we focus on the issue of knowledge representation.

1.1 Defining the Problem as a State Space Search

Suppose we start with the problem statement "Play chess." Although there are a lot of people to whom we could say that and reasonably expect that they will do as we intended, as our request now stands it is a very incomplete statement of the problem we want solved. To build a program that could "Play chess," we would first have to specify the starting position of the chess board, the rules that define the legal moves, and the board positions that represent a win for one side or the other. In addition, we must make explicit the previously implicit goal of not only playing a legal game of chess but also winning the game, if possible.

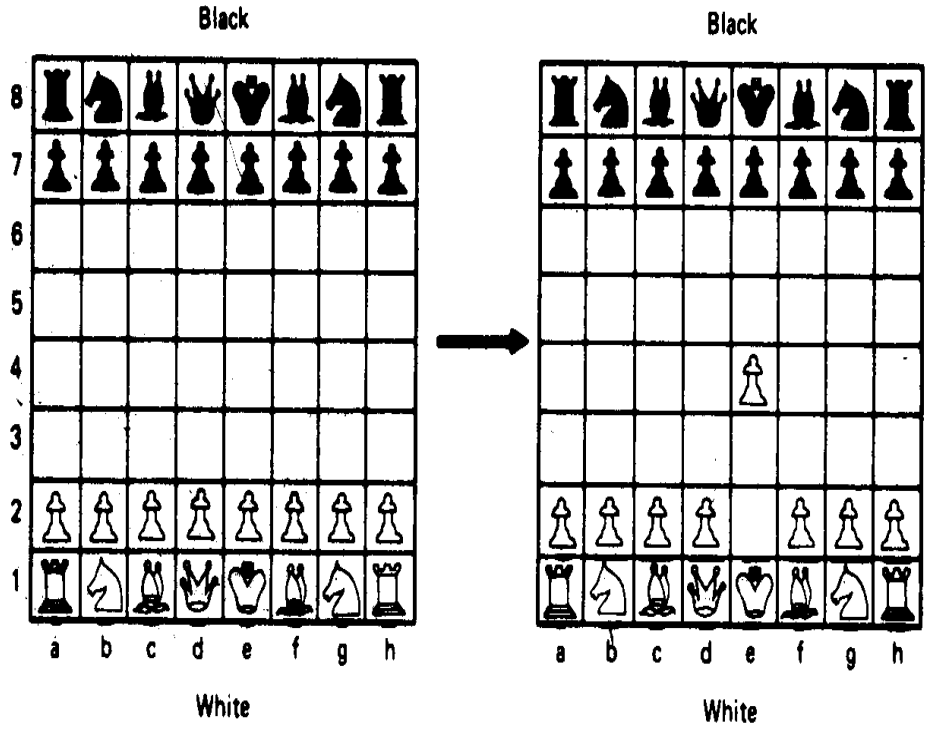


Figure 1.1: One Legal Chess Move

For the problem "Play chess," it is fairly easy to provide a formal and complete problem description. The starting position can be described as an 8-by-8 array where each position contains a symbol standing for the appropriate piece in the official chess opening position. We can define as our goal any board position in which the opponent does not have a legal move and his or her king is under attack. The legal moves provide the way of getting from the initial state to a goal state. They can be described easily as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that describes the change to be made to the board position to reflect the move. There are several ways in which these rules can be written. For example, we could write a rule such as that shown in Figure 1.1.

However, if we write rules like the one above, we have to write a very large number of them since there has to be a separate

rule for each of the roughly 10^{120} possible board positions. Using so many rules poses two serious practical difficulties:

- No person could ever supply a complete set of such rules. It would take too long and could certainly not be done without mistakes.

- No program could easily handle all those rules. Although a hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules poses serious difficulties.

In order to minimize such problems, we should look for a way to write the rules describing the legal moves in as general a way as possible. TO do this, it is useful to' introduce some convenient notation for describing patterns and substitutions. For example, the rule described in Figure 1.1, as well as many like it, could be written as shown in Figure 1..2, In general, the more succinctly we can describe the rules we need, the less work we will have to do to provide them and the more efficient the program that uses them can be.

We 'have just defined the problem of playing chess as a problem of moving around in a state space, where each state corresponds to a legal position of the board. We can

'To be completely accurate, this rule should include a check for pinned pieces, which have been ignored here.

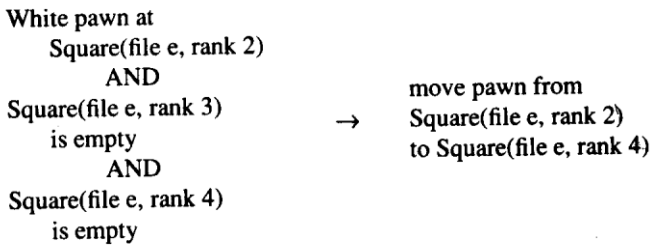


Figure 1.2: Another Way to Describe Chess Moves

play chess by starting at an initial state, using a set of rules to move from one state to another, and attempting to end up in one of a set of final states. This state space representation seems natural for chess because the set of states, which corresponds to the set of board positions, is artificial and well-organized. This same kind of representation is also useful for naturally occurring, less well-structured problems, although it may be necessary to use more complex structures than a matrix to describe an individual state. The state space representation forms the basis of most of the AI methods we discuss here. Its structure corresponds to the structure of problem solving in two important ways:

- It allows for a formal definition of a problem as the need to convert some given situation into some desired situation using a set of permissible operations.

- It permits us to define the process of solving a particular problem as a combination of known techniques (each represented as a rule defining a single step in the space) and search, the general technique of exploring the space to try to find some path from the current state to a goal state. Search is a very important process in the solution of hard problems for which no more direct techniques are available.

In order to show the generality of the state space representation, we use it to describe a problem very different from that of chess.

A Water Jug Problem:

You are given two jugs, a 4-gallon one and a 3-gallon one. Neither have any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

The state space for this problem can be described as the set of ordered pairs of integers (x, y), such that x = 0,1,2,3, or 4 and y = 0,1,2, or 3; x represents the number of gallons of water in the 4-gallon jug, and y represents the quantity of water in the 3-gallon jug. The start state is (0, 0). The goal state is (2, n) for

any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug).

The operators to be used to solve the problem can be described as shown in Figure 1.3. As in the chess problem, they are represented as rules whose left sides are matched against the current state and whose right sides describe the new state that results from applying the rule. Notice that in order to describe the operators completely, it was necessary to make explicit some assumptions not mentioned in the problem statement. We have assumed that we can fill a jug from the pump, that we can pour water out of a jug onto the ground, that we can pour water from one jug to another, and that there are no other measuring devices available. Additional assumptions such as these are almost always required when converting from a typical problem statement given in English to a formal representation of the problem suitable for use by a program.

To solve the water jug problem, all we need, in addition to the problem description given above, is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state. As long as it does not, the cycle continues. Clearly the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to be performed. In Chapter 3, we discuss several ways of making that selection.

For the water jug problem, as with many others, there are several sequences of operators that solve the problem. One such sequence is shown in Figure 1.4. Often, a problem contains the explicit or implied statement that the shortest (or cheapest) such sequence be found. If present, this requirement will have a significant effect on the choice of an appropriate mechanism to guide the search for a solution.

Several issues that often arise in converting an informal problem statement into a formal problem description are illustrated by this sample water jug problem. The first of these

issues concerns the role of the conditions that occur in the left sides of the rules. All but one of the rules shown in Figure 1.3 contains conditions that must be satisfied before the operator described by the rule can be applied. For example, the first rule says, "If the 4-gallonjug is not already full, fill it." This rule could, however, have been written as, "Fill the 4-gallon jug," since it is physically possible to fill the jug even if it is already full. It is stupid to do so since no change in the problem state results, but it is possible. By encoding in the left sides of the rules constraints that are not strictly necessary but that restrict the application of the rules to states in which the rules are most likely to lead to a solution, we can generally increase the efficiency of the problem-solving program that uses the rules.

Each entry in the move vector corresponds to a rule that describes an operation. The left side of each rule describes a board configuration and is represented implicitly by the index position. The right side of each rule describes the operation to be performed and is represented by a nine-element vector that corresponds to the resulting board configuration. Each of these rules is maximally specific; it applies only to a single board configuration, and, as a result, no search is required when such rules are used. However, the drawback to this extreme approach is that the problem solver can take no action at all in a novel situation. In fact, essentially no problem solving ever really occurs. For a tic-tac-toe playing program, this is not a serious problem, since it is possible to enumerate all the situations (i.e., board configurations) that may occur. Bull for most problems, this is not the case. In order to solve new problems, more general rules must be available.

1	(x, y) if $x < 4$	$\rightarrow (4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	$\rightarrow (x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	$\rightarrow (x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	$\rightarrow (x, y - d)$	Pour some water out of the 3-gallon jug
5	(x, y) if $x > 0$	$\rightarrow (0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	$\rightarrow (x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	$\rightarrow (4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	$\rightarrow (x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	$\rightarrow (x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	$\rightarrow (0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	$\rightarrow (2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	$\rightarrow (0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Figure 1.3: Production Rules for the Water Jug Problem

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

Figure 1.4: One Solution to the Water Jug Problem

A second issue is exemplified by rules 3 and 4 in Figure1.3. Should they or should they not be included in the list of available operators? Emptying an unmeasured amount of water onto the ground is certainly allowed by the problem statement. But a superficial preliminary analysis of the problem makes it clear that doing so will never get us any closer to a solution. Again, we see the tradeoff between writing a set of rules that describe just the problem itself, as opposed to a set of rules that describe both the problem and some knowledge about its solution.

Rules 11 and 12 illustrate a third issue. To see the problem-solving knowledge that these rules represent, look at the last two steps of the solution shown in Figure 1.4. Once the state (4, 2) is reached, it is obvious what to do next. The desired 2 gallons have been produced, but they are in the wrong jug. So the thing to do is to move them (rule 11). But before that can be done, the water that is already in the 4-gallon jug must be emptied out (rule 12). The idea behind these special-purpose rules is to capture the special-case knowledge that can be used at this stage in solving the problem. These rules do not actually add power to the system since the operations they describe are already provided by rule 9 (in the case of rule 11) and by rule 5 (in the case of rule 12). In fact depending on the control strategy that is used for selecting rules to use during

problem solving, the use of these rules may degrade performance. But the use of these roles may also improve performance if preference is given to special-case rules as we discuss in further chapters.

We have now discussed two quite different problems, chess and the water jug problem. From these discussions, it should be clear that the first step toward the design of a program to solve a problem must be the creation of a formal and manipulatable description of the problem itself. Ultimately, we would like to be able to write programs that can themselves produce such formal descriptions from informal ones. This process is called operationalization. It is not at all well-understood how to construct such programs. Until it becomes possible to automate this process, it must be done by hand, however. For simple problems, such as chess or the water Jug, this is not very difficult. The problems are artificial and highly structured. For other problems,, particularly naturally occurring ones, this step is much more difficult. Consider, for example, the task of specifying precisely what it means to understand an English sentence. Although such a specification must somehow be provided before we can design a program to solve ten problem, producing such a specification is itself a very hard problem. Although our ultimate goal is to be able to solve difficult, unstructured problems, such as natural language understanding, it is useful to explore simpler problems, such as the water jug problem, in order to gain insight into the details of methods that can form the basis for solutions to the harder problems.

Summarizing what we have just said, in order to provide a formal description of a problem, we must do the following:

1. Define a state space that contains all the possible configurations of the relevant objects (and perhaps some impossible ones). It is, of course, possible to define this space without explicitly enumerating all of the states it contains.

2. Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the initial states.

3. Specify one or more states that would be acceptable as solutions to the problem. These states are called goal states.

4. Specify a set of rules that describe the actions (operators) available. Doing this will require giving thought to the following issues:

• What unstated assumptions are present in the informal problem description?

• How general should the rules be?

• How much of the work required to solve the problem should be precompiled and represented in the rules?

The problem can then be solved by using the rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found. Thus the process of search is fundamental to the problem-solving process. The fact that search provides the basis for the process of problem solving does not, however, mean that other, more direct approaches cannot also be exploited. Whenever possible, they can be included as steps in the search by encoding them into the rules. For example, in the water jug problem, we use the standard arithmetic operations as single steps in the rules. We do not use search to find a number with the property that it is equal to y — (4 — x). Of course, for complex problems, more sophisticated computations will be needed. Search is a general mechanism that can be used when no more direct method is known. At the same time, it provides the framework into which more direct methods for solving subparts of a problem can be embedded.

1.2 Production Systems

Since search forms the core of many intelligent processes, it is useful to structure AI programs in a way that facilitates describing and performing the search process. Production systems provide such structures. A definition of a production system is given below. Do not be confused by other uses of the word production, such as to describe what is done in factories. A production system consists of:

• A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.[3]

• One or more knowledge/databases that contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.

• A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.

• A rule applier.

So far, our definition of a production system has been very general. It encompasses a great many systems, including our descriptions of both a chess player and a water jug problem solver. It also encompasses a family of general production system interpreters, including:

• Basic production system languages, such as OPS5 and ACT*.

• More complex, often hybrid systems called expert system shells, which provide complete (relatively speaking) environments for the construction of knowledge-based expert systems.

• General problem-solving architectures like SOAR, a system based on a specific set of cognitively motivated hypotheses about the nature of problem solving.

All of these systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved.

We have now seen that in order to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state space (including the start and goal states) and a set of operators for moving in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modeled as a production system. In the rest of this section, we look at the problem of choosing the appropriate control structure for the production system so that the search can be as efficient as possible.

1.2.1 Control Strategies

So far, we have completely ignored the question of how to decide which rule to apply next during the process of searching for a solution to a problem. This question arises since often more than one rule (and sometimes fewer than one rule) will have its left side match the current state. Even without a great deal of thought, it is clear that how such decisions are made will have a crucial impact on how quickly, and even whether, a problem is finally solved.

• The first requirement of a good control strategy is that it cause motion. Consider again the water jug problem of the last section. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem. We would continue indefinitely filling the 4-gallon jug with water Control strategies that do not cause motion will never lead to a solution.

• The second requirement of a good control strategy is that it he systematic. Here is another simple control strategy for the water jug problem: On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But we are likely to arrive at the same state several times during the process and to use many more steps than are necessary. Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for global motion (over the course of several steps) as well as for local motion (over the course of a single step). One systematic control strategy for the water jug problem is the following. Construct a tree with the initial state as its root. Generate all the offspring of the root by applying each of the applicable rules to the initial state. Figure 1.5 shows how the tree looks at this point. Now for each leaf node, generate all its successors by applying all the rules that are appropriate. The tree at this point is shown in Figure 1.6.[4] Continue this process until some rule produces a goal state. This process, called breadth-first search, can be described precisely as follows.

Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.

2. Until a goal state is found or NODE-LIST is empty do:

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

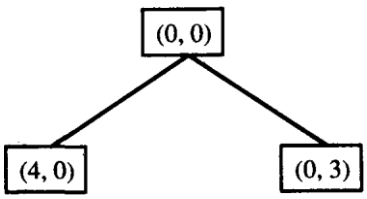


Figure 1.5: One Level of a Breadth-First Search Tree

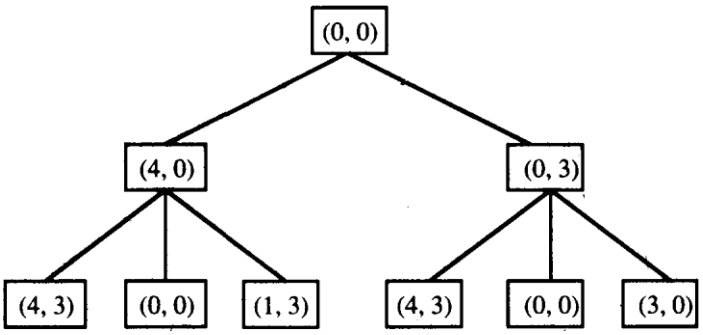


Figure 1.6: Two Levels of a Breadth-First Search Tree

(b) For each way that each rule can match the state described in E do:

i. Apply the rule to generate a new state. ii. If the new state is a goal state, quit and return this state. iii. Otherwise, add the new state to the end of NODE-LIST.

Other systematic control strategies are also available. For example, we could purse a single branch of the tree until it yields a solution or until a decision to terminate the(path is made. It makes sense to terminate a path if it reaches a dead-end, produces i previous state, or becomes longer than some pre-specified "futility" limit. In such a case backtracking occurs. The most recently created state from which alternative moves an available will be revisited and a new state will be created. This form of backtracking i; called chronological backtracking because the order in which steps are undone depend; only on the temporal sequence in which the steps were originally made. Specifically the most recent step is always the first to be undone. This form of backtracking is what is usually meant by the simple term backtracking. But there are other ways of retracting steps of a computation. We discuss one important such way, dependency-directs backtracking, in Chapter 7. Until then, though, when we use the term backtracking, i means chronological backtracking.

The search procedure we have just described is also called depth-first search. The< following algorithm describes this precisely.

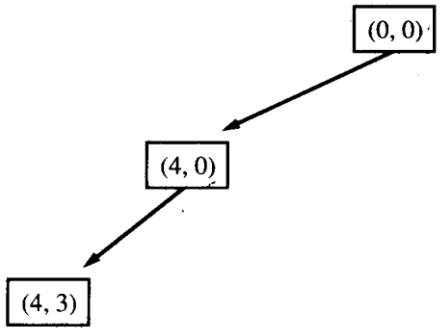


Figure 1.7: A Depth-First Search Tree

Algorithm: Depth-First Search

1. If the initial state is a goal state, quit and return success.

2. Otherwise, do the following until success or failure is signaled:

(a) Generate a successor, E, of the initial state. If there are no more successors, signal failure.

(b) Call Depth-First Search with E as the initial state.

(c) If success is returned, signal success. Otherwise continue in this loop.

Figure 1.7 shows a snapshot of a depth-first search for the water jug problem. A comparison of these two simple methods produces the following observations.

Advantages of Depth-First Search

• Depth-first search requires less memory since only the nodes on the current path are stored. This contrasts with breadth-

first search, where all of the tree that has so far been generated must be stored.

• By chance (or if care is taken in ordering the alternative successor states), depth-first search may find a solution without examining much of the search space at all. This contrasts with breadth-first search in which all parts of the tree must be examined to level n before any nodes on level n + 1 can be examined. This is particularly significant if many acceptable solutions exist. Depth-first search can stop when one of them is found.

Advantages of Breadth-First Search

• Breadth-first search will not get trapped exploring a blind alley. This contrast with depth-first searching, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors. This is a particular problem in depth-first search if there are loops (i.e., a state has a successor that is also one of its ancestors) unless special care is expended to test for such a situation. The example in Figure 1.7, if it continues always choosing the first (in numerical sequence) rule that applies, will have exactly this problem.

If there is a solution, then breadth-first search is guaranteed to find it. Furthermore, if there are multiple solutions, then a minimal solution (i.e., one that requires the minimum number of steps) will be found. This is guaranteed by the fact that longer paths are never explored until all shorter ones have already been examined. This contrasts with depth-first search, which may find a long path to a solution in one part of the tree, when a shorter path exists in some other, unexplored part of the tree.

Clearly what we would like is a way to combine the advantages of both of these methods. In other section we will talk about one way of doing this when we have some additional information. Later, in later section, we will describe an uninformed way of doing so.

For the water jug problem, most control strategies that cause motion and are systematic will lead to an answer. The problem is simple. But this is not always the case. In order to solve some problems during our lifetime, we must also demand a control structure that is efficient.

Consider the following problem.

The Traveling Salesman Problem:

A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

A simple, motion-causing and systematic control structure could, in principle, solve this problem. It would simply explore all possible paths in the tree and return the one with the shortest length. This approach will even work in practice for very short lists of cities. But it breaks down quickly as the number of cities grows. If there are N cities, then the number of different paths among them is 1 •2-- -(N- l),or(N- 1)!. The time to examine a single path is proportional to N. So the total time required to perform this search is proportional to Nl. assuming there are only 10 cities, 10! is 3,628,800, which is a very large number. The salesman could easily have 25 cities to visit. To solve this problem would take more time than he would be willing to spend. This phenomenon is called combinatorial explosion. To combat it, we need a new control strategy.

We can beat the simple strategy outlined above using a technique called branch-and-bound. Begin generating complete paths, keeping track of the shortest path found so far. Give up exploring any path as soon as its partial length becomes greater than the shortest path found so far. Using this technique, we are still guaranteed to find the shortest path. Unfortunately, although this algorithm is more efficient than the first one, it still requires exponential time. The exact amount of time it saves for a particular problem depends on

the order in which the paths are explored. But it is still inadequate for solving large problems.

1.2.2 Heuristic Search

In order to solve many hard problems efficiently, it is often necessary to compromise the requirements of mobility and systematicity and to construct a control structure that is no longer guaranteed to find the best answer but that will almost always find a very good answer. Thus we introduce the idea of a heuristic.[5] IA heuristic is a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness/ Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions; they are bad to the extent that they may miss points of interest to particular individuals. Some heuristics help to guide a search process without sacrificing any claims to completeness that the process might previously have had. Others (in fact, many of the best ones) may occasionally cause an excellent path to be overlooked. But, on the average, they improve the quality of the paths that are explored. Using good heuristics, we can hope to get good (though possibly non optimal) solutions to hard problems, such as the traveling salesman, in less than exponential time. There are some good general-purpose heuristics that are useful in a wide variety of problem domains. In addition, it is possible to construct special-purpose heuristics that exploit domain-specific knowledge to solve particular problems.

One example of a good general-purpose heuristic that is useful for a variety of combinatorial problems is the nearest neighbor heuristic, which works by selecting the locally superior alternative at each step. Applying it to the traveling salesman problem, we produce the following procedure:

1. Arbitrarily select a starting city.

2. To select the next city, look at all cities not yet visited, and select the one-closest to the current city. Go to it next.

3. Repeat step 2 until all cities have been visited.

This procedure executes in time proportional to N^2, a significant improvement over N, and it is possible to prove an upper bound on the error it incurs. For general-purpose heuristics, such as nearest neighbor, it is often possible to prove such error bounds, which provides reassurance that one is not paying too high a price in accuracy for speed.

In many AI problems, however, it is not possible to produce such reassuring bounds. This is true for two reasons:

- For real world problems, it is often hard to measure precisely the value of a particular solution. Although the length of a trip to several cities is a precise notion, the appropriateness of a particular response to such questions as "Why has inflation increased?" is much less so.

- For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is often impossible to define this knowledge in such a way that a mathematical analysis of its effect on the search process can be performed.

The word heuristic comes from the Greek word heuriskein, meaning "to discover," which is also the origin of eureka, derived from Archimedes' reputed exclamation, heurika (for "I have found"), uttered when he had discovered a method for determining the purity of gold.

There are many heuristics that, although they are not as general as the nearest neighbor heuristic, are nevertheless useful in a wide variety of domains. For example, consider the task of discovering interesting ideas in some specified area. The following heuristic [Lenat, 1983b] is often useful:

If there is an interesting function of two arguments/(A-, y), look at what happens if the two arguments are identical.

In the domain of mathematics, this heuristic leads to the discovery of squaring if/ is the multiplication function, and it leads to the discovery of an identity function if is the function

of set union. In less formal domains, this same heuristic leads to the discovery of introspection if/ is the function contemplate or it leads to the notion of suicide iff is the function kill.

Without heuristics, we would become hopelessly ensnarled in a combinatorial explosion. This alone might be a sufficient argument in favor of their use. But there are other arguments as well:

- Rarely do we actually need the optimum solution; a good approximation will usually serve very well. In fact, there is some evidence that people, when they solve problems, are not optimizers but rather are satisfiers -[Simon, 1981]. In other words, they seek any solution that satisfies some set of requirements, and as soon as they find one they quit. A good example of this is the search for a parking space. Most people stop as soon as they find a fairly good space, even if there might be a slightly better space up ahead.

- Although the approximations produced by heuristics may not be very good in the worst case, worst cases rarely arise in the real world. For example, although many graphs are not separable (or even nearly so) and thus cannot be considered as a set of small problems rather than one large one, a lot of graphs describing the real world are.[6]

- Trying to understand why a heuristic works, or why it doesn't work, often leads to a deeper understanding of the problem.

One of the best descriptions of the importance of heuristics in solving interesting problems is How to Solve It. Although the focus of the book is the solution of mathematical problems, many of the techniques it describes are more generally applicable. For example, given a problem to solve, look for a similar problem you have solved before. Ask whether you can use either the solution of that problem or the method that was used to obtain the solution to help solve the new problem. Polya's work serves as an excellent guide for people who want to become better problem solvers. Unfortunately, it is not a panacea for AI for a couple of reasons. One is that it relies on human abilities that we must first understand well enough to

build into a program. For example, many of the problems Polya discusses are geometric ones in which once an appropriate picture is drawn, the answer can be seen immediately. But to exploit such techniques in programs, we must develop a good way of representing and manipulating descriptions of those figures. Another is that the rules are very general.

They have extremely underspecified left sides, so it is hard to use them to guide a search too many of them are applicable at once. Many of the rules are really only useful for looking back and rationalizing a solution after it has been found. In essence, the problem is that Polya's rules have not been operationalized.

Nevertheless, Polya was several steps ahead of AI. A comment he made in the preface to the first printing of the book is interesting in this respect:

> The following pages are written somewhat concisely, but as simply as possible, and are based on a long and serious study of methods of solution. This sort of study, called heuristic by some writers, is not in fashion nowadays but has a long past and, perhaps, some future.

There are two major ways in which domain-specific, heuristic knowledge can be incorporated into a rule-based search procedure:

- In the rules themselves. For example, the rules for a chess-playing system might describe not simply the set of legal moves but rather a set of "sensible" moves, as determined by the rule writer.

- As a heuristic function that evaluates individual problem states and determines how desirable' they are.

(A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers} Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way

that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution. Sometime is very simple heuristic functions can provide a fairly good estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed. Figure 1.8 shows some simple heuristic functions for a few problems. Notice that sometimes a high value of the heuristic function indicates a relatively good position (as shown for chess and tic-tac-toe), while at other times a low value indicates an advantageous situation (as shown for the traveling salesman). It does not matter, in general, which way the function is stated. The program that uses the values of the function can attempt to minimize it or to maximize it as appropriate.

The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available. The more accurately the heuristic function estimates the true merits of each node in the search tree (or graph), the more direct the solution process. In the extreme, the heuristic function would be so good that essentially no search would be required. The system would move directly to a solution. But for many problems, the cost of computing the value of such a function would outweigh the effort saved in the search process, After all, it would be possible to compute a perfect heuristic function by doing a complete search from the node in question and determining whether it leads to a good solution,. In general, there is a trade-off between the cost of evaluating a heuristic function and the savings in search time that the function provides.

Chess	the material advantage of our side over the opponent
Traveling Salesman	the sum of the distances so far
Tic-Tac-Toe	1 for each row in which we could win and in which we already have one piece plus 2 for each such row in which we have two pieces

Figure 1.8: Some Simple Heuristic Functions

In the previous section, the solutions to AI problems were described as centering on a search process. From the discussion in this section, it should be clear that it can more precisely be described as a process of heuristic search. Some heuristics will be used to define the control structure that guides the application of rules in the search process. Others, as we shall see, will be encoded in the rules themselves. In both cases, they will represent either general or specific world knowledge that makes the solution of hard problems feasible. This leads to another way that one could define artificial intelligence: the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about the problem domain,

1.3 Problem Characteristics

Heuristic search is a very general method applicable to a large class of problems. It encompasses a variety of specific techniques, each of which is particularly effective for a small class of problems. In order to choose the most appropriate method (or combination of methods) for a; particular problem, it is necessary to analyze the problem along several key dimensions

- Is the problem decomposable into a set of (nearly) independent smaller or easier sub problems?

- Can solution steps be/ignored or at least undone if they prove unwise?
- Is the problem's universe predictable?
- Is a good solution to the problem obvious without comparison to all other possible solutions?
- Is the desired/Solution a state of the world or a path to a state?
- Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search?
- Can a computer that is simply given the problem return the solution, or will the solution of the problem require interaction between the computer and a person?

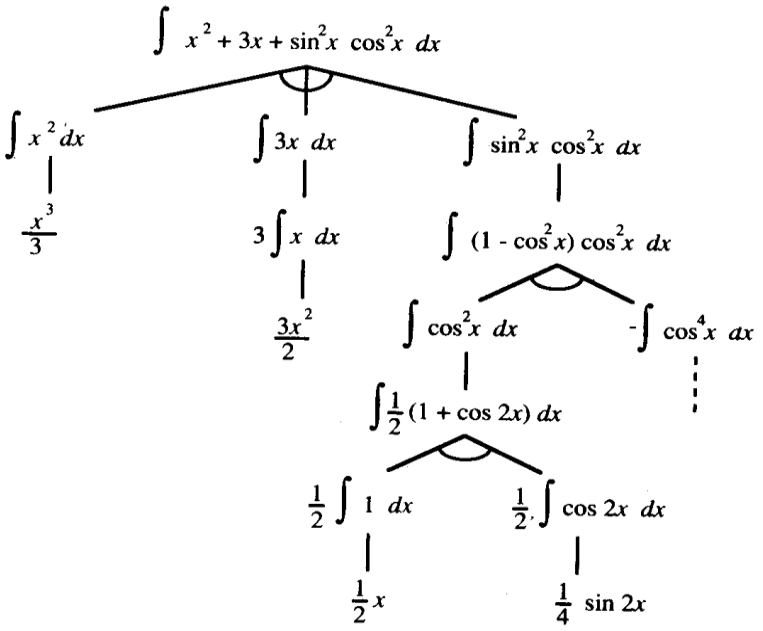


Figure 1.9: A Decomposable Problem

In the rest of this section, we examine each of these questions in greater detail. Notice that some of these questions involve not just the statement of the problem itself but also characteristics of the solution that is desired and the circumstances under which the solution must take place.

1.3.1 Is the Problem Decomposable?

Suppose we want to solve the problem of computing the expression

$$\int (x^2 + 3x + \sin^2 x \cdot \cos^2 x)\, dx$$

We can solve this problem by breaking it down into three smaller problems, each of which we can then solve by using a small collection of specific rules. Figure 1.9 shows the problem tree that will be generated by the process of problem decomposition as it can be exploited by a simple recursive integration program that works as follows: At each step, it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly. If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems. If it can, it creates those problems and calls itself recursively on them. Using this technique of problem decomposition, we can often solve very large problems easily.

Now consider the problem illustrated in Figure 1.10. This problem is drawn from the domain often referred to in AI literature as the blocks world. Assume that the following operators are available:

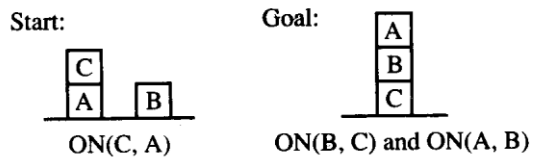


Figure 1.10: A Simple Blocks World Problem

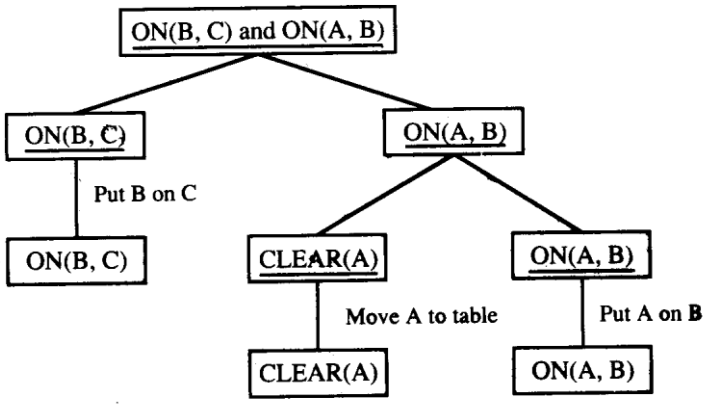


Figure 1.11: A Proposed Solution for the Blocks Problem

1. CLEARS) [block x has nothing on it] -<• ON(-c, Table) [pick up x and put it on the table]

2. CLEAR (JC) and CLEARS) -^ ON(x, y) [put .corny]

Applying the technique of problem decomposition to this simple blocks world example would lead to a solution tree such as that shown in Figure 1.11. In the figure, goals are underlined. States that have been achieved are not underlined. The idea of this solution is to reduce the problem of getting B on C and A on B to two separate problems, The first of these new problems, getting B on C, is simple, given the start state. Simply put B on C. The second sub goal is not quite so simple. Since the only operators we have allow us to pick up single blocks at a time, we have to clear off A by removing (^ before we can pick up A and put it on B. This can easily be done. However, if we now try to combine the two sub solutions into one solution, we will fail. Regardless of which one we do first, we will not be able to do the second as we had planned. In this problem) the two sub problems are not independent. They interact and those interactions must be considered in order to arrive at a solution for the entire problem.

These two examples, symbolic integration and the blocks world, illustrate the difference between decomposable and no decomposable problems.

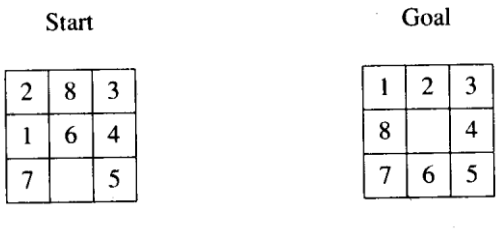


Figure 1.12: An Example of the 8-Puzzle

1.3.2 Can Solution Steps Be Ignored or Undone?

Suppose we are trying to prove a mathematical theorem. We proceed by first proving a lemma that we think will be useful. Eventually, we realize that the lemma is no help at all. Are we in trouble?

No. Everything we need to know to prove the theorem is still true and in memory, if it ever was. Any rules that could have been applied at the outset? can still be applied. We can just proceed as we should have in the first place. All we have lost is the effort that was spent exploring the blind alley.

Now consider a different problem.

The 8-Puzzle:

The 8-puzzle is a square tray in which are placed eight square tiles. The remaining ninth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

A sample game using the 8-puzzle is shown in Figure 1.12. In attempting to solve the 8-puzzle, we might make a stupid move. For example, in the game shown above, we might start by sliding tile 5 into the empty space. Having done that, we cannot change our mind and immediately slide tile 6 into the empty space since the empty space will essentially have moved. But we can backtrack and undo the first move, sliding tile 5 back to where it was, then we can move tile 6. Mistakes can still be recovered from but not quite as easily as in the theorem-proving problem. An additional step must be performed to undo each incorrect step, whereas no action was required to "undo" a useless lemma. In addition, the control mechanism for an 8 puzzle solver must keep track of the order in which operations are performed so that the operations can be undone one at a time if necessary. The control structure for a theorem proven does not need to record all that information.

Now consider again the problem of playing chess. Suppose a chess-playing program makes a stupid move and realizes it a couple of moves later. It cannot simply play as though it had never made the stupid move. Nor can it simply back up and start the game over from that point. All it can do is to try to make the best of the current situation and go on from there.

These three problems theorem proving, the 8-puzzle, and chess illustrate the differences between three important classes of problems:

• Ignorable (e.g., theorem proving), in which solution steps can be ignored

• Recoverable (e.g., 8-puzzle), in which solution steps can be undone

• Irrecoverable (e.g., chess), in which solution steps cannot be undone

These three definitions make reference to the steps of the solution to a problem and thus may appear to characterize particular production systems for solving a problem rather than the problem itself. Perhaps a different formulation of the

same problem would lead to the problem being characterized differently. Strictly speaking, this is true. But for a great many problems, there is only one (or a small number of essentially equivalent) formulations that naturally describe the problem. This was true for each of the problems used as examples above. When this is the case, it makes sense to view the recoverability of a problem as equivalent to the recoverability of a natural formulation of it.

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for the problem's solution. Ignorable problem can be solved using a simple control structure that never backtracks. Such a control structure is easy to implement. Recoverable problems can be solved by a slightly more complicated control strategy that does sometimes make mistakes. Backtracking will be necessary to recover from such mistakes, so the control structure must be implemented using a push-down stack, in which decisions are recorded in case they need to be undone later. Irrecoverable problems, on the other hand, will need to be solved by a system that expends a great deal of effort making each decision since the decision must be final. Some irrecoverable problems can be solved by recoverable style methods used in a planning process, in which an entire sequence of steps is analyzed in advance to discover where it will lead before the first step is actually taken. We discuss next the kinds of problems in which this is possible.

1.3.3 Is the Universe Predictable?

Again suppose that we are playing with the 8-puzzle. Every time we make a move, know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be. We can use planning to avoid having to undo actual moves, although it will still be necessary to backtrack past those moves one at a time during the planning process. Thus a CONTROL structure that allows backtracking will be necessary.

However, in games other than the 8-puzzle, this planning process may not be possible suppose we want to play bridge.

One of the decisions we will have to make is which card to play on the first trick. What we would like to do is to plan the entire hand before making that first play. But now it is not possible to do such planning with certainty since we cannot know exactly where all the cards are or what the other players will do on their turns. The best we can do is to investigate several plans and use probabilities of the various outcomes to choose a plan that has the highest estimated probability of leading to a good score on the hand.

These two games illustrate the difference between certain-outcome (e.g., 8-puzzle) in uncertain-outcome (e.g., bridge) problems. One way of describing planning is that it problem solving without feedback from the environment. For .solving certain-outcome problems, this open-loop approach will work fine since the result of an action can be predicted perfectly. Thus, planning can be used to generate a sequence of operators that is guaranteed to lead to a solution. For uncertain outcome problems, however, planning an at best generate a sequence of operators that has a good probability of leading to a Bullion. To solve such problems, we need to allow for a process of plan revision to take lace as the plan is carried out and the necessary feedback is provided. In addition to providing no guarantee of an actual solution, planning f0r uncertain-outcome problems as the drawback that it is often very expensive since the number of solution paths that need to be explored increases exponentially with the; number of points at which the outcome cannot be predicted.

The last two problem characteristics we have discussed, ignorable versus recoverable rises irrecoverable and certain-outcome versus uncertain-outcome, interact in an in-resting way. As has already been mentioned, one way to solve irrecoverable problems into plan an entire solution before embarking on an implementation of the plan. **But** his planning process can only be done effectively for certain-outcome problems. Thus ne of the hardest types of problems to solve is the irrecoverable, uncertain-outcome. I few examples of such problems are:

» Playing bridge. But we can do fairly well since we have available accurate estimates of the probabilities of each of the possible outcomes.

- Controlling a robot arm. The outcome is uncertain for a variety of reasons. Someone might move something into the path of the arm. The gears of the am might stick. A slight error could cause the arm to knock over a whole stack of things.

- Helping a lawyer decide how to defend his client against a murder charge. Here we probably cannot even list all the possible outcomes, much less assess their probabilities.

1.3.4 Is a Good Solution Absolute or Relative?

Consider the problem of answering questions based on a database of simple facts, such f the following:

1. Marcus was a man.

2. Marcus was a Pompeian.

3. Marcus was born in 40 A.D. j 4. Ail men are mortal. ». 5. All Pompeian's died when the volcano erupted in 79 A.D.

6. No-mortal lives longer than 150 years.

7. It is now 1991 A.D.

Suppose we ask the question "Is Marcus alive?" By representing each of these facts in a formal language, such as predicate logic, and then using formal inference methods,

		Justification
1.	Marcus was a man.	axiom 1
4.	All men are mortal.	axiom 4
8.	Marcus is mortal.	1, 4
3.	Marcus was born in 40 A.D.	axiom 3
7.	It is now 1991 A.D.	axiom 7
9.	Marcus' age is 1951 years.	3, 7
6.	No mortal lives longer than 150 years.	axiom 6
10.	Marcus is dead.	8, 6, 9

OR

7.	It is now 1991 A.D.	axiom 7
5.	All Pompeians died in 79 A.D.	axiom 5
11.	All Pompeians are dead now.	7, 5
2.	Marcus was a Pompeian.	axiom 2
12.	Marcus is dead.	11, 2

Figure 1.13: Two Ways of Deciding That Marcus Is Dead

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

Figure 1 14: An Instance of the Traveling Salesman Problem

We can fairly easily derive an answer to the question. In fact, either of two reasoning paths will lead to the answer, as shown in Figure 1.13. Since all we are interested in is the answer to the question, it does not matter which path we follow. If we do follow one path successfully to the answer, there is no reason to go back and see if some other path might also lead to a solution.

But now consider again the traveling salesman problem. Our goal is to find the shortest route that visits each city exactly once. Suppose the cities to be visited and the distances between them are as shown in Figure 1.14.

One place the salesman could start is Boston. In that case, one path that might be followed is the one shown in Figure 1.15, which is 8850 miles long. But is this the solution to the problem? The answer is that we cannot be sure unless we also try all other paths to make sure that none of them is shorter. In this case, as can be seen from Figure 1.16, the first path is definitely not the solution to the salesman's problem.

These two examples illustrate the difference between any-path problems and best-path problems. Best-path problems are, in general, computationally harder than any-path problems. Any-path problems can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore. (See the discussion of best-first search in Chapter 3 for one way of doing this.) If the heuristics are not perfect, the search for a solution may not be as direct as possible, but that does not matter. For true best-path problems, however, no heuristic that could possibly miss the best solution can be used. So a much more exhaustive search will be performed.

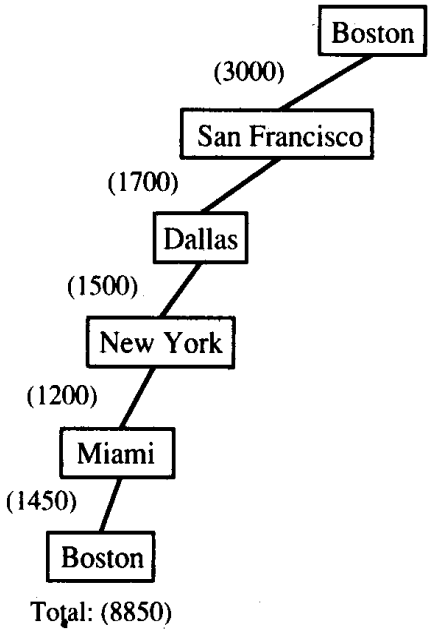


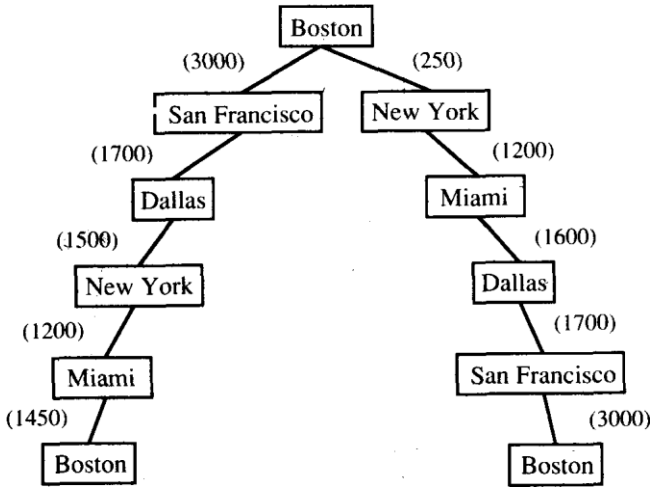
Figure 1.15: One Path among the Cities

1.3.5 Is the Solution a State or a Path?

Consider the problem of finding a consistent interpretation for the sentence. The bank president ate a dish of pasta salad with the fork.

There are several components of this sentence, each of which, in isolation, may have more than one interpretation. But the components must form a coherent whole, and so they constrain each other's interpretations. Some of the sources of ambiguity in this sentence are the following:

The word "bank" may refer either to a financial institution or to a side of a river. But only one of these may have a president.



Total: (8850) Total: (7750)

Figure 1.16: Two Paths Among the Cities

- The word "dish" is the object of the verb "eat." It is possible that a dish was eaten. But it is more likely that the pasta salad in the dish was eaten.

- Pasta salad is a salad containing pasta. But there are other ways meanings can be formed from pairs of nouns. For example, dog food does not normally contain dogs.

- The phrase "with the fork" could modify several parts of the sentence. In this case, it modifies the verb "eat." But, if the phrase had been "with vegetables," then the modification structure would be different. And if the phrase had been "with her friends," the structure would be different still.

Because of the interaction among the interpretations of the constituents of this sentence, some search may be required to find a complete interpretation for the sentence. But to solve the problem of finding the interpretation we need to produce only the interpretation itself. No record of the processing by which the interpretation was found is necessary.

Contrast this with the water jug problem. Here it is not sufficient to report that we have solved the problem and that the final state is (2,0). For this kind of problem, what we really must report is hot the final state but the path that we found to that state. Thus a statement of a solution to this problem must be a sequence of operations (sometimes called apian) that produces the final state.

These two examples, natural language understanding and the water jug problem, illustrate the difference between problems whose solution is a state of the world and problems whose solution is a path to a state. At one level, this difference can be ignored and all problems can be formulated as ones in which only a state is required to be reported. If we do this for problems such as the water jug, then we must re describe our states so that each state represents a partial path to a solution rather than just a single state of the world. So this question is not a formally significant one. But, just as for the question of ignitability versus recoverability, there is often a natural (and economical) formulation of a problem in which problem states correspond to situations in the world, not sequences of operations. In this case, the answer to this question tells us whether it is necessary to record the path of the problem-solving process as it proceeds.

1.3.6 What Is the Role of Knowledge?

Consider again the problem of playing chess. Suppose you had unlimited computing power available. How much knowledge would be required by a perfect program? The answer to this question is very little just the rules for determining legal moves and some simple control mechanism that implements an appropriate search procedure. Additional knowledge about such things as good strategy and tactics could of course help considerably to constrain the search and speed up the execution of the program.

But now consider the problem of scanning daily newspapers to decide which are supporting the Democrats and which are supporting the Republicans in some upcoming election. Again assuming unlimited computing power, how much knowledge would be required by a computer trying 'to solve this problem? This time the answer is a great deal. It would have to know such things as:

- The names of the candidates in each party.
- The fact that if the major thing you want to see done is have taxes lowered, you are probably supporting the Republicans.
- The fact that if the major thing you want to see done is improved education for minority students, you are probably supporting the Democrats.
- The fact that if you are opposed to big government, you are probably supporting the Republicans.
- And so on …

These two problems, chess and newspaper story understanding, illustrate the difference between problems for which a lot of knowledge is important only to constrain the search for a solution and those for which a lot of knowledge is required even to be able to recognize a solution.

1.3.7 Does the Task Require Interaction with a Person?

Sometimes it is useful to program computers to solve problems in ways that the majority of people would not be able to understand. This is fine if the level of the interaction between the computer and its human users is problem-in solution-out. But increasingly we are building programs that require intermediate interaction with people, both to provide additional input to the program and to provide additional reassurance to the user.

Consider, for example, the problem of proving mathematical theorems.

1. All we want is to know that there is a proof

2. The program is capable of finding a proof by itself

then it does not matter what strategy the program takes to find the proof. It can use, for example, the resolution procedure, which can be very efficient but which does not appear natural to people. But if either of those conditions is violated, it may matter very much how a proof is found. Suppose that we are trying to prove some new, very difficult theorem. We might demand a proof that follows traditional patterns so that a mathematician can read the proof and check to make sure it is correct. Alternatively, finding a proof of the theorem might be sufficiently difficult that the program does not know where to start. At the moment, people are still better at doing the high-level strategy required for a proof. So the computer might like to be able to ask for advice. For example, it is often much easier to do a proof in geometry if someone suggests the right line to draw into the figure. To exploit such advice, the computer's reasoning must be analogous to that of its human advisor, at least on a few levels. As computers move into areas of great significance to human lives, such as medical diagnosis, people will be very unwilling to accept the verdict of a program whose reasoning they cannot follow. Thus we must distinguish between two types of problems:

- Solitary, in which the computer is given a problem description and produces an answer with no intermediate communication and with no demand for an explanation of the reasoning process

- Conversational, in which there is intermediate communication between a person and the computer, either to provide additional assistance to the computer or to provide additional information to the user, or both

Of course, this distinction is not a strict one describing particular problem domains. As we just showed, mathematical theorem proving could be regarded as either. But for a particular application, one or the other of these types of systems will usually be desired and that decision will be important in the choice of a problem-solving method.

1.3.8 Problem Classification

When actual problems are examined from the point of view of all of these questions, it becomes apparent that there are several broad classes into which the problems fall. These classes can each be associated with a generic control strategy that is appropriate for solving the problem. For example, consider the generic problem of classification. The task here is to examine an input and then decide which of a set of known classes the input is an instance of. Most diagnostic tasks, including medical diagnosis as well as diagnosis of faults in mechanical devices, are examples of classification. Another example of a generic strategy is propose and refine. Many design and planning problems can be attacked with this strategy.

Depending on the granularity at which we attempt to classify problems and control strategies, we may come up with different lists of generic tasks and procedures. The important thing to remember here, though, since we are about to embark on a discussion of a variety of problem-solving methods, is that there is no one single way of solving all problems. But neither must each new problem be considered totally ab initio. Instead, if we analyze our problems carefully and sort our

problem-solving methods by the kinds of problems for which they are suitable, we will be able to bring to each new problem much of what we have learned from solving other, similar problems.

1.4 Production System Characteristics

We have just examined a set of characteristics that distinguish various classes of problems. We have also argued that production systems are a good way to describe the operations that can be performed in a search for a solution to a problem. Two questions we might reasonably ask at this point are:

1. Can production systems, like problems, be described by a set of characteristics that shed some light on how they can easily be implemented?

2. If so, what relationships are there between problem types and the types of production systems best suited to solving the problems?

The answer to the first question is yes. Consider the following definitions of classes of production systems. A monotonic production system is a production system in which the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected. A nonmonotonic production system is one in which this is not true. A partially commutative production system is a production system with the property that if the application of a particular sequence of rules transforms state x into state y, then any permutation of those rules that is allowable (i.e., each rule's preconditions are satisfied when it is applied) also transforms state x into state. A commutative production system is a production system that is both monotonic and partially commutative.

The significance of these categories of production systems lies in the relationship between the categories and appropriate implementation strategies. But before discussing that relationship, it may be helpful to make the meanings of the

definitions clearer by showing how they relate to specific problems.

Thus we arrive at the second question above, which asked whether there is an interesting relationship between classes of production systems and classes of problems. For any solvable problem, there exist an infinite number of production systems that describe ways to find solutions. Some will be more natural or efficient than others. Any problem that can be solved by any production system can be solved by a commutative one (our most restricted class), but the commutative one may be so unwieldy as to be practically useless* It may use individual states to represent entire sequences of applications of rules of a simpler, noncommutative system. So in a formal sense, there is no relationship between kinds of problems and kinds of production systems since all problems can be solved by all kinds of systems. But in a practical sense, there definitely is such a relationship between kinds of problems and the kinds of systems that lend themselves naturally to describing those problems. To see this, let us look at a few examples. Figure 1.17 shows the four categories of production systems produced by the two dichotomies, monotonic versus nonmonotonic and partially commutative versus nonpartially commutative, along with some problems that can naturally be solved by each type of system. The upper left comer represents commutative systems.

	Monotonic	Non-monotonic
Partially commutative	Theorem proving	Robot navigation
Not partially commutative	Chemical synthesis	Bridge

Figure 1.17: The Four Categories of Production Systems

Partially commutative, monotonic production systems are useful for solving ignorable problems. This is not surprising since the definitions of the two are essentially the same. But recall that ignorable problems are those for which a natural formulation leads to solution steps that can be ignored. Such a natural formulation will then be a partially commutative, monotonic system. Problems that involve creating new things rather than changing old ones are generally ignorable. Theorem proving, as we have described it, is one example of such a creative process. Making deductions from some known facts is a similar creative process. Both of those processes can easily be implemented with a partially commutative, monotonic system.

Partially commutative, monotonic production systems are important from an implementation standpoint because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed. Although it is often useful to implement such systems with backtracking in order to guarantee a systematic search, the actual database representing the problem state need not be restored. This often results in a considerable increase in efficiency, particularly because, since the database will never have to be restored, it is not necessary to keep track of wherein the Search process every change was made.

We have now discussed partially commutative production systems that are also monotonic. They are good for problems where things do not change; new things get created. Nonmonotonic, partially commutative systems, on the other hand, are useful for problems in which changes occur but can be reversed and in which order of operations is not critical. This is usually the case in physical manipulation problems, such as robot navigation on a flat plane. Suppose that a robot has the following operators: go north (N), go east (E), go south (S), and go west (W). To reach its goal, it does not matter whether the robot executes N-N-E or N-E-N. Depending on how the operators are chosen, the 8-Puzzle and the blocks world problem can also be considered partially commutative.

Both types of partially commutative production systems are significant from an implementation point of view because they tend to. lead to many duplications of individual states during the search process. This is discussed further in Section 1.5.

Production systems that are not partially commutative are useful for many problems in which irreversible changes occur. For example, consider the problem of determining a process to produce a desired chemical compound. The operators available include such things as "Add chemical x to the pot" or "Change the temperature to 1 degrees." These operators may cause irreversible changes to the potion being brewed. The order

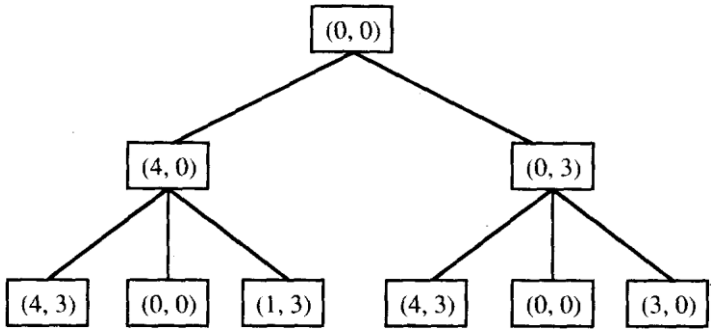


Figure 1.18: A Search Tree for the Water Jug Problem

in which they are performed can be very important in determining the final output. It is possible that if y is added to y, a stable compound will be formed, so later addition of 2 will have no effect; if : is added to y, however, a different stable compound may be formed, so later addition of will have no effect. No partially commutative production systems are less likely to produce the same node many times in the search process. When dealing with ones that describe irreversible processes, it is particularly important to make correct decisions the first time, although rather universe is predictable, planning can be used to make that less important.

1.5 Issues in the Design of Search Programs

Every search process can be viewed as a traversal of a tree structure in which each node represents a problem state and

each arc represents a relationship between the states represented by the nodes it connects. For example. Figure 1.18 shows part of a search tree for a water jug problem. The arcs have not been labeled in the figure, but they correspond to particular water-pouring operations. The search process must find a path or paths through the tree that connect an initial state with one or more final states. The tree that must be searched could, in principle, be constructed in its entirety from the rules that define allowable moves in the problem space. But, in practice, most of it never is. It is too large and most of it need never be explored. Instead of first building the tree explicitly and then searching it, most search programs represent the tree implicitly in the rules and generate explicitly only those parts that they decide to explore. Throughout our discussion of search methods, it is important to keep in mind this distinction between implicit search, trees and the explicit partial search trees that are actually constructed by the search program.

In the next chapter, we present a family of general-purpose search techniques. But before doing so we need to mention some important issues that arise in all of them:

• The direction in which to conduct the search (forward versus backward reasoning).

We can search forward through the state space from the start state to a goal state or we can search backward from the goal.

• How to select applicable rules (matching). Production systems typically spend most of their time looking for rules to apply, so it is critical to have efficient procedures for matching rules against states.

• How to represent each node of the search process (the knowledge representation problem and the frame problem). For problems like chess, a node can be fully represented by a simple array. In more complex problem solving, however, it is inefficient and/or impossible to represent all of the facts in the world and to determine all of the side effects an action may have.

We discuss the knowledge representation and frame problems further. We investigate matching and forward versus backward reasoning when we return to production systems.

One other issue we should consider at this point is that of search trees versus search graphs. As mentioned above, we can think of production rules as generating nodes in a search tree. Each node can be expanded in turn, generating a set of successors. This process continues until a node representing a solution is found. Implementing such a procedure requires little bookkeeping. However, this process often results in the same node being generated as pan of several paths and so being processed more than once. This happens because the search space may really be an arbitrary directed graph rather than a tree.

For example, in the tree shown in Figure 1.18, the node (4,3), representing 4 gallons of water in one jug and 3 gallons in the other, can be generated either by first filling the 4-gallon jug and then the 3-gallon one or by filling them in the opposite order. Since the order does not matter, continuing to process both these nodes would be redundant. This example also illustrates another problem that often arises when the search process operates as a tree walk. On the third level, the node (0, 0) appears. (In fact, it appears twice.) But this is the same as the top node of the tree, which has already been expanded. Those two paths have not gotten us anywhere. So we would like to eliminate them and continue only along the other branches.

The waste of effort that arises when the same node is generated more than once can be avoided at the price of additional bookkeeping. Instead of traversing a search tree, we traverse a directed graph. This graph differs from a tree in that several paths may come together at a node. The graph corresponding to the tree of Figure 1.18 is shown in Figure 1.19.

Any tree search procedure that keeps track of all the nodes that have been generated so far can be convened to a graph search procedure by modifying the action performed each time

a node is generated. Notice that of the two systematic search procedures we have discussed so far, this requirement that nodes be kept track of is met by breadth-first search but not by depth-first search. But, of course, depth-first search could be modified, at the expense of additional storage, to retain in memory nodes that have been expanded and then backed-up over. Since all nodes are saved in the search graph, we must use the following algorithm instead of simply adding a new node to the graph.

Algorithm: Check Duplicate Nodes

1. Examine the set of nodes that have been created so far to see if the new node already exists.

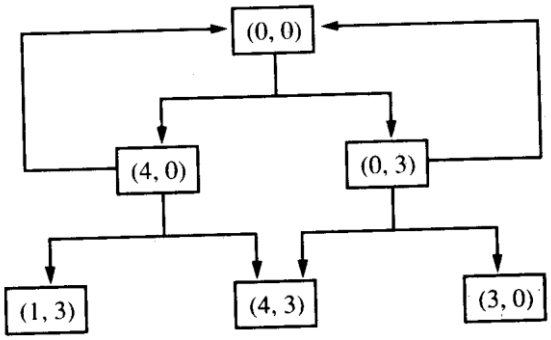


Figure 1.19: A Search Graph for the Water Jug Problem

2. If it does not/simply add it to the graph just as for a tree.

3. If it does already exist, then do the following:

(a) Set the node that is being expanded to point to the already existing node corresponding to its successor rather than to the new one. The new one can simply be thrown away.

(b) If you are keeping track of the best (shortest or otherwise least-cost) path to each node, then check to see if the new path is better or worse than the old one. If worse, do nothing. If better, record the new path as the correct path to use to get to

the node and propagate the corresponding change in cost down through successor nodes as necessary.

One problem that may arise here is that cycles may be introduced into the search graph. A cycle is a path through the graph in which a given node appears more than once.

Treating the search process as a graph search rather than as a tree Search reduces the amount of effort that is spent exploring essentially the same path several times. But it requires additional effort each time a node is generated to see if it has been generated before. Whether this effort is justified depends on the particular problem. If it is very likely that the same node will be generated in several different ways, then it is more worthwhile to use a graph procedure than if such duplication will happen only rarely.

Graph search procedures are especially useful for dealing with partially commutative production systems in which a given set of operations will produce the same result regardless of the order in which the operations are applied. A systematic search procedure will try many of the permutations of these operators and so will generate the same node many times. This is exactly what happened in the water jug example shown above.

1.6 Additional Problems

Several specific problems have been discussed throughout this chapter. Other problems have not yet been mentioned, but are common throughout the AI literature. Some have become such classics that no AI book could be complete without them, so we present them in this section. A useful exercise, at this point, would be to evaluate each of them in light of the seven problem characteristics we have just discussed.

A brief justification is perhaps required before this parade of toy problems is presented. Artificial intelligence is not merely a science of toy problems and micro worlds (such as the blocks world). Many of the techniques that have been developed for these problems have become the core of systems that solve very monotony problems. So think about these problems not

as defining the scope of Al but rather as providing a core from which much more has developed.

The Missionaries and Cannibals Problem

Three missionaries and three cannibals find themselves on one side to a river. They have agreed that they would all like to get to the other side. But the/missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two people at a time. How can everyone get across the rivet without the missionaries risking being eaten?

The Tower of Hanoi

Somewhere near Hanoi there is a monastery whose monks devote their lives to a very important task. In their courtyard are three tall posts. On these posts is a set of sixty-four disks, each with a hole in the center and each of a different radius. When the monastery was established, all of the disks were on one of the posts, each disk resting on the one just larger than it. The monks' task is to move all of the disks to one of the other pegs. Only one disk may be moved at a time, and all the other disks must be on one of the pegs. In addition, at no time during the process may a disk be placed on top of a smaller disk. The third peg can, of course, be used as a temporary resting place for the disks. What is the quickest way for the monks to accomplish their mission?

Even the best solution to this problem will take the monks a very long time. This is fortunate, since legend has it that the world will end when they have finished.

The Monkey and Bananas Problem

A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey,

unfortunately, cannot reach the bananas. However, in the room there are 'also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey to take to acquire lunch?

```
  SEND          DONALD        CROSS
 +MORE         +GERALD       +ROADS
 -----         -------       ------
 MONEY          ROBERT       DANGER
```

Figure 1.20: Some Cryptarithmetic Problems

Cryptarithmetic

Consider an arithmetic problem represented in letters, as shown in the examples in Figure 1.20. Assign a decimal digit to each of the letters in such a way that the answer to the problem is correct. If the same letter occurs more than once, it must be assigned the same digit each time. No two different letters may be assigned the same digit.

People's strategies for solving Cryptarithmetic problems have been, studied intensively by Newell and Simon.

1.7 Summary

In this chapter we have discussed the first two steps that must be taken toward the design of a program to solve a particular problem:

1. Define the problem precisely. Specify the problem space, the operators for moving within the space, and the starting and goal state(s).

2. Analyze the problem to determine where it falls with respect to seven important issues.

The last two steps for developing a program to solve that problem are. of course:

3. Identify and represent the knowledge required by the task.

4. Choose one or more techniques for problem solving, and apply those techniques to the problem.

Several general-purpose, problem-solving techniques are presented in the next chapter, and several of them have already been alluded to in the discussion of the problem characteristics in this chapter. The relationships between problem characteristics and specific techniques should become even clearer as we go on. Then, in later art II, we discuss the issue of how domain knowledge ‘is to be represented.

1.9. Model Questions

1. In this chapter the following problem were mentioned

    a. chess

    b. water jug

    c. 8-puzzel

    d. travelling salesman

    e. missionaries and cannibals

    f. tower of hanio

    g. monkey and bananas

    h. cryptarithmetic

    i. bridge

Before we can solve a problem using state space search, we must define an appropriate state space. For each of the problems mentioned above for which it was not done in the text. Find a good state space representation.

2. Describe how the branch-and-bound technique be used to find the shortest solution to a water jug problem.

3 For each of the following types of problems, try to describe a good heuristic function.

a. blocks of word

b. Theorem proving

c. missionaries and cannibals

4. Give an example of a problem for which breadth –first search would work better than depth-first search. Give an example of a problem for which depth-first search would work better than breadth-first search.

5. Write an algorithm to perform breadth-first search of a problem graph. Make sure your algorithm works properly when a single node is generated at more than one level in the graph.

UNIT – II

HEURISTIC SEARCH TECHNIQUES

Objectives

In this chapter we will discuss about various heuristic search techniques.

- Generate - and – Test
- Hill Climbing
- Best first Search
- Problem Reduction
- Constrained Satisfaction
- Means – ends Analysis

Introduction

In the last chapter, we saw that many of the problems that fall within the purview of artificial intelligence are too complex to be solved by direct techniques; rather they must be attacked by appropriate search methods armed with whatever direct techniques are available to guide the search. In this chapter, a framework for describing search methods is provided and several general-purpose search techniques are discussed. These methods are all varieties of heuristic search. They can be described independently of any particular task or problem domain. But when applied to particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search

processes are so vulnerable. For this reason, these techniques are often called weak methods. Although a realization of the limited effectiveness of these weak methods to solve hard problems by themselves has been an important result that emerged from the last three decades of AI research, these techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning. Thus they continue to form the core of most AI systems. We have already discussed two very basic search strategies:

• Depth-first search

• Breadth-first search In the rest of this chapter, we present some others:

• Generate-and-test

• Hill climbing

• Best-first search

• Problem reduction

• Constraint satisfaction

• Means-ends analysis

2.1 Generate-and-Test

The generate-and-test strategy is the simplest of all the approaches we discuss. It consists of the following steps:

Algorithm: Generate-and-Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.

3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" may be a very long time.

The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can, of course, also operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found. In this form, it is also known as the British Museum algorithm, a reference to a method for finding an object in the British Museum by wandering randomly.' Between these two extremes lies a practical middle ground in which the search process proceeds systematically, but some paths are not considered because they seem unlikely to lead to a solution.

The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, as described above, to traverse a graph rather than a tree.

For simple problems, exhaustive generate-and-test is often a reasonable technique. For example, consider the puzzle that

consists of four six-sided cubes, with each side of each cube painted one of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person (who is a much slower processor for this son of thing than even a very cheap computer) in several minutes by systematically and exhaustively trying all possibilities. It can be solved even .more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. As many of them as possible should be placed to abut the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

Or, as another story goes, if a sufficient number of monkeys were placed in front of a set of typewriters and left alone long enough, then they would eventually produce all of the works of Shakespeare.

Unfortunately, for problems much harder than this, even heuristic generate-and-test, all by itself, is not a very effective technique. But when combined with other techniques to restrict the space in which to search even further, the technique can be very effective.

For example, one early example of a successful AI program is DENDRAL which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data. It uses a strategy called plan-generate-test, in which a planning process that uses constraint-satisfaction techniques creates lists of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way, the generate-and-test procedure has proved highly effective.

This combination of planning, using one problem-solving method (in this case, constraint satisfaction) with the use of

the plan by another problem-solving method, generate-and-test, is an excellent example of the way techniques can be combined to overcome the limitations that each possesses individually. A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only to produce pieces of solutions that will then be exploited in the generate-and-test process, the lack of detailed accuracy becomes unimportant. And, at the same time, the combinatorial problems that arise in simple generate-and-test are avoided by judicious reference to the plans.)

2.2 Hill Climbing

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. But if the test function is augmented with a heuristic function[2] that provides an estimate of how close a given state is to a goal state, the generate procedure can exploit it as shown in the procedure below. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

2.2.1 Simple Hill Climbing

The simplest way to implement hill climbing is as follows.

Algorithm: Simple Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until there are no new operators left to be applied in the current state:

(a) Select an operator that has riot yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state.

If it is a goal state, then return it and quit. ii. If it is not a goal state but it is better than the current state, then make it the current state. iii. If it is not better than the current state, then continue in the loop.

The key difference between this algorithm and the one we gave for generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process. It is the use of such knowledge that makes this and the other methods discussed in the rest of this chapter heuristic search methods, and it is that same knowledge that gives these methods their power to solve some otherwise intractable problems.

Notice that in this algorithm, we have asked the relatively vague question, "Is one state better than another?" For the algorithm to work, a precise definition of better must be provided. In some cases, it means a higher value of the heuristic function. In others, it means a lower value. It does not matter which, as long as a particular hill-climbing program is consistent in its interpretation.

To see how hill climbing works, let's return to the puzzle of the four colored blocks. To solve the problem, we first need to define a heuristic function that describes how close a particular configuration is to being a solution. On such

function it^ simply the sum of the number of different colors on each of the four sides. A solution to the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration into another. Actually, one rule will suffice. It says simply pick a block and rotate it 90 degrees in any direction. Having provided these definitions, the next step is to generate a starting configuration. This can either be done at random or with the aid of the heuristic function described in the last section. Now hill climbing can begin. We generate a new state by selecting a block and rotating it. If the resulting state is better, then we keep it. If not, we return to the previous state and try a different perturbation.

2.2.2 Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called steepest-accent hill climbing or gradient search. Notice that this contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

Algorithm: Steepest-Ascent Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until a complete iteration produces no change to current state:

(a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.

(b) For each operator that applies to the current state do:

i. Apply the operator and generate a new state.

ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.

(c) If the SUCC is better than current state, then set current state to SUCC.

To apply steepest-ascent hill climbing to the colored blocks problem, we must consider all perturbations of the initial state and choose the best. For this problem, this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move (usually longer for steepest-ascent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem.

Both basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached a local maximum, a plateau, or a ridge.

A local maximum is a state that is better than all its neighbors but is not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called foothills.

A plateau is a flat area of the search space in which a whole set of neighboring states has the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

A ridge is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the High region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

There are some ways of dealing with these problems, although these methods are by no means guaranteed:

• Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.

• Make a big jump in some direction to try to get to a new section of the search space. This is a particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.

• Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a particularly good strategy for dealing with ridges.

Even with these first-aid measures, hill climbing is not always very effective. It is particularly unsuited to problems where the value of the heuristic function drops off suddenly as you move away from a solution. This is often the case whenever any sort of threshold effect is present. Hill climbing is a local method, by which we mean that it decides what to do next by looking only at the "immediate" consequences of its choice rather than-by exhaustively exploring all the consequences. It shares with other local methods, such as the nearest neighbor heuristic described in Section 1.2.2, the advantage of being less combinatorial explosive than comparable global methods. But it also shares with other local methods a lack of a guarantee that it will be effective. Although it is true that the hill-climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function. Consider the blocks world problem shown in Figure 3.1. Assume the same operators (i.e., pick up one block and put it on the table; pick up one block and put it on another one) that were used in Section 2.3.1. Suppose we use the following heuristic function:

Local: Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

Using this function, the goal state has a score of 8. The initial state has a score of 4 (since it gets one point added for blocks C, D, E, F, G, and H and one point subtracted for blocks A and B). There is only one move from the initial state, namely to move block A to the table. That produces a state with a score of 6 (since now A's position causes a point to be added rather than subtracted). The hill-climbing procedure will accept that move. From the new state, there are three possible moves, leading to the three states shown in Figure 2.2. These states have the scores: (o) 4, (h) 4, and (c) 4. Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not the global maximum. The problem is that by purely local examination of support structures, the current state appears to be better than any of its successors because more blocks rest on the correct objects. To solve this problem, it is necessary to disassemble a good local structure (the stack B through H) because it is in the wrong global context.

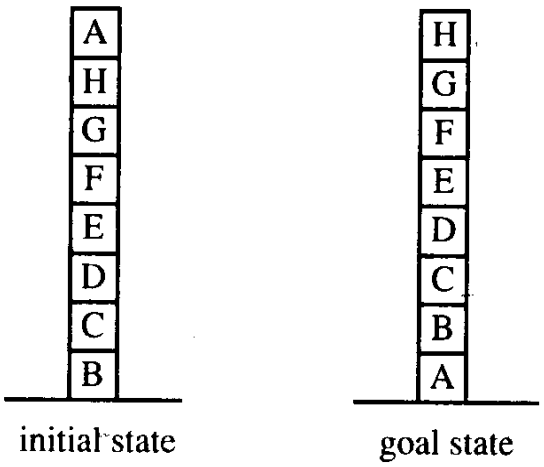


Figure 2.1: A Hill-Climbing problem

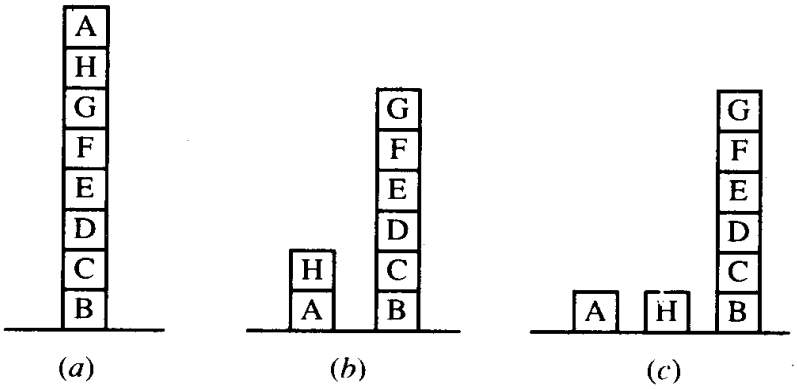


Figure 2.2: Three Possible Moves

We could blame hill climbing itself for this failure to look far enough ahead to find a solution. But we could also blame the heuristic function and try to modify it. Suppose we try the following heuristic function in place of the first one:

Global: For each block that has the correct support structure (i.e., the complete structure underneath it is exactly as it should be), add one point for every block in the support structure. For each block that has an incorrect support structure, subtract one point for every block in the existing support structure.

Using this function, the goal state has the score 28 (1 for B, 2 for C, etc.). The initial state has the score -28. Moving A to table yields a state with a score of -21 since A no longer has seven wrong blocks under it. The three Mates that can be produced next now have the following scores: (a) —28, (b) —16, and (c) —15. This time, steepest-ascent hill climbing will choose move (c), which is the correct one. This new heuristic function captures the two key aspects of this problem: incorrect structures are bad and should be taken apart; and correct structures are good and should be built up. As a result, the same hill climbing procedure that failed with the earlier heuristic function now works perfectly.

Unfortunately, it is not always possible to construct such a perfect heuristic function. For example, consider again the

problem of driving downtown. The perfect heuristic function would need to have knowledge about one-way and dead-end streets, which, in the case of a strange city, is not always available. And even if perfect knowledge is, in principle, available, it may not be computationally tractable to use. As an extreme example, imagine a heuristic function that computes a value for a state by invoking its own problem-solving procedure to look ahead from 'the state it is given to find a solution. It then knows the exact cost of finding that solution and can return that cost as its value. A heuristic function that does this converts the local hill-climbing procedure into a global method by embedding a global method within it. But now the computational advantages of a local method have been lost. Thus it is still true that hill climbing can be very inefficient in a large, rough problem space. But it is often useful when combined with other methods that get it started in the right general neighborhood.

2.2.3 Simulated Annealing

Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau, or a ridge.

In order to be compatible with standard usage in discussions of simulated annealing, we make two notational changes for the duration of this section. We use the term objective function in place of the term heuristic/unction.

And we attempt to minimize rather than maximize the value of the objective function. Thus we actually describe a process of valley descending rather than hill climbing.

Simulated annealing as a computational process is patterned after the physical process of annealing, in which physical substances such as metals are melted (i.e., raised to high energy levels) and then gradually cooled until some solid state is reached. The goal of this process is to produce a minimal-

energy final state. Thus this process is one of valley descending in which the objective function is the energy level. Physical substances usually move from higher energy configurations to lower ones. so the valley descending occurs naturally. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function

$$P = e^{-\Delta E/kt}$$

where A £ is the positive change in the energy levels T is the temperature, and k is Boltzmann's constant. Thus, in the physical valley descending that occurs during annealing, the probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Thus such moves are more likely during the beginning of the process when the temperature is high, and they become less likely at the end as the temperature becomes lower. One way to characterize this process is that downhill moves are allowed anytime. Large upward moves may occur early on, but as the process progresses, only relatively small upward moves are allowed until finally the process converges to a local minimum configuration.

The rate at which the system is cooled is called the annealing schedule. Physical annealing processes are very sensitive to the annealing schedule. If cooling occurs too rapidly, stable regions of high energy will form. In other words, a local but not global minimum is reached. If, however, a slower schedule is used, a uniform crystalline structure, which corresponds to a global minimum, is more likely to develop. But, if the schedule is too slow, time is wasted. At high temperatures, where essentially random motion is allowed, nothing useful happens. At low temperatures a lot of time may be wasted after the final structure has already been formed. The optimal annealing schedule for each particular annealing problem must usually be discovered empirically.

These properties of physical annealing can be used to define an analogous process of simulated annealing, which can be used (although not always effectively) whenever simple hill climbing

can be used. In this analogous process, A£ is generalized so that it represents not specifically the change in energy but more generally, the change in the value of the objective function, whatever it is. The analogy for kT is slightly less straightforward. In the physical process, temperature is a well-defined notion, measured in standard units. The variable k describes the correspondence between the units of temperature and the units of energy. Since, in the analogous process, the units for both E and T are artificial, it makes sense to incorporate k into T, selecting values for T that produce desirable behavior on the part of the algorithm. Thus we use the revised probability formula

$$P' = e^{-\Delta E/t}$$

But we still need to choose a schedule of values for T (which we still call temperature). We discuss this briefly below after we present the simulated annealing algorithm.

The algorithm for simulated annealing is only slightly different from the simple hill-climbing procedure. The three differences are:

• The annealing schedule must be maintained.

• Moves to worse states may be accepted.

• It is a good idea to maintain, in addition to the current state, the best state found so far. Then, if the final state is worse than that earlier state (because of bad luck in accepting moves to worse states), the earlier state is still available.

Algorithm: Simulated Annealing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Initialize BEST-SO-FAR to the current state.

3. Initialize T according to the annealing schedule.

4. Loop until a solution is found or until there are no new operators left to be applied in the current state.

(a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state. Compute

A£ = (value of current) — <value of new state)

• If the new state is a goal state, then return it and quit.

• If it is not a goal state but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.

• If it is not better than the current state, then make it the current state with probability// as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0,1]. If that number is less than p', then the move is accepted. Otherwise, do nothing.

(c) Revise T as necessary according to the annealing schedule.

5. Return BEST-SO-FAR, as the answer.

To implement this revised algorithm, it is necessary to select a;", annealing schedule, which has three components. The first is the initial value to be used for temperature. The second is the criteria that will be used to decide when the temperature of the system should be reduced. The third is the amount by which the temperature will be reduced each time it is changed. There may also be a fourth component of the schedule, namely, when to quit. Simulated annealing is often used to solve problems in which the number of moves from a given state is very large (such as the number of permutations that can be made to a proposed traveling salesman route). For such problems, it may not make sense to try all possible moves.

Instead, it may be useful to exploit some criterion involving the number of moves that have been tried since an improvement was found.

Experiments that have been done with simulated annealing on a variety of problems suggest that the best way to select an annealing schedule is by trying several and observing the effect on both the quality of the solution that is found and the rate at which the process converges. To begin to get a feel for how to come up with a schedule, the first thing to notice is that as T approaches zero, the probability of accepting a move to a worse state goes to zero and simulated annealing becomes identical to simple hill climbing. The second thing to notice is that what really matters in computing the probability of accepting a move is the ratio A E/T. Thus it is important that values of T be scaled so that this ratio is meaningful. For example, T could be initialized to a value such that, for an average A E, p' would be 0.5.

2.3 Best-First Search

Until now, we have really only discussed two systematic control strategies, breadth-first search and depth-first search (of several varieties). In this section, we discuss a new method, best-first search, which is a way of combining the advantages of both depth-first and breadth-first search into a single method.

2.3.1 OR Graphs

Depth-first search is good because it allows a solution to be found without all competing branches having to be expanded. Breadth-first search is good because it does not get trapped on dead-end paths. One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.

At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues. Usually what happens is that a bit of depth-first searching occurs as the most promising branch is explored. But eventually, if a solution is not found, that branch will start to look less promising than one of the top-level branches that had been ignored. At that point, the now more promising, previously ignored branch will be explored. But the old branch is not forgotten.. Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough that it is again the most promising path.

Figure 2.3 shows the beginning of a best-first search procedure. Initially, there is only one node, so it will be expanded. Doing so generates three new nodes. The heuristic function, which, in this example, is an estimate of the cost of getting to a solution from a given node, is applied to each of these new nodes. Since node D is the most promising, it is expanded next, producing two successor nodes, E and F. But then the heuristic function is applied to them. Now another path, that going through node B, looks more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J. At the next step, J will be expanded, since it is the most promising. This process can continue until a solution is found.

Notice that this procedure is very similar to the procedure for steepest-ascent hill climbing, with two exceptions. In hill climbing, one move is selected and all the others are rejected, never to be reconsidered. This produces the straight-line behavior that is characteristic of hill climbing. In best-first search, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising. Further, the best available state is selected in

best-first search, even if that state has a value that is lower than the value of the state that was just explored. This contrasts with hill climbing, which will stop if there are no successor states with better values than the current state.

Although the example shown above illustrates a best-first search of a tree, it is sometimes important to search a graph instead so that duplicate paths will not be pursued. An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain, in addition to a description of the problem state it represents, an indication of how promising it is, a parent link that points back to the best node from which it came, and a list of the nodes that were generated from it. The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors. We will call a graph of this sort an OR graph, since each of its branches represents an alternative problem-solving path.

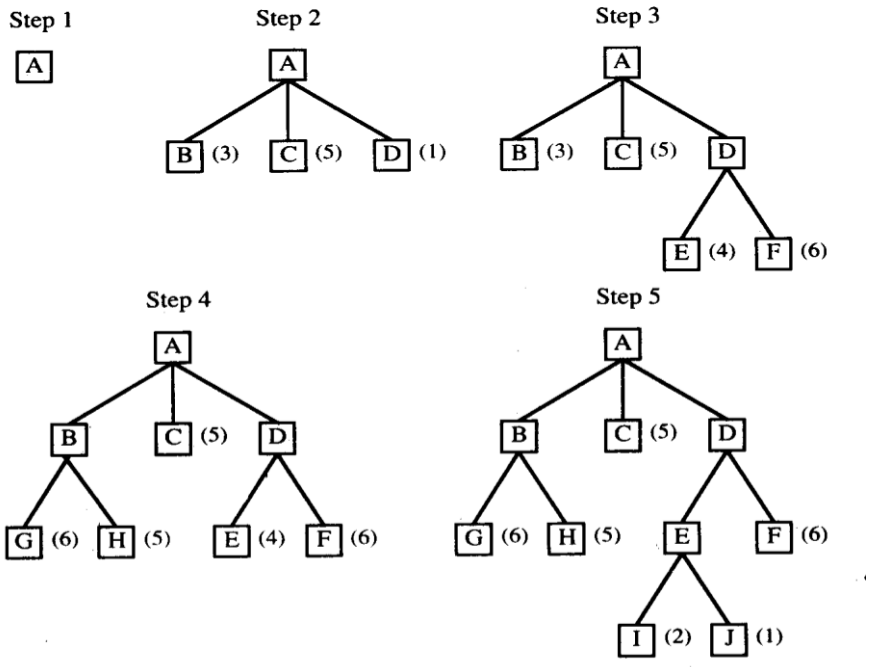


Figure 2.3: A Best-First Search

To implement such a graph-search procedure, we will need to use two lists of nodes:

• OPEN—nodes that have been generated and have had the heuristic function applied to thena but which have not yet been examined (i.e., had their successors generated). OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function. Standard techniques for manipulating priority queues can be used to manipulate the list.

• CLOSED—nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated; we need to check whether it has been generated before.

We will also need a heuristic function that estimates the merits of each node we generate. This will enable the algorithm to search more promising paths first. Call this function (to indicate that it is an approximation to a function/that gives the true evaluation of the node). For many applications, it is convenient to define this function as the sum of two components that we calling and h'. The function g is a measure of the cost of getting from the initial state to the current node. Note that g is not an estimate of anything; it is known to be the exact sum of the costs of applying each of the rules that were applied along the best path to the node. The function h' is an estimate of the additional cost of getting from the current node to a goal state. This is the place where knowledge about the problem domain is exploited. The combined function/', then, represents an estimate of the cost of getting from the initial state to a goal state along the path that generated the current node. If more than one path generated the node, then the algorithm will record the best one. Note that because g and h' must be added, it is important that h' be a measure of the cost of getting from the node to a solution (i.e., good nodes get low values; bad nodes get high values) rather than a measure of the goodness of a node (i.e., good nodes get high values). But that is easy to arrange with judicious placement of minus signs. It is also important that g be nonnegative. If this is not true, then paths that traverse cycles in the graph will appear to get better as they get longer.

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor. Then the next step begins.

This process can be summarized as follows.

Algorithm: Best-First Search

1. Start with OPEN containing just the initial state.

2. Until a goal is found or there are no nodes left on OPEN do:

(a) Pick the best node on OPEN.

(b) Generate its successors.

(c) For each successor do:

i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

The basic idea of this algorithm is simple. Unfortunately, it is rarely the case that graph traversal algorithms are simple to write correctly. And it is even rarer that it is simple to guarantee the correctness' of such algorithms. In the section that follows, we describe this algorithm in more detail as an example of the design and analysis of a graph-search program.

2.3.2 The A* Algorithm

The best-first search algorithm that was just presented is a simplification of an algorithm called A*, which was first presented by Hart et al. [1968; 1972]. This algorithm uses the same/', g, and h' functions, as well as the lists OPEN and CLOSED, that we have already described.

Algorithm: A*

1. Start with OPEN containing only the initial node. -Set that-node's g value to 0, its h' value to whatever it is, and its/' value to h' + 0, or h'. Set CLOSED to the empty list.

2. Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise, pick the node on OPEN with the lowest/' value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution (either BESTNODE if all we-want is the node or the path that has been created between the initial state and BESTNODE if we are interested in the path). Otherwise, generate the successors of BESTNODE but do not set BESTNODE to point to them yet. (First we need to see if any of them have already been generated.) For each such SUCCESSOR, do the following:

(a) Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.

(b) Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR.

(c) See if SUCCESSOR is the same as any node on OPEN (i.e., it has already been generated but not processed). If so, call that node OLD. Since this node already exists in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODE's successors. Now we must decide whether OLDS parent link should be reset to point to BESTNODE. It should be if the path we have just found to SUCCESSOR is cheaper

than the current best path to OLD (since SUCCESSOR and OLD are really the same node). So see whether it is cheaper to get to OLD via its current parent or to SUCCESSOR via BESTNODE by comparing their g values. If OLD is cheaper (or just as cheap), then we need do nothing, if SUCCESSOR is cheaper, then reset OLD'S parent link to point to BESTNODE, record the new cheaper path in g(OLD), and update /'(OLD).

(d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors. Check to see if the new path or the old path is better just as in step 2(t), and set the parent link and g and/' values appropriately. If we have just found a better path to OLD, we must propagate the improvement to OLD'S successors. This is a bit tricky. OLD points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still on OPEN or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node's g value (and thus also its/' value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found. This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

(e) If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute /'(SUCCESSOR) = g(SUCCESSOR) + ^(SUCCESSOR).

Several interesting observations can be made about this algorithm. The first concerns the role of the g function. It lets us choose which node to expand next on the basis not only of how good the node itself looks (as measured by h'), but also pn the basis of how good the path to the node was. By incorporating^ into/', we will not always choose as our next node to expand the node that appears to be closest to the goal. This is useful if we care about the path we find. If, on the other hand, we only care about getting to a solution somehow, we can define g always to be 0, thus always choosing the node that seems closest to a goal. If we want to find a path involving the fewest number of steps, then we set the cost of going from a node to its successor as a constant, usually 1. If, on the other hand, we want to find the cheapest path and some operators cost more than others, then we set the cost of going from one node to another to reflect those costs. Thus the A* algorithm can be used whether we are interested in finding a minimal-cost overall path or simply any path as quickly as possible.

The second observation involves h', the estimator of h, the distance of a node to the goal. If h' is a perfect estimator of h, then A* will converge immediately to the goal with no search. The better h' is, the closer we will get to that direct approach. It, on the other hand, the value of h' is always 0, the search will be controlled by g. If the value of g is also 0, the search strategy will be random. If the value of g is always 1, the search will be breadth first. All nodes on one level will have lower g values, and thus lower/' values than will all nodes on the next level. What if, on the other hand, h' is neither perfect nor O? Can we say anything interesting about the behavior of the search? The answer is yes if we care guarantee that h' never overestimates h. In that case, the A* algorithm is guaranteed to find an optimal (as determined by g) path to a goal, if one exists. This can easily be seen from a few examples.

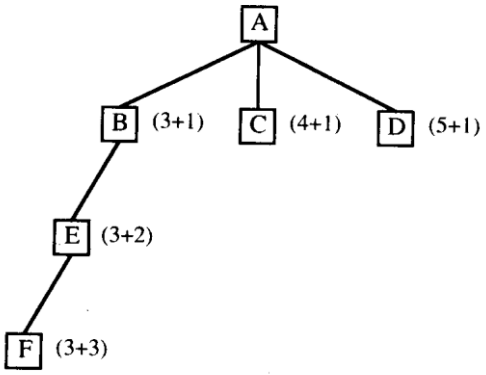


Figure 2.4: h' Underestimates h

Consider the situation shown in Figure 2.4. Assume that the cost of all arcs is I. Initially, all nodes except A are on OPEN (although the figure shows the situation two steps later, after B and E have been expanded). For each node,/' is indicated as the sum of h' and g. In this example, node B has the lowest/', 4, so it is expanded first. Suppose it has only one successor E, which also appears to be three moves away from a goal. Now /'(E) is 5, the same as /'(C). Suppose we resolve this in favor of the path we are currently following. Then we will expand E next. Suppose it too has a single successor F, also judged to be three moves from a goal. We are clearly using up moves and making no progress. But (F) = 6, which is greater than (C). So we will expand C next. Thus we see that by underestimating (B) we have wasted some effort. But eventually we discover that B was farther away than we thought and we go back and try another path.

Now consider the situation shown in Figure 2.5. Again we expand B on the first step. On the second step we again expand E. At the next step we expand F, and finally we generate G, for a solution path of length 4. But suppose there is a direct path from D to a solution, giving a path of length 2. We will never find it. By overestimating h'(D) we make D look so bad that we may find' some other, worse solution without ever expanding D.

Consider, for example, the task faced by the mathematics discovery program AM, written by Leant .AM was given a small set of starting facts about number theory and a set of operators it could use to develop new ideas. These operators included such things as "Find examples of a concept you already know." AM's goal was to generate new "interesting" mathematical concepts. It succeeded in discovering such things as prime numbers and Holdback's conjecture.

Armed solely with its basic operators, AM would have been able to create a great many new concepts, most of which would have been worthless. It needed a way to decide intelligently which rules to apply. For this it was provided with a set of heuristic rules that said such things as "The extreme cases of any concept are likely to be interesting." "Interest" was then used as the measure of merit of individual tasks that the system could perform. The system operated by selecting at each cycle the most interesting task, doing it, and possibly generating new tasks in the process. This corresponds to the selection of the most promising node in the best-first search procedure. But in AM's situation the fact that several paths recommend the same task does matter. Each contributes a reason why the task would lead to an interesting result. The more such reasons there are, the more likely it is that the task really would lead to 'something good. So we need a way to record proposed tasks along with the reasons they have been proposed. AM used a task agenda. An agenda is a list of tasks a system could perform. Associated with each task there are usually two things: a list of reasons why the task is being proposed (often called justifications) and a rating representing the overall weight of evidence suggesting that the task would be useful.

An agenda-driven system uses the following procedure.

Algorithm: Agenda-Driven Search

1. Do until a goal state is reached or the agenda is empty:

(a) Choose the most promising task from the agenda. Notice that this task can be represented in any desired form. It can be

thought of as an explicit statement of what to do next or simply as an indication of the next node to be expanded.

(b) Execute the task by devoting to it the number of resources determined .by its importance. The important resources to consider are time and space. Executing the task will probably generate additional tasks (successor nodes). For each of them, do the following:

i. See if it is already on the agenda. If so, then see if this same reason for doing it is already on its list of justifications. If so, ignore this current evidence. If this justification was not already present, add it to the list. If the task was not on the agenda, insert it.

ii. Compute the new task's rating, combining the evidence from all its justifications. Not all justifications need have equal weight. It is often useful to associate with each justification a measure of how strong areason it is. These measures are then combined at this step to produce an overall rating for the task.

One important question that arises in agenda-driven systems is how to find the most promising task on each cycle. One way to do this is simple. Maintain the agenda sorted by rating. When a new task is created, insert it into the agenda in- its proper place. When a task has its justifications changed, recompute its rating and move it to the correct place in the list. But this method causes a great deal of time to be spent keeping the agenda in perfect order. Much of this time is wasted since we do not need perfect order. We only need to know the proper first element. The following modified strategy may occasionally cause a task other than the best to be executed, but it is significantly cheaper than the perfect method. When a task is proposed, or a new justification is added to an existing task, compute the new rating and compare it against the top few (e.g., five or ten) elements on the agenda. If it is better, insert the node into its proper position at the top of the list. Otherwise, leave it where it is or simply insert it at the end of the agenda. At the beginning of each cycle, choose the first task on the agenda. In addition, once in a while, go through the agenda and reorder it properly.

An agenda-driven control structure is also useful if some tasks (or nodes) provide negative evidence about the merits of other tasks (or nodes). This can be represented by justifications with negative weightings. If these negative weightings are used, it may be important to check not only for the possibility of moving a task to the head of the agenda but also of moving a top task to the bottom if new, negative justifications appear. But this is easy to do.

As you can see, the agenda mechanism provides a good way of focusing the attention of a complex system in the areas suggested by the greatest number of positive indicators. But the overhead for each task executed may be fairly high. This raises the question of the proper grain size for the division of the entire problem-solving process into individual tasks. Suppose each task is very small. Then we will never do even a very small thing unless it really is the best thing to do. But we will spend a large percentage of our total effort on figuring out what to do next. If, on the other hand, the size of an individual task is very large, then some effort may be spent finishing one task when there are more promising ones that could be done. But a smaller percentage of the total time will be Spent on the overhead of figuring out what to do. The exact choice of task size for a particular system depends on the extent to which doing one small thing really means that a set of other small things is likely to be very good to do too. It often requires some experimentation to get right.

There are some problem domains for which an agenda mechanism is inappropriate. The agenda mechanism assumes that if there is good reason to do something now, then there will also be the same good reason to do something later unless something better comes along in the interim. But this is not always the case, particularly for systems that are interacting with people. The following dialogue would not be acceptable to most people:

Person: I don't want to read any more about China. Give me something else.

Computer: OK. What else are you interested in?

Person: How about Italy? I think I'd find Italy fascinating.

Computer: What things about Italy are you interested in reading about?

Person: I think I'd like to start with its history.

Computer: Why don't you want to read any more about China?

It would have been fine to have tried to find out why the person was no longer interested in China right after he or she mentioned it. The computer chose instead to try to find a new area of positive interest, also a very reasonable thing to do. But in conversations, the fact that something is reasonable now does not mean that it will continue to be so after the conversation has proceeded for a while. So it is not a good idea simply to put possible statements on an agenda, wait until a later lull, and then pop out with them. More precisely, agendas are a good way to implement monotonic production systems (in the sense of Section 1.4) and a poor way to implement nonmonotonic ones.

Despite these difficulties, agenda-driven control structures are very useful. They provide an excellent way of integrating information from a variety of sources into one program since each source simply adds tasks and justifications to the agenda. As Al programs become more complex and their knowledge bases grow, this becomes a particularly significant advantage.

2.4 Problem Reduction

So far, we have considered search strategies for OR graphs through which we want to find a single path to a goal. Such structures represent the fact that we will know how to get from a node to a goal state if we can discover how to get from that node to a goal state along any one of the branches leaving it.

2.4.1 AND-OR Graphs

Another kind of structure, the AND-OR graph (or tree), is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved. This is why the structure is called not simply an AND graph but rather an AND-OR graph. An example of an AND-OR graph (which also happens to be an AND-OR tree) is given in Figure 2.6. AND arcs are indicated with a line connecting all the components,

In order to find solutions in an AND-OR graph, we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately. This algorithm should find a path from the starting node of the graph to a set of nodes representing solution states. Notice that it may be necessary to get to more than one solution state since each arm of an AND arc must lead to its own solution node.

To see why our best-first search algorithm is not adequate for searching AND-OR graphs, consider Figure 2.7(a). The top node. A, has been expanded, producing two arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of/' at that node. We assume, for simplicity, that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with

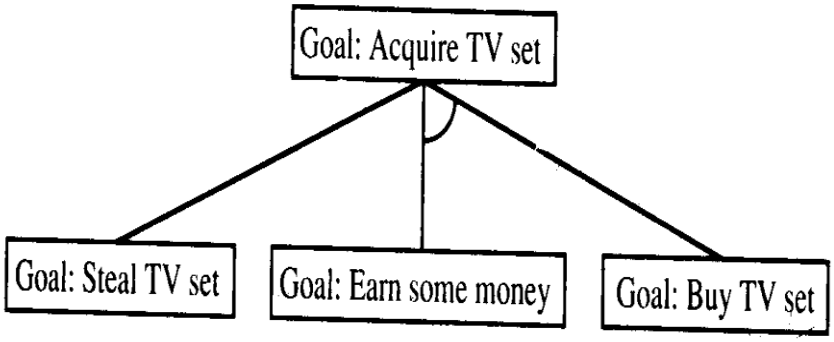


Figure 2.6: A Simple AND-OR Graph

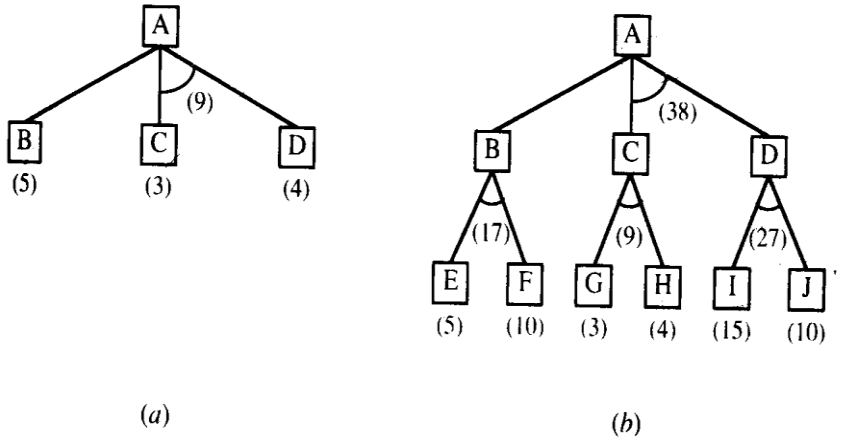


Figure 2.7: AND-OR Graphs

multiple successors has a cost of 1 for each of its components. If we look just at the nodes and choose for expansion the one with the lowest/' value, we must select C. But using the information now available, it would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C+D+2) compared to the cost of 6 that we get by going through B. The problem is that the choice of which node to expand next must depend not only on the/' value of that node but also on whether that node is part of the current best path from the initial node. The tree shown in Figure 2.7(b)

makes this even clearer. The most promising single node is G with an /' value of 3. It is even part of the most promising arc G-H, with a total cost of 9. But that arc is not part of the current best path since to use it we must also use the arc I-J, with a cost of 27. The path from A, through B, to E and F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

In order to describe an algorithm for searching an AND-OR graph, we need to exploit a value that we call. FUTILITY. If the estimated cost of a solution becomes greater than the value of FUTILITY, then we abandon the search. FUTILITY should be chosen to correspond to a threshold such that any solution with a cost above it is too expensive to be practical, even if it could ever be found. Now we can state the algorithm.

Algorithm: Problem Reduction

1. Initialize the graph to the starting node.

2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY:

(a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.

(b) Pick one of these unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph and for each of them compute /' (use only h' and ignore g, for reasons we discuss below). If/' of any node is 0, mark that node as SOLVED.

(c) Change the estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED. At each node that is visited while going up

the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propagation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their/' values be the best estimates available.

This process is illustrated in Figure 2.8. At step 1, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C, and D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9. (Marked arcs are indicated in the figures by arrows.) In step 2, node D is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the/' value of D to 10. Going back one more level, we see that this makes the AND arc B-C better than the arc to D, so it is labeled as the current best path. At step 3, we traverse that arc from A and discover the unexpanded nodes B and C. If we are going to find a solution along this path, we will have to expand both B and C eventually, so let's choose to explore B first. This generates two new arcs, the ones to G and to H. Propagating their/' values backward, we update/' of B to 6 (since that is the best we think we can do, which we can achieve by going through G). This requires updating the cost of the AND arc B-C to 12 (6+4+2). After doing that, the arc to D is again the better path from A, so we record that as the current best path and either node E or node F will be chosen for expansion at step 4. This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

In addition to the difference discussed above, there is a second important way in which an algorithm for searching an AND-OR graph must differ from one for searching an OR graph. This difference, too, arises from the fact that individual paths from node to node cannot be considered independently of the paths through other nodes connected

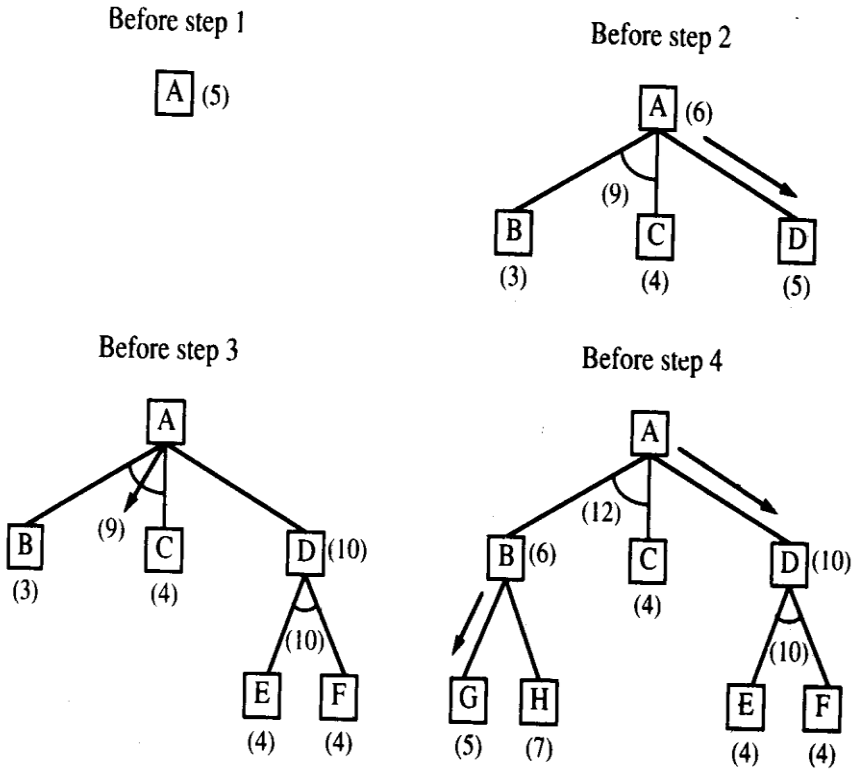


Figure 2.8: The Operation of Problem Reduction

to the original ones by AND arcs. In the best-first search algorithm, the desired path from one node to another was always the one with the lowest cost. But this is not always the case when searching an AND-OR graph.

Consider the example shown in Figure 2.9(a). The nodes were generated in alphabetical order. Now suppose that node J is expanded at the next step and that one of its successors is node E, producing the graph shown in Figure 2.9(b). This new path to E is longer than the previous path to E going through C. But since the path through C will only lead to a solution if there is also a solution to D, which we know there is not, the path through J is better.

There is one important limitation of the algorithm we have just described. It fails to take into account any interaction between sub goals. A simple example of this failure is shown in Figure

2.10. Assuming that both node C and node E ultimately lead to a solution, our algorithm will report a complete solution that includes both of them. The AND-OR graph states that for A to be solved, both C and D must he solved. But then the algorithm considers the solution of D as a completely separate process from the solution of C. Looking just at the alternatives from D, E is the best path. But it turns out that C is necessary anyway, so it would be better also to use it to satisfy D. But since our algorithm does not consider such interactions, it will find a nonoptimal path. problem-solving methods that can consider interactions among subgoals art presented.

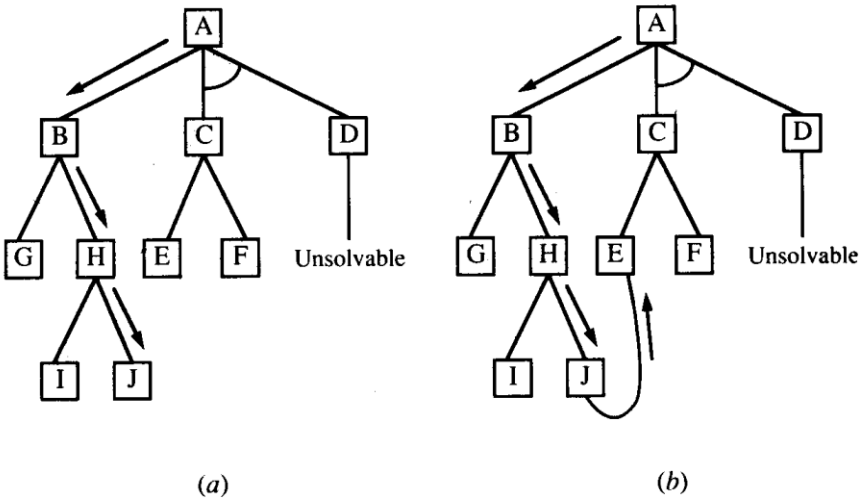


Figure 3.9: A Longer Path May Be Better

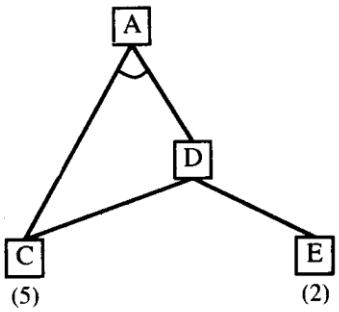


Figure 2.10: Interacting Subgoals

2.4.2 The AO* Algorithm

The problem reduction algorithm we just described is a simplification of an algorithm described in Martelli and Montanari [1973],MartelliandMontanari [1978],and Nilsson [)980]. Nilsson calls it the AO* algorithm, the name we assume.

Rather than the two lists, OPEN and CLOSED, that were used in the A* algorithm, the AO* algorithm will use a single structure GRAPH, representing the pan of the search graph that has been explicitly generated so far. Each node in the graph will point both down to its immediate successors and up to its immediate predecessors. Each node in the graph will also have associated with it an h' value, an estimate of the cost of a path from itself to a set of solution nodes. We will not store g (the cost of getting from the start node to the current node) as we did in the A* algorithm. It is not possible to compute a single such value since there may be many paths to the same state. And such a value is not necessary because of the top-down traversing of the best-known path, which guarantees that only nodes that are in the best path will ever be considered for expansion. So h' will serve as the estimate of goodness of a node.

Algorithm: AO*

1. Let GRAPH consist only <>the node representing the initial state. (Call this node INIT.) Compute h'(/NIT),

2. Until INIT is labeled SOLVED or until INlT's h' value becomes greater than FUTILITY, repeat the following procedure:

(a) Trace the labeled arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node NODE.

(b) Generate the successors of NODE. If there are none, then assign FUTILITY as the h' value of NODE. This is equivalent to saying that NODE is not solvable. If there are successors, then

for each one (called SUCCESSOR) that is not also an ancestor of NODE do the following:

i. Add SUCCESSOR to GRAPH. ii. If SUCCESSOR is a terminal

ii. node, label it SOLVED and assign it an h' value of 0.

iii. If SUCCESSOR is not a terminal node, compute its h' value.

(c) Propagate the newly discovered information up the graph by doing the

following: Let S be a set of nodes that have been labeled SOLVED or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to NODE. Until S is empty, repeat the following procedure:

i. If possible, select from S a node none of whose descendants in GRAPH occurs in S. If there is no such node, select any node from S. Call thisnode CURRENT, and remove it from S.

ii. Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as CURRENT'S new h' value the minimum of the costs just computed for the arcs emerging from it.

iii. Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.

iv. Mark CURRENT SOLVED if all of the nodes connected to it through the new, labeled arc have been labeled SOLVED.

v. If CURRENT has been labeled SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of CURRENT to S.

It is worth noticing a couple of points about the operation of this algorithm. In step 2(c)v, the ancestors of a node whose cost was altered are added to the set of nodes whose costs must also be revised. As stated, the algorithm will insert all the node's ancestors' into the set, which may result in the propagation of the cost change back up through a large number of paths that are already known not to be very good. For example. in Figure 2.11, it is clear that the path through C will always be better than the path through B, so work expended on the path through B is wasted. But if the cost of E is

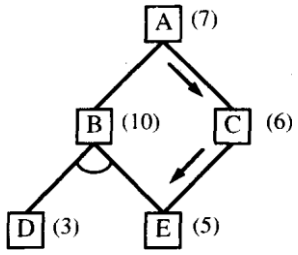


Figure 2.11: An Unnecessary Backward Propagation

revised and that change is not propagated up through B as well as through C, B may appear to be better. For example, if, as a result of expanding node E, we update its cost to 10, then the cost of C will be updated to 11. If this is all that is done, then when A is examined, the path through B will have a cost of only 11 compared to 12 for the path through C, and it will be labeled erroneously as the most promising path. In this example, the mistake might be detected at the next step, during which D will be expanded. If its cost changes and is propagated back to B, B's cost will be recomputed and the new cost of E will be used. Then the new cost of B will propagate back to A. At that point, the path through C will again be better. All that happened was that some time was wasted in expanding D. But if the node whose cost has changed is farther down in the search graph, the error may never be detected. An example of this is shown in Figure 2.12(a). If the cost of G is revised as shown in Figure 2.12(b) and if it is not immediately propagated back to E, then the change will never be recorded and a nonoptimal solution through B may be discovered.

A second point concerns the termination of the backward cost propagation of step 2(c). Because GRAPH may contain cycles, there is no guarantee that this process will terminate simply because it reaches the "top" of the graph. It turns out that the process can be guaranteed to terminate for a different reason, though.

2.5 Constraint Satisfaction

Many problems in AI can be viewed as problems of constraint satisfaction in which the goal is to discover some problem state that satisfies a given set of constraints. Examples of this sort of problem include crypt arithmetic puzzles (as described in Section 1.6) and many real-world perceptual labeling problems. Design tasks can also be viewed as constraint-satisfaction problems in which a design must be created within fixed limits on time, cost, and materials.

By viewing a problem as one of constraint satisfaction, it is often possible to reduce substantially the amount of search that is required as compared with a method that attempts to form partial solutions directly by choosing specific values for components of the eventual solution. For example, a straightforward search procedure to solve a crypt arithmetic problem might operate in a state space of partial solutions in which letters are assigned particular numbers as their values. A depth-first control scheme could then follow a path of assignments until either a solution or an inconsistency is discovered. In contrast to this, a constraint satisfaction approach to solving this problem avoids making guesses on particular assignments of numbers to letters until it has to. Instead, the initial set of constraints, which says that each number may correspond to only one letter and that the sums of the digits must be as they are given in the problem, is first augmented to include restrictions that can be inferred from the rules of arithmetic. Then, although guessing may still be required, the number of allowable guesses is reduced and so the degree of search is curtailed.

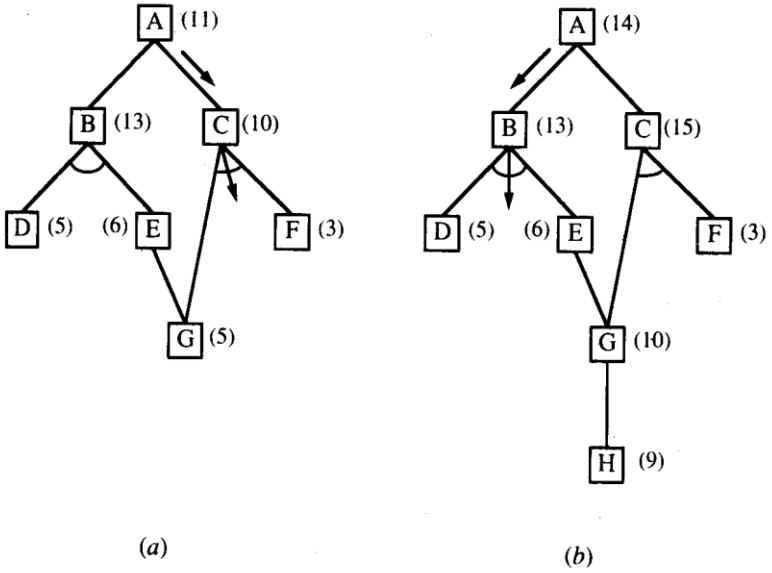


Figure 2.12: A Necessary Backward Propagation

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained "enough," where "enough" must be defined for each problem. For example, for crypt arithmetic, enough means that each letter has been assigned a unique numeric value.

Constraint satisfaction is a two-step process. First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.

The first step, propagation, arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint. So, for example, assume we start with one constraint, N = E + I. Then. if we added the constraint N = 3,

we could propagate that to get a stronger constraint on E, namely E = 2. Constraint propagation also arises from the presence of inference rules that allow additional constraints to be inferred from the ones that are given. Constraint propagation terminates for one of two reasons. First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints. If the contradiction involves only those constraints that were given as part of the problem specification (as opposed to ones that were guessed during problem solving), then no solution exists. The second possible reason for termination is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If this happens and a solution has not yet been adequately specified, then search is necessary to get the process moving again.

At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In the case of the crypt arithmetic problem, for example, this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still more guesses are required, they can be made. If a contradiction is detected, then backtracking can be used to try a different guess and proceed with it. We can state this procedure more precisely as follows:

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty:

(a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.

(b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been

examined, then add to OPEN all objects that share any constraints with OB.

(c) Remove OB from OPEN.

2. If the union of the constraints discovered above defines a solution, then quit and report the solution.

3. If the union of the constraints discovered above defines a contradiction, then return failure.

4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:

(a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.

(b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

This algorithm has been stated as generally as possible. To apply it in a particular problem domain requires the use of two kinds of rules: rules that define the way constraints may validly be propagated and rules that suggest guesses when guesses are necessary. It is worth noting, though, that in some problem domains guessing may not be required. For example, the Waltz algorithm for propagating line labels in a picture is a version of this constraint satisfaction algorithm with the guessing step eliminated. In general, the more powerful the rules for propagating constraints, the less need there is for guessing.

Problem:

```
  SEND
+MORE
-----
MONEY
```

Initial State:

No two letters have the same value.

The sums of the digits must be as shown in the problem.

Figure 2.13: A Crypt arithmetic Problem

To see how this algorithm works, consider the crypt arithmetic problem shown in Figure 2.13. The goal state is a problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied.

The solution process proceeds in cycles. At each cycle, two significant things are done (corresponding to steps 1 and 4 of this algorithm):

1. Constraints are propagated by using rules that correspond to the properties of arithmetic.

2. A value is guessed for some letter whose value is not yet determined.

In the first step, it does not usually matter a great deal what order the propagation is done in, since all available propagations will be performed before the step ends. In the second step, though, the order in which guesses are tried may have a substantial impact on the degree of search that is necessary. A few useful heuristics can help to select the best guess to try first. For example, if there is a letter that has only two possible values and another with six possible values, there

is a better chance of guessing right on the first than on the second. Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will usually lead quickly either to a contradiction (if it is wrong) or to the generation of many additional constraints (if it is right), A guess on a less constrained letter, on the other hand, provides less information.

The result of the first few cycles of processing this example is shown in Figure 2.14. Since constraints never disappear at lower levels, only the ones being added are shown for each level. It will not be much harder for the problem solver to access the constraints as a set of lists than as one long list, and this approach is efficient both in terms of storage space and the ease of backtracking. Another reasonable approach for this problem would be to store all the constraints in one central database and also to record at each node the changes that must be undone during backtracking. Cl, C2, C3, and C4 indicate the carry bits out of the columns, numbering from the right. Initially, rules for propagating constraints generate the following additional constraints:

- M = 1, since two single-digit numbers plus a carry cannot total more than 19.

- S = 8 or 9, since S + M + C3 > 9 (to generate the carry) and M = 1, S + 1 + C3 > 9, so S + C3 > 8 and C3 is at most 1.

- •; 0=0, since S +M(1) + C3 (<= 1) must be .at least 10 to generate a carry and it can be at most 11. But M is already 1, so 0 must be 0.

- N = E or E + 1, depending on the value of C2. But N cannot have the same value as E. So N = E + 1 and C2 is 1.

- In order for C2 to be 1, the sum of N + R + Cl must be greater than 9, so N + R must be greater than 8.

- N + R cannot be greater than 18, even with a carry in, so E cannot be 9.

At this point, let us assume that no more constraints can be generated. Then, to make progress from here, we must guess. Suppose E is assigned the value 2. (We chose to guess a value for E because it occurs three times and thus interacts highly with the other letters.) Now the next cycle begins.

The constraint propagator now deserves that:

- N = 3, **since N = E + 1.**

- R= 8 or 9, since R+N (3)+C1 (1 or O) = 2 or 12. But since N is already 3, the sum of these nonnegative numbers cannot be less than 3. Thus R + 3 + (0 or 1) = 12 and R =8 or 9.

- 2 + D = Y or 2 + D = 10 + Y, from the sum in the rightmost column.

Again, assuming no further constraints can be generated, a guess is required. Suppose Cl is chosen to guess a value for. If we try the value 1, then we eventually reach dead ends, as shown in the figure. When this happens, the process will backtrack and Cl =0.

A couple of observations are worth making on this process. Notice that all that is required of the constraint propagation rules is that they not infer spurious constraints. They do not nave to infer all legal ones. For example, we could have reasoned through to the result that Cl equals 0. We could have done so by observing that for Cl to be 1, the following must hold: 2 + D = 10 + Y. For this to be the case, D would have to be 8 or 9. But both S and R must be either 8 or 9 and three letters cannot share two values. So Cl cannot be 1. If we-had realized this initially, some search could have been avoided. But since the constraint propagation rules we used were not that sophisticated,

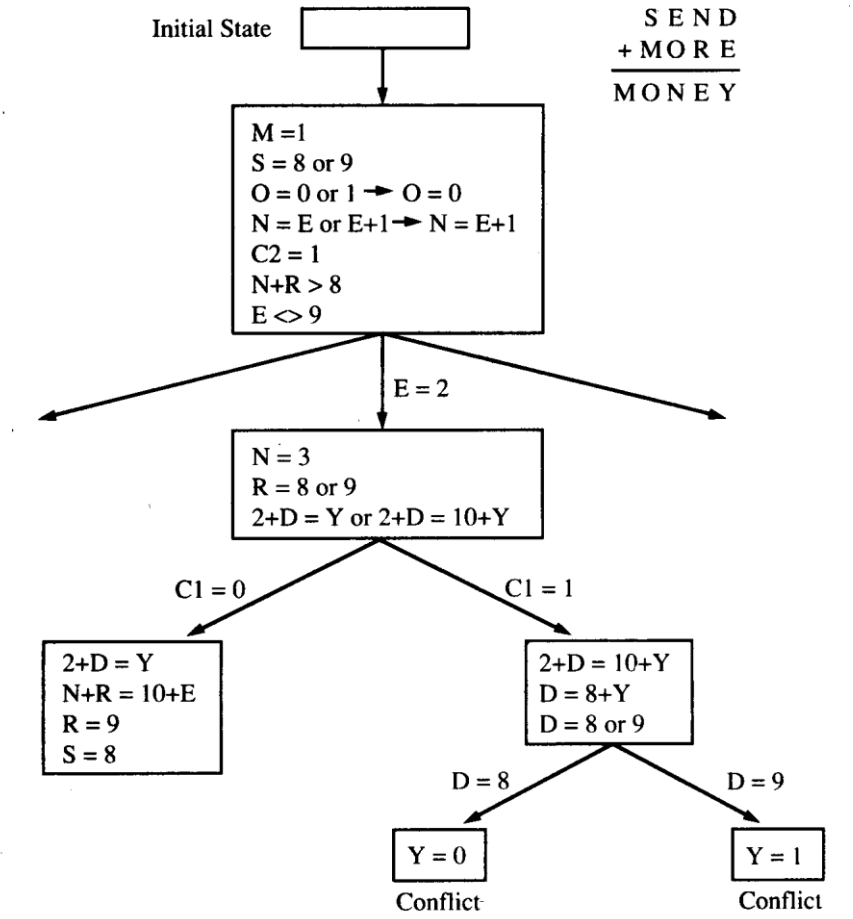


Figure 2.14: Solving a Cryptarithmetic problem

it took some search. Whether the search route takes more or less actual time than does the constraint propagation route depends on how long it takes to perform the reasoning required for constraint propagation.

A second thing to notice is that there are often two kinds of constraints. The first kind is simple; they just list possible values for a single object. The second kind is more complex; they describe relationships between or among objects. Both kinds of constraints play the same role in the constraint satisfaction process, and in the cryptarithmetic example they were treated identically. For some problems, however, it may

be useful to represent the two kinds of constraints differently. The simple, value-listing constraints are always dynamic, and so must always be represented explicitly in each problem state. The more complicated, relationship-expressing constraints are dynamic in the cryptarithmetic domain since they are different for each cryptarithmetic problem. But in many other domains they are static. For example, in the Waltz line labeling algorithm, the only binary constraints arise from the nature of the physical world, in which surfaces can meet in only a fixed number of possible ways. These ways are the same for all pictures that that algorithm may see. Whenever the binary constraints are static, it may be computationally efficient not to represent them explicitly in the state description but rather to encode them in the algorithm directly. When this is done, the only things that get propagated are possible values. But the essential algorithm is the same in both cases.

So far, we have described a fairly simple algorithm for constraint satisfaction in which chronological backtracking is used when guessing leads to an inconsistent set of constraints. An alternative is to use a more sophisticated scheme in which the specific cause of the inconsistency is identified and only constraints that depend on that culprit are undone. Others, even though they may have been generated after the culprit, are left alone if they are independent of the problem and its cause. This approach is called dependency-directed backtracking (DDB).

2.6 Means-Ends Analysis

So far, we have presented a collection of search strategies that can reason either forward or backward, but for a given problem, one direction or the other must be chosen. Often, however, a mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the major parts of a problem first and then go back and solve the small problems that arise in "gluing" the big pieces together. A technique known as means-ends analysis allows us to do that.

The means-ends analysis process centers around the detection of differences between the current state and the goal state.

Once such a difference is isolated, an operator that can reduce the difference must be found. But perhaps that operator cannot be applied to the current state. So we set up a subproblem of getting to a state in which it can be applied. The kind of backward chaining in which operators are selected and then subgoals are set up to establish the preconditions of the operators is called operator subgoaling. But maybe the operator does not produce exactly the goal state we want. Then we have a second subproblem of getting from the state it does produce to the goal. But if the difference was chosen correctly and if the operator is really effective at reducing the difference^ then the two subproblems should be easier to solve than the

Operator	*Preconditions*	*Results*
PUSH(obj, loc)	at(robot, obj) ∧ large(obj) ∧ clear(obj) ∧ armempty	at(obj, loc) ∧ at(robot, loc)
CARRY(obj, loc)	at(robot, obj) ∧ small(obj)	at(obj, loc) ∧ at(robot, loc)
WALK(loc)	none	at(robot, loc)
PICKUP(obj)	at(robot, obj)	holding(obj)
PUTDOWN(obj)	holding(obj)	¬ holding(obj)
PLACE(obj1, obj2)	at(robot, obj2) ∧ holding(obj1)	on(obj1, obj2)

Figure 2.15: The Robot's Operators

original problem. The means-ends analysis process can then be applied recursively. In order to focus the system's attention on the big problems first, the differences can be assigned priority levels. Differences of higher priority can then be considered before lower priority ones.

The first AI program to exploit means-ends analysis was the General Problem Solver (GPS) [Newell and Simon, 1963;-Ernst and Newell, 1969]. Its design was motivated by the observation

that people often use this technique when they solve problems. But GPS provides a good example of the fuzziness of the boundary between building programs that simulate what people do and building programs that simply solve a problem any way they can.

Just like the other problem-solving techniques we have discussed, means-ends analysis relies on a set of rules that can transform one problem state into another. These rules are usually not represented with complete state descriptions on each side. Instead, they are represented as a left side that describes the conditions that must be met for the rule to be applicable (these conditions are called the rule's preconditions)' and a right side that describes those aspects of the problem state that will be changed by the application of the rule. A separate data structure called a difference table indexes the rules by the differences that they can be used to reduce.

Consider a simple household robot domain. The available operators are shown in Figure 2.15, along with their preconditions and results. Figure 2.16 shows the difference table that describes when each of the operators is appropriate. Notice that sometimes there may be more than one operator that can reduce a given difference and that a given operator may be able to reduce more than one difference.

Suppose that the robot in this domain were given the problem of moving a desk with two things on it from one room to another. The objects on top must also be moved. The

	Push	Carry	Walk	Pickup	Putdown	Place
Move object	*	*				
Move robot			*			
Clear object				*		
Get object on object						*
Get arm empty					*	*
Be holding object				*		

Figure 2.16: A Difference Table

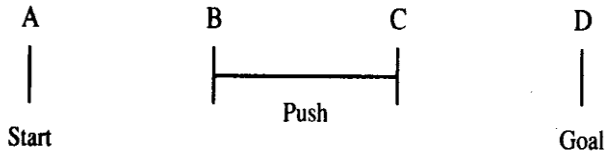


Figure 2.17: The Progress of the Means-Ends Analysis Method

main difference between the start state and the goal state would be the location of the desk. To reduce this difference, either PUSH or CARRY could be chosen. If CARRY is chosen first, its preconditions must be met. This results in two more differences that must be reduced: the location of the robot and the size of the desk. The location of the robot can be handled by applying WALK, but there are no operators than can change the size of an object (since we did not include SAW-APART). So this path leads to a dead-end. Following the other branch, we attempt to apply PUSH. Figure 2.17 shows the problem solver's progress at this point. It has found a way of doing something useful. But it is not yet in a position to do that thing. And the thing does not get it quite to the goal state. So now the differences between A and B and between C and D must be reduced.

PUSH has four preconditions, two of which produce differences between the start and the goal states: the robot must be at the desk, and the desk must be clear. Since the desk is already large, and the robot's arm is empty, those two preconditions can be ignored. The robot can be brought to the correct location by using WALK. And the surface of the desk can be cleared by two uses of PICKUP. But after one PICKUP, an attempt to do the second results in another difference—the arm must be empty. PUTDOWN can be used to reduce that difference.

Once PUSH is performed, the problem state is close to the goal state, but not quite. The objects must be placed back on the desk. PLACE will put them there. But it cannot be applied immediately. Another difference must be eliminated, since the robot must be holding the objects. The progress of the problem solver at this point is shown in Figure 2.18.

The final difference between C and E can be reduced by using WALK to get the robot back to the objects, followed by PICKUP and CARRY.

The process we have just illustrated (which we call MEA for short) can be summarized as follows:

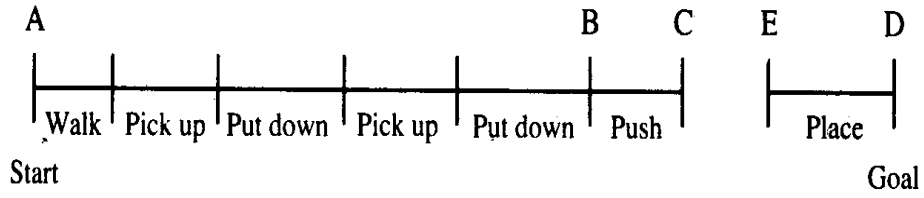


Figure 2.18: More Progress of the Means-Ends Method

Algorithm: Means-Ends Analysis (CURRENT, GOAL)

1. Compare CURRENT to GOAL. If there are no differences between them then return.

2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:

(a) Select an as yet untried operator 0 that is applicable to the current difference. If there are no such operators, then signal failure.

(b) Attempt to apply 0 to CURRENT. Generate descriptions of two states:

> 0-START, a state in which 0's preconditions are satisfied and 0-RESULT, the state that would result if 0 were applied in 0-START.

(c) If (FIRST-PART <- MEA(.CURRENT, 0-START)) and

> (LAST-PART <- MEA(0-RESULT, GOAL))

are successful, then signal success and return the result of concatenating

FIRST-PART, 0, and LAST-PART.

Many of the details of this process have been omitted in this discussion. In particular, the order in which differences are considered can be critical. It is important that significant differences be reduced before less critical ones. If this is not done. a great deal of effort may be wasted on situations that take care of themselves once the main parts of the problem are solved.

The simple process we have described is usually not adequate for solving complex problems. The number of permutations of differences may get too large. Working on one difference may interfere with the plan for reducing another. And in complex worlds, the required difference tables would be immense. In Chapter 13 we look at some ways in which the basic means-ends analysis approach can be extended to tackle some of these problems.

2.7 Summary

We listed four steps that must be taken to design a program to solve an AI problem. The first two steps were:

1. Define the problem precisely. Specify the problem space, the operators for moving within the space, and the starting and goal state(s).

2. Analyze the problem to determine where it falls with respect to seven important issues.

The other two steps were to isolate and represent the task knowledge required, and to choose problem solving techniques and apply them to the problem. In this chapter, we began our discussion of the last step of this process by presenting some general-purpose, problem-solving methods. There are several important ways in which these algorithms differ, including:

- What the states in the search space(s) represent. Sometimes the states represent complete potential solutions (as in hill climbing). Sometimes they represent solutions that are partially specified (as in constraint satisfaction).

- How, at each stage of the search process, a state is selected for expansion.

- How operators to be applied to that node are selected.

- Whether an optimal solution can be guaranteed.

- Whether a given state may end up being considered more than once.

- How many state descriptions must be maintained throughout the search process.

- Under what circumstances should a particular search path be abandoned.

In the chapters that follow, we talk about ways that knowledge about task domains can be encoded in problem-solving programs and we discuss techniques for combining problem-solving techniques with knowledge to solve several important classes of problems.

2.8. Model Questions

1. Discuss about Means – ends Analysis

2. when would best-first search be worse than sinple breadth-first search.

3. suppose we have a problem that we intend to solve using a heuristic best-first search procedure. We need to decide whether to implement it as a tree search or as a graph search. Suppose that we know that on the

average, each distinct node will be generated N times during the search process. We also know that if we use a graph, it will take, on the average, the same amount of time to check a node to see if it has already been generated as it takes to process M nodes if no checking is done. How can we describe whether to use a tree or a graph? In addition to the parameters N and M. what other assumptions must be made?

4. describe the behavior of a revised of the steepest ascent hill climbing algorithm in which step 2© is replaced by "set current state to best successor".

5. formalize the graceful decay of admissibility corollary and prove that it is true of the A* algorithm

6. consider again the AO* algorithm. Under what circumstances will it happen that there are nodes in S but there are no nodes in S that have no descendants also in S?

7. the constraint satisfaction procedure we have described performs depth-first search whenever some kind of search is necessary. But depth-first is not the only way to conduct such a search

    a. rewrite the constraint satisfaction procedure to use breadth-first search

    b. rewrite the constraint satisfaction procedure to use best-first search

7. show how means-ends analysis could be used to solve the problem of getting from one place to another, assume that the available operates are walk drive, take the bus, take a cab and fly.

UNIT – III

USING PREDICATE LOGIC

I In this chapter, we begin exploring one particular way of representing facts—the language of logic. Other representational formalisms are discussed in later chapters. The logical formalism is appealing because it immediately suggests a'powerful way of deriving new knowledge from old—mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.

One of the early domains in which AI techniques were explored was mechanical theorem proving, by which was meant proving statements in various areas of mathematics. For example, the Logic Theorist proved theorems from the tirst chapter of Whitehead and Russell's *Principia Mathematica* [1950]. Another iheorem prover proved theorems in geometry. Mathematical theorem proving is still an active area of AI research. But, as we sh6w in this chapter, the usefulness of some mathematical techniques extends well beyond the traditional scope of mathematics. It turns out that mathematics is no different from any other complex intellectual endeavor in requiring both reliable deductive mechanisms and a mass of heuristic knowledge to control what would otherwise be a completely intractable search problem.

At this point, readers who are unfamiliar with prepositional and predicate logic may want to consult a good introductory logic text before reading the rest of this chapter. Readers who want a more complete and formal presentation of the material

in this chapter should consult Chang and Lee [1973]. Throughout the chapter, we use the following standard logic symbols: "—»" *(material implication},* "-i" *(not),* "V" (or), "A" *(and),* "V" *(for all),* and "3" *(there exists).*

3.1 Representing Simple Facts in Logic

Let's first explore the use of prepositional logic as a way of representing the sort of world knowledge that an AI system might need. Prepositional logic is appealing because it is simple to deal with and a decision procedure for it exists. We can easily

It is raining.
RAINING

It is sunny.

SUNNY

It is windy.
WINDY

If it is raining, then it is not sunny. *RAINING* -> -. *SUNNY*

Figure 3.1: Some Simple Facts in Prepositional Logic

represent real-world facts as logical *propositions* written as *well-formed formulas (wffs)* in prepositional logic, as shown in Figure 3.1. Using these propositions, we could, for example, conclude from the fact that it is raining the fact that it is not sunny. But very quickly we run up against the limitations of prepositional logic. Suppose we want to represent the obvious fact stated by the classical sentence

Socrates is a man. We could write:

SOCRATESMAN But if we also wanted to represent

PIatu is a man. we would have to write something such as:

PLATOMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(SOCRATES) MAN(PLATO)

since now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use predicates applied to arguments. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal. We could represent this as:

MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal. To do that, we really need variables and quantification unless we are willing to write separate statements about the mortality of every known man.

So we appear to be forced to move to first-order predicate logic (or just predicate logic, since we do not discuss higher order theories in this chapter) as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in prepositional logic. In predicate logic, we can represent real-world facts as *statements* written as wff's.

But a major motivation for choosing to use logic at all is that if we use logical statements as a way of representing knowledge, then we have available a good way of reasoning with that knowledge. Determining the validity of a proposition in prepositional logic is straightforward, although it may be computationally hard. So before we adopt predicate logic as a good medium for representing knowledge, we need to ask

whether it also provides a good way of reasoning with the knowledge. At first glance, the answer is yes. It provides a way of deducing new statements from old ones. Unfortunately, however, unlike prepositional logic, it does not possess a decision procedure, even an exponential one. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. In other words, although first-order predicate logic is not decidable, it is semidecidable. A simple such procedure is to use the rules of inference to generate theorems from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought. This method is not particularly efficient, however, and we will want to try to find a better one.

Although negative results, such as the fact that there can exist no decision procedure for predicate logic, generally have little direct effect on a science such as AI, which seeks positive methods for doing things, this particular negative result is helpful since it tells us that in our search for an efficient proof procedure, we should be content if we find one that will prove theorems, even if it is not guaranteed to halt if given a nontheorem. And the fact that there cannot exist a decision procedure that halts on all possible inputs does not mean that there cannot exist one that will halt on almost all the inputs it would see in the process of trying to solve real problems. So despite the theoretical undecidability of predicate logic, it can still serve as a useful way of representing and manipulating some of the kinds of knowledge that an AI system might need.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences

1. Marcus was a man.

2. Marcus was a Pompeian.

3. All Pompeians were Romans.

4. Caesar was a ruler.

3. All Romans were either loyal to Caesar or hated him.

6. Everyone is loyal to someone.

7. People only try to assassinate rulers they are not loyal to.

8. Marcus triedio assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.
man(Marcus

This representation captures the critical fact of Marcus being a man. It fails to, capture some of the information in the English sentence, namely the notion of past i tense. Whether this omission is acceptable or not depends on the use to which we | intend to put the knowledge. For this simple example, it will be all right.

2. Marcus was a Pompeian.
Pompeian (Marcus)

3. All Pompeians were Romans.

VJT : *Pompeian(x) —> Roman(x)*

4. Caesar was a ruler.
ruler(Caesar)

Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name is being referred to in a particular statement may require a fair amount of knowledge and reasoning.

3. All Romans were either loyal to Caesar or hated him.

Roman(x) —» loyalto(x. Caesar) V *hate(x, Caesar)*

In English, the word "or" sometimes means the logical *inclusive-or* and sometimes means the logical *exclusive-or*

(XOR). Here we have used the inclusive interpretation. Some people will argue, however, that this English sentence is really stating an.exclusive-or. To express that, we would have to write:

VJ: : *Roman(x) -> [(loyallo(x, Caesar)* V *hate(x, Caesar))* A *-i(loyallo(x, Caesar)* A *hate(x. Caesar))]*

6. Everyone is loyal to someone.

V.r : **3y** : loyalto(x, y)

A.major problem that arises when trying to convert English sentences into logical statements is the scope of quantifiers. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to

whom he or she is loyal, possibly a different someone for everyone? Or does it say that there exists someone to whom everyone is loyal (which would be written as *3y :* VJT : *loyalto(x,y)Y!* Often only one of the two interpretations seems likely, so people tend to favor it.

7. People only try to assassinate rulers they are not loyal to.

V.< : Vy : *person(x)* A *rulerty)* A *tryassassinate(x, y)* —» *~iloyalto{x,y)*

This sentence, too, is ambiguous. Does it mean that the only rulers that people try to assassinate are those to whom they are not loyal (the interpretation used here), or does it mean that the only thing people try to do is to assassinate rulers to whom they are not loyal?

In representing this sentence the way we did, we have chosen to write "try to assassinate" as a single predicate. This gives a fairly simple representation with which we can reason about trying to assassinate. But using this representation, the connections between trying to

assassinate and trying to do other things and between trying to assassinate and actually assassinating could not be made easily. If such connections were necessary, we would need to choose a different representation.

8. Marcus tried to assassinate Caesar.
tryassassinate(Marcus, Caesar}

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is. For a good description of many issues involved in this process.

Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to Caesar (again ignoring the distinction between past and present tense). Now let's try to produce a formal proof, reasoning backward from the desired goal:

-iloyalto(Marcus, Caesar)

In order to prove" the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can in turn be transformed, and so on, until there are no unsatisfied goals remaining. This process may require the search of an AND-OR graph (as described in Section 3.4) when there are alternative ways of satisfying individual goals. Here, for simplicity, we show only a single path. Figure 3.2 shows an attempt to produce a proof of the goal by reducing the set of necessary but as yet unattained goals to the empty set. The attempt fails, however, since there is no way to satisfy the goal *person(Marcus)* with the statements we have available.

The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. We need to-add the representation of another fact to our system, namely:

$\neg loyalto(Marcus, Caesar)$

↑ (7, substitution)

$person(Marcus) \wedge$
$ruler(Caesar) \wedge$
$tryassassinate(Marcus, Caesar)$

↑ (4)

$person(Marcus)$
$tryassassinate(Marcus, Caesar)$

↑ (8)

$person(Marcus)$

Figure 3.2: An Attempt to Prove *-iloyalto(Marcus, Caesar)*

9. All men are people.

VJC : *man(x) —» person(x)*

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.

From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:

• Many English sentences are ambiguous (for example, 5,6, and 7 above). Choosing the correct interpretation may be difficult.

• There is often a choice of how to represent the knowledge (as discussed in connection with 1, and 7 above). Simple representations are desirable, but they may preclude certain kinds of reasoning. The expedient" representation for a particular set of sentences depends on the use to which the knowledge contained in the sentences will be put.

• Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively, it is usually necessary to have access to another set

of statements that represent facts that people consider too obvious to mention.

An additional problem arises in situations where we do not know in advance which statements to deduce. In the example just presented, the object was to answer the question "Was Marcus loyal to Caesar?" How would a program decide whether it should try to prove

loyalto(Marcus, Caesar) or

-iloyalto{Marcus, Caesar)

There are several -things it could do. It could abandon the strategy we have outlined of reasoning backward from a proposed truth to the axioms and instead try to reason forward and see which answer it gets to. The problem with this approach is that, in general, the branching factor going forward from the axioms is so great that it would probably not get to either answer in any reasonable amount of time. A second thing it could do is use some sort of heuristic rules for deciding which answer is more likely and then try to prove that one first. If it fails to find a proof after some reasonable amount of effort, it can try the other answer. This notion of limited effort is important, since any proof procedure we use may not halt if given a nontheorem. Another thing it could do is simply try to prove both answers simultaneously and stop when one effort is successful. Even here, however, if there is not enough information available to answer the question with certainty, the program may never halt. Yet a fourth strategy is to try both to prove one answer and to disprove it, and to use information gained in one of the processes to guide the other.

3.2 Representing Instance and Isa Relationships

We discussed the specific attributes *instance* and *isa* and described the important role they play in a particularly useful form of reasoning, property inheritance. But if we look back at the way we just represented our knowledge about Marcus and Caesar, we do not appear to have used these attributes at all. We certainly have not used predicates with those names. Why not? The answer is that although we have not used the predicates *instance* and *isa* explicitly, we have captured the

relationships they are used to express, namely class membership and class inclusion.

Figure 3.3 shows the first five sentences of the last section represented in logic in three different ways. The first pan of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as *Roman),* each of which corresponds to a class. Asserting that *P(x)* is true is equivalent to asserting that *x* is an instance (or element) of *P.* The second pan of the figure contains representations that use the *instance* predicate explicitly. The predicate *instance* is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs. But these representations do not use an explicit *isa* predicate. Instead, subclass relationships, such as that between Pompeians and Romans, are described as shown in sentence 3. The implication rule there states that if an object is an instance of the subclass *Pompeian* then it is an instance of the superclass *Roman.* Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship. The third pan contains representations that use both the *instance* and *isa* predicates explicitly. The use of the *isa* predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided. This additional axiom describes how an *instance* relation and an *isa* relation can be combined to derive a new *instance* relation. This one additional axiom is general, though, and does not need to be provided separately for additional *isa* relations.

1. *man(Marcus)*

2. *Pompeian(Marcus)*

3. VJt: *Pompeian(x) —> Roman(x)*

4. *ruler(Caesar)*

3. V.r : *Roman(x) —> loyalto(x, Caesar)* V *hate(x, Caesar)*

1. *instance(Marcus, man)*

2. *instance(Marcus, ompeian)*

3. V.r : *instance(x. Pompeian) —» instancef.x, Roman)*

4. *instance(Caesar, ruler)*

3. VJC *: mstance(x. Roman) —> loyalto(x, Caesar)* V *hale(x, Caesar)*

1. *instance(Marcus, man)*

2. *instance(Marcus, Pompeian)*

3. *isa(Pompeian, Roman)*

4. *instance(Caesar, ruler)*

3. tx: instance(x, Roman) —> loyalto{x, Caesar) V *nate(x, Caesar)*

6. VJT : Vy : Vz : *instance(x, y)* A i'M(>', 2) —> *instance(x, z)*

Figure 3.3: Three Ways of Representing Class Membership

These examples illustrate two points. The first is fairly specific. It is that, although class and superclass memberships are important facts that need to be represented, those memberships need not be represented with predicates labeled *instance* and *isa.* In fact, in a logical framework it is usually unwieldy to do that, and instead unary predicates corresponding to the classes are often used. The second point is more general. There are usually several different ways of representing a given fact within a particular representational framework, be it logic or anything else. The choice depends partly on which deductions need to be supported most efficiently and partly on taste. The only important thing is that within a particular knowledge base consistency of representation is critical. Since any particular inference rule is designed to work on one particular form of representation, it is necessary that all the knowledge to which that rule is intended to apply be in the form that the rule demands. Many errors in

the reasoning performed by knowledge-based programs are the result of inconsistent representation decisions. The moral is simply to be careful.

There is one additional point that needs to be made here on the subject of the use of *isa* hierarchies in logic-based systems. The reason that these hierarchies are so important is not that they permit the inference of superclass membership. It is that by permitting the inference of superclass membership, they permit the inference of other properties associated with membership in that superclass. So, for example, in our sample knowledge base it is important to be able to conclude that Marcus is a Roman because we have some relevant knowledge about Romans, namely that they either hate

Caesar or are loyal to him. we were able to associate knowledge with superclasses that could then be overridden by more specific knowledge associated either with individual instances or with subclasses. In other words, we recorded default values that could be accessed whenever necessary. For example, there was a height associated with adult males and a different height associated with baseball players. Our procedure for manipulating the *isa* hierarchy guaranteed that we always found the correct (i.e., most specific) value for any attribute. Unfortunately, reproducing this result in logic is difficult.

Suppose, for example, that, in addition to the facts we already have, we add the following.'

Pompeian(Paulus} -i *[loyalto(Paulus, Caesar)* V *hate(Paulus, Caesar)]*

In other words, suppose we want to make Paulus an exception to the general rule about Romans and their feelings toward Caesar. Unfortunately, we cannot simply add these facts to our existing knowledge base the way we could just add new nodes into a semantic net. The difficulty is that if the old assertions are left unchanged, then the addition of the new assertions makes the knowledge base inconsistent. In order to restore consistency, it is necessary to modify the original assertion to

which an exception is being made. So our original sentence 5 must become:

VJC : *Roman{x)* A *~'eq(x, Paulus) —> loyalto(x. Caesar)* V *hate(x, Caesar)*

In this framework, every exception to a general rule' must be stated twice, once in a particular statement and once in an exception list that forms part of the general rule. This makes the use of general rules in this framework less convenient and less efficient when there are exceptions than is the use t)f general rules in a semantic net.

A further problem arises when information is incomplete and it is not possible to prove that no exceptions apply in a particular instance.

3.3 Computable Functions and Predicates

In the example we explored in the last section, all the simple facts were expressed as combinations of individual predicates, such as:

tryassassinate(Marcus, Caesar)

This is fine if the number of facts is not very large or if the facts themselves are sufficiently unstructured that there is little alternative. But suppose we want to express simple facts, such as the following greater-than and less-than relationships:

' For convenience, we now return to our original notation using unary predicates to denote class relations.

gt(l,0) /r(0,l) *gt(2,l)*
lt(l,2) *gt(3,2)*
/r(2,3)

Clearly we do not want to have to write out the representation of each of these facts individually. For one thing, there are infinitely many of them. But even if we only consider the finite number of them that can be represented, say, using a single machine word per number, it would be extremely inefficient to

store explicitly a large set of statements when we could, instead, so easily compute each on& as we need it. Thus it becomes useful to augment our representation by these *computable predicates.* Whatever proof procedure we use, when it comes Upon one of these predicates, instead of searching for it explicitly in the database or attempting to deduce it by further reasoning, we can simply invoke a procedure, which we will specify in addition to our regular rules, that will evaluate it and return true or false.

It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of

^(2+3,1)

To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to *gt.*

The next example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects tobe substituted for each other whenever it appears helpful to do so during a proof.

Consider the following set of facts, again involving Marcus:

1. Marcus was a man.

man(Marcus) Again we ignore the issue of tense.

2. Marcus was a Pompeian.

Pompeian(Marcus)

3. Marcus was born in 40 A.D. *horn (Marcus,* 40)

For simplicity, we will not represent A.D. explicitly, just as we normally omit it in everyday discussions. If we ever need to represent dates B.C., then we will have to decide on a way to do that, such as by using negative numbers. Notice that the representation of a sentence does not have to look

like the sentence itself as long as there is a way to convert back and forth between them. This allows us to choose a representation, such as positive and negative numbers, that is easy for a program to work with.

4. All men are mortal.

VJ: : *man(x) —> mortal(x)*

3. All Pompeians died when the volcano erupted in 79 A.D. *erupted(volcano,79)* A *^x : [Pompeian(x) —> died(x,79)]*

This sentence clearly asserts the two facts represented above. It may also assert another that we have not shown, namely that the eruption of the volcano caused the death of the Pompeians. People often assume causality between concurrent events if such causality seems plausible.

Another problem that arises in interpreting this sentence is that of determining the referent of the phrase "the volcano." There is more than one volcano in the world. Clearly the one referred to here is Vesuvius, which is near Pompeii and erupted in 79 A.D. In general, resolving references such as these can require both a lot of reasoning and a lot of additional knowledge.

6. No mortal lives longer than 150 years.

VA- : Vfi : V/2 : *mortal(x)* A *horn(x,t\)* A *gt(t-i* — ?i;150) —» *dead{x, ti)*

There are several ways that the content of this sentence could be expressed. For example, we could introduce a function *age* and assert that its value is never greater than 150. The representation shown above is simpler, though, and it will suffice for this example.

7. It is now 1991.
now= 1991

Here we will exploit the idea of equal quantities that can be substituted for each other.

Now suppose we want to answer the question "Is Marcus alive?" A quick glance through the statements we have suggests that there may be two ways of deducing an answer. Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible. As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking. So we add the following facts:

8. Alive means not dead.

VJC : Vr: *[alive(x, t) —> -^dead(x,t)]* A *[-'dead(x, t) —> alive(x, t)]*

This is not strictly correct, since *-'dead* implies alive only for animate objects. (Chairs can be neither dead nor alive.) Again, we will ignore this for now. This is an example of the fact that rarely do two expressions have truly identical meanings in all circumstances.

9. If someone dies, then he is dead at all later times.

\/ *x :* V/i : Vf2 : *died(x,* f|) A *gt{l^ t\) -> dead(x, tz)*

This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall x : Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(volcano, 79)$
7. $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t : [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

Figure 3.4: A Set of Facts about Marcus

To answer that requires breaking time up into smaller units than years. If we do that, we can then add rules that say such things as "One is dead at *time{yeart, monthi)* if one died during *(yearl, month!)* and *month!* precedes *month]."* We can extend this to days, hours, etc., as necessary. But vye do not want to reduce all time statements to that level of detail, which is unnecessary and often not available.

A summary of all the facts we have now represented is given in Figure 3.4. (The numbering is changed slightly because sentence 5 has been split into two pans.) Now let's attempt to answer the question"Is Marcus alive?" by proving:

-ialive(Marcus, now)

Two such proofs are shown in Figures 3.5 and 3.6. The term *nil* at the end of each proof indicates that the list of conditions remaining to be proved is empty and so the proof has succeeded. Notice in those proofs that whenever a statement of the form:

a A *b* —> *c*

was used, a and *h* were set up as independent subgoals. In one sense they are, but in another sense they are not if they share the same bound variables, since, in that case, consistent

substitutions must be made in each of them. For example, in Figure 3.6 look at the step justified by statement 3. We can satisfy the goal

born(Marcus, t\)

usmg statement.S by binding *t* to 40, but then we must also bind /i to 40 in *gl(naw - t*, 150)

since the two fi's were the same variable in statement 4, from which the two goals came. A good computational proof procedure has to include both a way of determining

$\neg alive(Marcus, now)$

↑ (9, substitution)

$dead(Marcus, now)$

↑ (10, substitution)

$died(Marcus, t_1) \wedge gt(now, t_1)$

↑ (5, substitution)

$Pompeian(Marcus) \wedge gt(now, 79)$

↑ (2)

$gt(now, 79)$

↑ (8, substitute equals)

$gt(1991, 79)$

↑ (compute gt)

nil

Figure 3.5: One Way of Proving That Marcus Is Dead

that a match exists and a way of guaranteeing uniform substitutions throughout a proof. Mechanisms for doing both those things are discussed below.

From looking at the proofs we have just shown, two things should be clear:

• Even very simple conclusions can require many steps to prove.

• A variety of processes, such as matching, substitution, and application *of modus ponens* are involved in the production of a proof. This is true even for the simple statements we are using. It would be worse if we had implications with more than a single term on the right or with complicated expressions involving *ands* and *ors* on the left.

The first of these observations suggests that if we want to be able to do nontrivial reasoning, we are going to need some statements that allow us to take bigger steps along the way. These should represent the facts that people gradually acquire as they become experts. How to get computers to acquire them is a hard problem for which no very good answer is known.

The second observation suggests that actually building a program to do what people do in producing proofs such as these may not be easy. In the next section, we introduce a proof procedure called *resolution* that reduces some of the complexity because it operates on statements that have first been converted to a single canonical form. 1

3.4 Resolution

As we suggest above, it would be useful from a computational point of view ffwe had a proof procedure that carried out in a single operation the variety of processes involved

$\neg alive(Marcus, now)$

↑ (9, substitution)

$dead(Marcus, now)$

↑ (7, substitution)

$mortal(Marcus) \wedge$
$born(Marcus, t_1) \wedge$
$gt(now - t_1, 150)$

↑ (4, substitution)

$man(Marcus) \wedge$
$born(Marcus, t_1) \wedge$
$gt(now - t_1, 150)$

↑ (1)

$born(Marcus, t_1) \wedge$
$gt(now - t_1, 150)$

↑ (3)

$gt(now - 40, 150)$

↑ (8)

$gt(1991 - 40, 150)$

↑ (compute minus)

$gt(1951, 150)$

↑ (compute gt)

nil

Figure 3.6: Another Way of Proving That Marcus Is Dead

in reasoning with statements in predicate logic. Resolution is such a procedure, which gains its efficiency from the fact that it operates on statements that have been convened to a very convenient standard form, which is described below.

Resolution produces proofs by *refutation.* In other words, to prove a statement (i.e., show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable). This approach contrasts with the technique that we have been using to generate proofs by chaining backward from the theorem to be proved to the axioms. Further discussion of how resolution operates will be much more

straightforward after we have discussed the standard form in which statements will be represented, so we defer it until then.

3.4.1 Conversion to Clause Form

Suppose we know that all Romans who know Marcus either hate Caesar or think that anyone who hates anyone is crazy. We could represent that in the following wff:

1x : [Roman(x) A *know(.x, Marcus)} —>*

[hate(x. Caesar) V (V>': 3z : *hate(y, z) —> thinkcrazy(x. y))]*

To use this formula in a proof requires a complex matching process. Then, having matched one piece of it, such as *thinkcrazy(x, y),* it is necessary to do the right thing with the rest of the formula including the pieces in which the matched part is embedded and those in which it is not. If the formula were in a simpler form, this process would be much easier. The formula would be easier to work with if

- It were flatter, i.e., there was less embedding of components.

- The quantifiers were separated from the rest of the formula so that they did not need to be considered.

Conjunctive normal form [Davis and Putnam, 1960] has both of these properties. For example, the formula given above for the feelings of Romans who know Marcus would be represented in conjunctive normal form as

~'Roman(x) V *-^know(x. Marcus)* V

hate(x, Caesar) V *-•hate(y, z)* V *thinkcrazy(x, z)*

Since there exists an algorithm for converting any wff into conjunctive normal form, we lose no generality if we employ a proof procedure (such as resolution) that operates only on wff's in this form. In fact, for resolution to work, we need to go one step further. We need to reduce a set of wff's to a set of *clauses,* where a clause is defined to be a wff in conjunctive normal form but with no instances of the connector A. We can do this by first converting each wff into conjunctive normal

form and then breaking apart each such expression into clauses, one for each conjunct. All the conjuncts will be considered to be conjoined together as the proof procedure operates. To convert a wff into clause form, perform the following sequence of steps.

Algorithm: Convert to Clause Form

1. Eliminate —>, using the fact that *a* —> *b* is equivalent to -ia V *b.* Performing this transformation on the wff given above yields

VJC : *-i[Rffman(x)* A *know(x, Marcus)]* V *[hate(x, Caesar)* V (Vy : -<3z : *hate(y, z))* V *thinkcrazy(x,y))]*

2. Reduce the scope of each -i to a single term, using the fact that -'(-'/?) = *p,* deMorgan's laws [which say that -<a A *b)* = -<*a* V *-'b* and -1(0 V *b)* = -<*a* A *-ih},* and the standard correspondences between quantifiers [-iV.r : *P(x)* = *3x : ~'P(x)* and -'It : *P(x)* = V.T : -'/'(.<•)]. Performing this transformation on the wff from step 1 yields

V.r : *[-iRoman(x)* V *-iknow(x, Marcus)]* V

[hate(x, Caesar) V (Vy : Vz : ^*hate(y, z)* V *thinkcrazy(x. y))]*

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula

V-t: *P{.\)* V V.v : 0(.r) would be converted to |

V.i : *P(x)* V Vv : *Q(y)* This step is in preparation for the next.

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of step 2, we get

V,y : Vy : V; *:* [Roman(x) *V* -know(x, Marcus)] *V* [hate Caesar) ***V*** **(hate(y,:)** ***V*** thinkcrazy(x. **y))]**

At this point, the formula is in what is known as *prenex normal form.* It consists *of & prefix* of quantifiers followed by a *matrix,* which is quantifier-free.

3. Eliminate existential quantifiers. A formula that contains an existentially quantified variable asserts that there is a value that can be substituted for the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

3y : President(y) can be transformed into the formula *President^ I)*

where 51 is a function with no arguments that somehow produces a value that satisfies President.

If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

V-r : **3y : father-of(y. x)**

the value *of y* that satisfies *father-of* depends on the particular value *of x.* Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

\rfx•.fathel•-of(S'2(x),x))

These generated functions are called *Skolem functions.* Sometimes ones with no arguments are called *Skolem constants.*

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use

that any variable it sees is universally quantified. Now the formula prod'iced in step 4 appears as

[^Roman(x) V *-^know(x, Marcus)]* V

[hate(x. Caesar) V *(^hate(y, z)* V *thinkcrazy(x.y))]*

7. Convert the matrix into a conjunction of disjuncts. In the case of our example. since there are no *and's,* it is only necessary to exploit the associative property of *or* [i.e., *a* V *(h* V *c)* =*a* V *b)* V c] and simply remove the parentheses, giving

-<Roman(x) V *-<know(x, Marcus)* V *hate(x. Caesar)* V *-ihate(y,* z) V *thinkcrazy(x, y)*

However, it is also frequently necessary to exploit the distributive property [i.e., *(a* A *h)* V c = *(a* V c) A (ft V c)]. For example, the formula

(winter A *wearingboots)* V *(summer* /\ *wearingsandals)* becomes, after one application of the rule

[winter V *(summer* A *wearingsandals)}*

A *[wearingboots*V *(summer* A *wearingsandals)}*

and then, after a second application, required since there are still conJuncts joined by OR'S,

(winter V *summer)* A

(winter V *wearingsandals)* A

(wearingboots V *summer)* A

(wearingboots V *wearingsandals)*

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the sameset of facts as were represented by the original wff's.

9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

(V.y : *P(x)* A *Q(x))* = V-r : *P(x)* A VA- : *Q(x)*

Thus since each clause is a separate conjunct and since all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it is sometimes necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But, in general, we want to keep clauses in their most general form as long as possible. So when a variable is instantiated, we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals.* These clauses can now be exploited by the resolution procedure to generate proofs.

3.4.2 The Basis of Resolution

The resolution procedure is a simple iterative process: at each step, two clauses, called *the parent clauses,* are compared *(resolved),* yielding a new clause that has been inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

winter V *summer*

-'winter V *cold*

Recall that this means that both clauses must be true (i.e., the clauses, although they look independent, aie really conjoined).

Now we observe that precisely one of *winter* and *-^winter* will be true at any point. If *winter* is true, then *cold* must be true to guarantee the truth of the second clause. If —*winter* is true, then *summer* must be true to guarantee the truth of the first clause. Thus we see that from these two clauses we can deduce

summer V *cold*

This is the deduction that the resolution procedure will make. Resolution operates by taking two clauses that each contain the same literal, in this example, *winter.* The literal must occur in positive form in one clause and in negative form in the other. The *resolvent* is obtained by combining all of the literals of the two parent clauses except the ones that cancel.

If the clause that is produced is the empty clause, then a contradiction has been found. For example, the two clauses

winter

-^winter

will produce the empty clause. If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although as we will see, there are often ways of detecting that no contradiction exists.

So far, we have discussed only resolution in propositionat logic. In predicate logic, the situation is more complicated since we must consider all possible ways of substituting values for the variables. The theoretical basis of the resolution procedure in predicate logic is Herbrand's theorem which tells us the following:

• To show that a set of clauses *S* is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the *Herhrand universe* of *S.*

• A set of clauses *S* is unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of *S* is unsatisfiable.

The second part of the theorem is important if there is to exist any computational procedure for proving unsatisfiability, since in a finite amount of time no procedure will be able to examine an infinite set. The first part suggests that one way to go about finding a contradiction is to try systematically the possible substitutions and see if each produces a contradiction. But that is highly inefficient. The resolution principle, first introduced by Robinson [1965], provides a way of finding contradictions by trying a minimum number of substitutions. The idea is to keep clauses in their general form as long as possible and only introduce specific substitutions when they are required. For more details on different kinds of resolution.

3.4.3 Resolution in Prepositional Logic

In order to make it clear how resolution works, we first present the resolution procedure for prepositional logic. We then expand it to include predicate logic.

In prepositional logic, the procedure for producing a proof by resolution of proposition *P* with respect to a set of axioms *F* is the following.

Algorithm: Prepositional Resolution

1. Convert all the propositions of *F* to clause form.

2. Negate *P* and convert the result to clause form. Add it to the set of clauses obtained in step 1.

3. Repeat until either a contradiction is found or no progress can be made:

   (a) Select two clauses. Call these the parent clauses.

(b) Resolve them together. The resulting clause, called the *resolvent,* will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals *L* and -'/. such that one of the parent clauses contains *L* and the other contains -'L, then select one such pair and eliminate both *L* and -iL from the resolvent.

(c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let's look at a simple example. Suppose we are given the axioms shown in the first column of Figure 3.7 and we want to prove *R.* First we convert the axioms to clause form, as shown in the second column of the figure. Then we negate /?, producing -i/?, which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of producing the empty clause (shown as a box). We might, for example, generate ^he sequence of resolvents shown in Figure 3.8. We begin by resolving with the clause -'/? since that is one of the clauses that must be involved in the contradiction we are trying to find.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and, based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause. To see how this works, let's look again at the example. In order for proposition 2 to be true, one of three things must be true: -if, -i0, or *R.* But we are assuming that -i/? is true. Given

Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Figure 3.7: A Few Facts in Propositional Logic

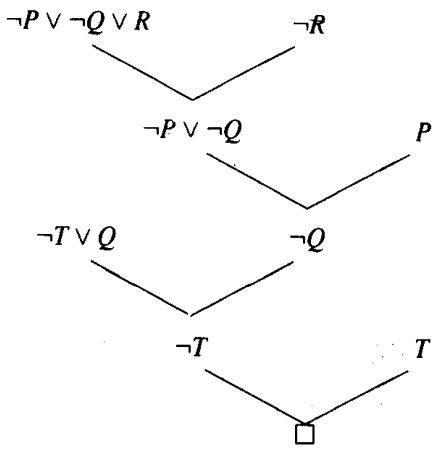


Figure 5-8: Resolution in Propositional Logic

that, the only way for proposition 2 to be true is for one of two things to be true: *-iP* or *-iQ.* That is what the first resolvent clause says. But proposition 1 says that *P* is true, which means that *-iP* cannot be true, which leaves only one way for proposition 2 to be true, namely for -ig to be true (as shown in the second resolvent clause). Proposition 4 can be true if either -if or *Q* is true. But since we now know that *-iQ must* be true, the only way for proposition 4 to be true is for -'7' to be true (the third resolvent). But proposition 5 says that *T* is true. Thus there is no way for all of these clauses to be true in a

single interpretation. This is indicated by the empty clause (the last resolvent).

3.4.4 The Unification Algorithm

In prepositional logic, it is easy to determine that two literals cannot both be true at the same time. Simply look for *L* and -iL. In predicate logic, this matching process is more complicated since the arguments of the predicates must be considered. For example, *man(John)* and -^*man{John)* is a contradiction, while *nwn(John)* and -'*man(Spot)* is not. Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. There is a straightforward recursive procedure, called the *unification algorithfn,* that does just this.

The basic idea of unification is very simple. To attempt to unify two literals, we first check if their initial predicate symbols are the same. If so, we can proceed. Otherwise, there is no way they can be unified, regardless of their arguments. For exaniple, the two literals

tryassassinate(Marcus, Caesar)
hate(Marcus, Caesar)

cannot be unified. If the predicate symbols match, then we must check the arguments, one pair at a time. If the first matches, we can continue with the secbnd, and so on. To test each argument pair, we can simply call the unification procedurd recursively. The matching rules are simple. Different constants or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a predicate expression, with the restriction that the predicate expression must not contain arty instances of the variable being matched.

The only complication in this procedure is that we must find/a single, consistent substitution for the entire literal, not separate ones for each piece of it. To do this, we must take each substitution that we find and apply it to the remainder of the literals before we continue trying to unify them. For example, suppose^ we want to unify the expressions

P(x,x) P(y,z)

The two instances of *P* match fine. Next we compare *x* and *y,* and decide that if we substitute *y* for *x,* they could match. We will write that substitution as

y/x

(We could, of course, have decided instead to substitute JC for *y,* since they are both just dummy variable names. The algorithm will simply pick one of these two substitutions.) But now, if we simply continue and match *x* and z, we produce the substitution *z/x.* But we cannot substitute both *y* and *z SOTX, so* we have not produced a consistent substitution.

What we need to do after finding the first substitution *y/x* is to make that substitution throughout the literals, giving

P(y,y) P(y,z)

Now we can attempt to unify arguments *y* and *z*, which succeeds with the substitution *z/y.* The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. We write the composition as

(z/y)(y/x)

following standard notation for function composition. In general, the substitution (01/02,03/04,.. •)(6i/&2,^3//'4,...)... means to apply all the substitutions of the rightmost list, then lake the result and apply all the ones of the next list, and so forth, until all substitutions have been applied.

The object of the unification procedure is to discover at least one substitution that causes two literals to match. Usually, if there is one such substitution there are many. For example, the literals

hale(x, y)
hate(Marcus, z)

could be unified with any of the following substitutions:

(Marcus/x, z/y)

(Marcus/x, y/z)

(Marcus I x, Caesar I y, Caesar/z)

(Marcus/x, Polonius/y, Polonius/z)

The first two of these are equivalent except for lexical variation. But the second two, although they produce a match, also produce a substitution that is more restrictive than absolutely necessary for the match. Because the final substitution produced by the unification process will be used by the resolution procedure, it is useful to generate the most general unifier possible. The algorithm shown below will do that.

Having explained the operation of the unification algorithm, we can now state it concisely. We describe a procedure Unify(Ll, *LI),* which returns as its value a list representing the composition of the substitutions that were performed during the match. The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(Ll, L2)

1. If LI or L2 are both variables or constants, then:

   (a) If LI and L2 are identical, then return NIL.

   (b) Else if LI is a variable, then if LI occurs in L2 then return {FAIL}, else return (L2/L1).

   (c) Else ifL2 is a variable then ifL2 occurs in LI then return {FAIL}, else return (L1/L2).

   (d) Else return {FAIL}.

2. If the initial predicate symbols in L 1 and L2 are not identical, then return {FAIL}.

3. If LI and L2 have a different number of arguments, then return {FAIL}.

4. Set *SUBST* to NIL. (At the end of this procedure, *SUBST* will contain all the substitutions used to unify LI and L2.)

3. For / <— 1 to number of arguments in / I •

(a) Call Unify with the ;th argument of LI and the /th argument ofL2, putting result in 5'.

(b) If *S* contains FAIL then return {FAIL}.

(c) *IfS* is not equal to NIL then:

i. Apply *S* to the remainder of both *L* 1 and *LI.* ii. *SUBST* := APPENDS, *SUBST).*

***6. Return* SUBST.**

The only part of this algorithm that we have not yet discussed is the check in steps *l(b)* and 1(c) to make sure that an expression involving a given variable is not unified with that variable. Suppose we were attempting to unify the expressions

AX,X) f(8(x),g(x))

If we accepted *g(x)* as a substitution *for x,* then we would have to substitute it for A in the remainder of the expressions. But this leads to infinite recursion since if will never be possible to eliminate *x.*

Unification has deep mathematical roots and is a useful operation in many AI programs, for example, theorem provers and natural language parsers. As a result, efficient data structures and algorithms for unification have been developed. For an introduction to these techniques and applications.

3.4.5 Resolution in Predicate Logic

We now have an easy way of determining that two literals are contradictory—they are if one of them can be unified with the negation of the other. So, for example, *man{x)* and *-'man(Spot)* are contradictory, since *man(x)* and *man(Spot)* can be unified. This corresponds to the intuition that says that *man(x)* cannot be true for all *x* if there is known to be some *x,* say Spot, for which *man(x) is* false. Thus in order to use resolution for

expressions in the predicate logic, we use the unification algorithm to locate pairs of literals that cancel out.

We also need to use the unifier produced by the unification algorithm to generate the resolvent clause. For example, suppose we want to resolve two clauses:

1. *man(Marcus)* *1.*
^*man(x\)* V *mortal(x\)*

The literal *man(Marcus}* can be unified with the literal *man(x\)* with the substitution *Marcus/x\,* telling us that for *x\ = Marcus, ^man(Mcircus)* is false. But we cannot simply cancel out the two *man* literals as we did in prepositional logic and generate the resolvent *mortal(x\).* Clause 2 says that for a given *x*, either *-^man(x\) or mortal(x\).* So for it to be true, we can now conclude only that *mortal(Marcus)* must be true. It is not necessary that *mortal{x\)* be true for all *x\,* since for some values ofA'i, *-'mun(xi)* might be true, making *mortal(x\)* irrelevant to the truth of the complete clause. So the resolvent generated by clauses 1 and 2 must be *mortai(Marcus),* which we get by applying the result of the unification process to the resolvent. The resolution process can

then proceed from there to discover whether *mortal(Marcus)* leads to a contradiction with other available clauses.

This example illustrates the importance of standardizing variables apart during the process of converting expressions to clause form. Given that that standardization has bee'n done, it is easy to determine how the unifier must be used to perform substitutions to create the resolvent. If two instances of the same variable occur, then they must be given identical substitutions.

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements *F* and a statement to be proved *P:*

Algorithm: Resolution

1. Convert all the statements of *F* to clause form.

2. Negate *P* and convert the result to clause form. Add it to the set of clauses obtained in 1.

3. Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended.

(a) Select two clauses. Call these the parent clauses.

(b) Resolve them together. The resolvent will be the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals Tl and -'77 such that one of the parent clauses contains *T*\ and the other contains *Tl* and if 7T and *T2* are unifiable, then neither *T*\ nor *T2* should appear in the resolvent. We call 7T and *T2 Complementary literals.* Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should be omitted from the resolvent.

(c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure,

If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably:

• Only resolve pairs of clauses that contain complementary literals, since only such resolutions produce new clauses that are harder to satisfy than their parents. To facilitate this, index clauses by the predicates they contain, combined with an indication of whether the predicate is negated. Then, given a particular clause, possible resolvents that contain a complementary occurrence of one of its predicates can be located directly.

• Eliminate certain clauses as soon as they are generated so that they cannot participate in later resolutions. Two kinds of clauses should be eliminated: tautologies (which can never be unsatisfied) and clauses that are subsumed

by other clauses (i.e., they are easier to satisfy. For example, *P* V *Q* is subsumed by *P.)*

• Whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with such a clause. This is called the *set-of-support strategy* and corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent.

• Whenever possible, resolve with clauses that have a single literal. Such resolutions generate new clauses with fewer literals than the larger of their parent clauses and thus are probably closer to the goal of a resolvent with zero terms. This method is called the *unit-preference strategy.*

Let's now return to our discussion of Marcus and show how resolution can be used to prove new things about him. Let's first consider the set of statements introduced in Section 3.1. To use them in resolution proofs, we must convert them to clause form as described in Section 3.4.1. Figure 3.9(a) shows the results of that conversion. Figure *3.9(b)* shows a resolution proof of the statement

hate(Marcus, Caesar)

Of course, many more resolvents could have been generated than we have shown, but we used the heuristics described above to guide the search. Notice (hat what we have done here essentially is to reason backward from the statement we want to show is a contradiction through a set of intermediate conclusions to the final conclusion of inconsistency.

Suppose our actual goal in proving the assertion

hate(Marcus, Caesar)

was to answer the question "Did Marcus hate Caesar?" In that case, we might just as easily have attempted to prove the statement

•*ihate(Marcus, Caesar)* To do so, we would have added *hate(Marcus, Caesar)*

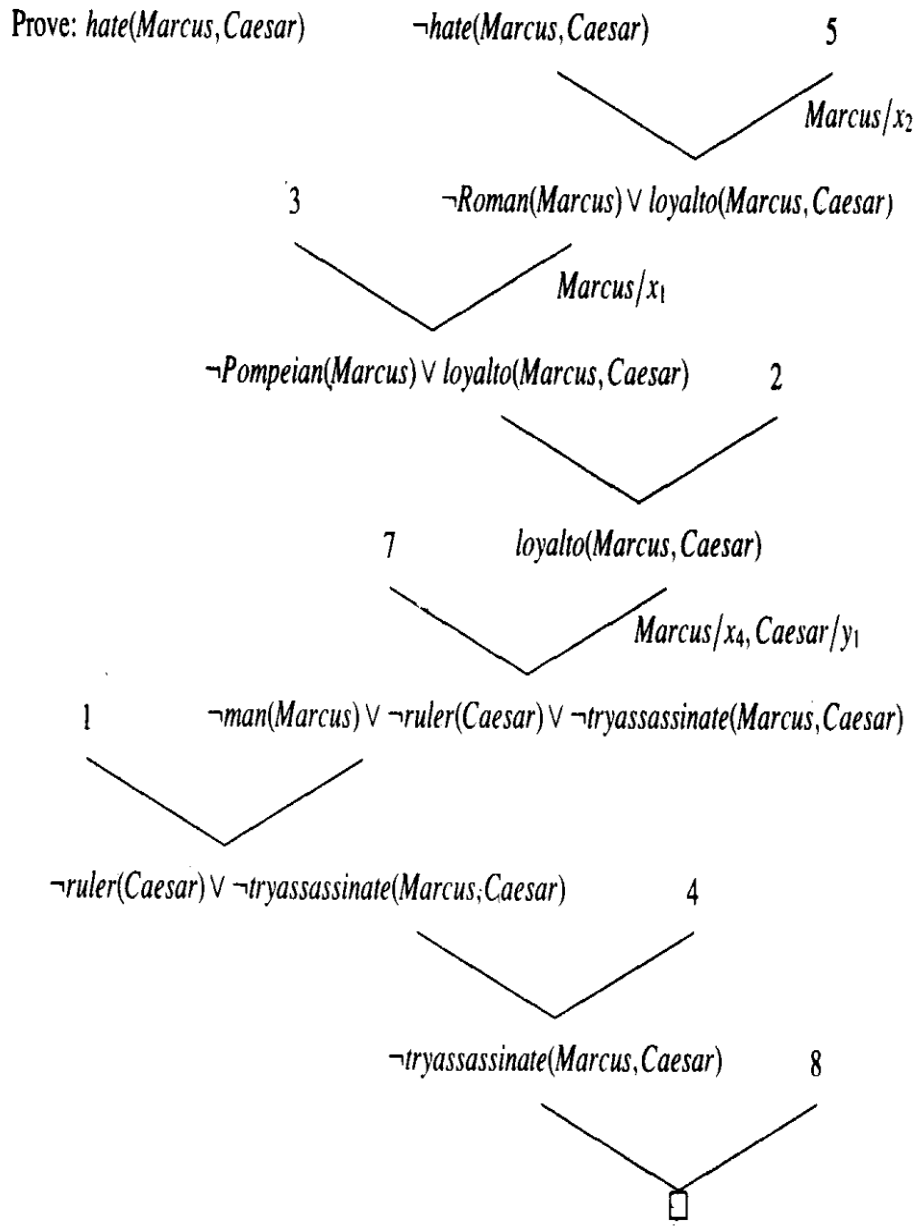
to the set of available clauses and begun the resolution process. But immediately we notice that there are no clauses that contain a literal involving -*'hate.* Since the resolution process can only generate new clauses that are composed of combinations of literals from already existing clauses, we know that no such clause can be generated and thus we conclude that *hate(Marcus, Caesar)* will not produce a contradiction with the known statements. This is an example of the kind of situation in which the resolution procedure can detect that no contradiction exists. Sometimes this situation is detected not at the beginning of a proof, but part way through, as shown in the example in Figure 3.10(a), based on the axioms given in Figure 3.9.

But suppose our knowledge base contained the two additional statements

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\neg Pompeian(x_1) \vee Roman(x_1)$
4. $ruler(Caesar)$
5. $\neg Roman(x_2) \vee loyalto(x_2, Caesar) \vee hate(x_2, Caesar)$
6. $loyalto(x_3, f1(x_3))$
7. $\neg man(x_4) \vee \neg ruler(y_1) \vee \neg tryassassinate(x_4, y_1) \vee loyalto(x_4, y_1)$
8. $tryassassinate(Marcus, Caesar)$

(a)



(b)

Figure 3.9: A Resolution Proof

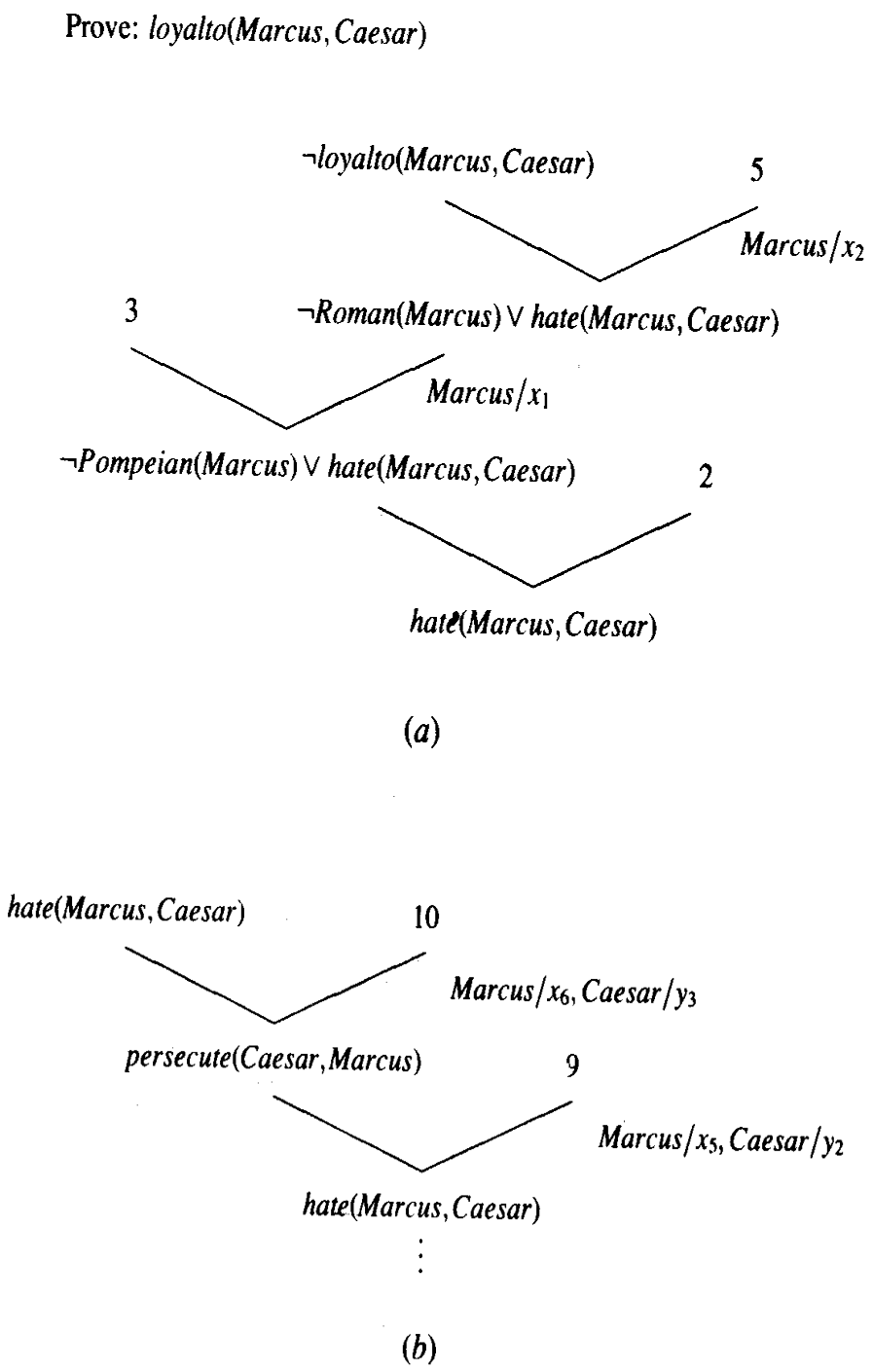


Figure 3.10: An Unsuccessful Attempt at Resolution

9. *persecute(x, y) —> hate(y, x)*

10. *hate(x,y) —> persecute(y,x)*

Converting to clause form, we get

9. *-'per secate(xs*, yz) V *hate(y^, Xs)*

10. *-'hate(X6, yz)* V *persecute^, xs)*

These statements enable the proof of Figure 3.10(a) to continue as shown in Figure 3.10). Now to detect that there is no contradiction we must discover that the only resolvents that can be generated have been generated before. In other words, although we can generate resolvents, we can generate no new ones.

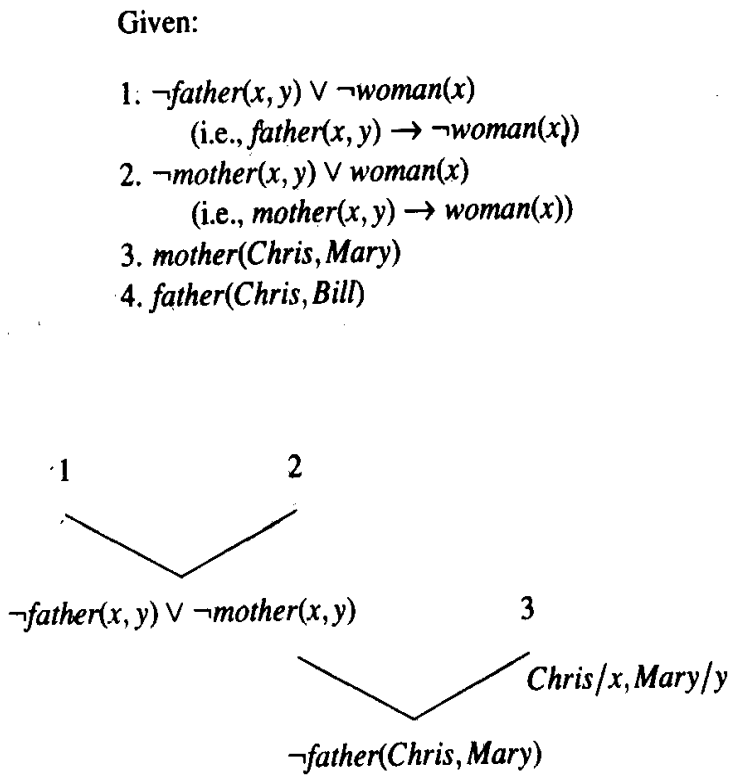


Figure 3.11: The Need to Standardize Variables

Recall that the final step of the process of converting a set of formulas to clause form was to standardize apart the variables that appear in the final clauses. Now that we have discussed the resolution procedure, we can see cleaily why this step is so important. Figure-3.11 shows an example of the difficulty that may arise if standardization is not done. Because the variable y occurs in both clause 1 and clause 2, the substitution at the second resolution step produces a clause that is too restricted and so does not lead to the contradiction that is present in the database. If, instead^e clause

-^father(Chris.y)

had been produced, the contradiction with clause 4 would have emerged. This would have happened if clause 2 had been rewritten-as

-imother^a, h) V *woman(a)*

In its pure form, resolution requires all the knowledge it uses to be represented in the form of clauses. But as we pointed out in Section 3.3, it is often more efficient to represent certain kinds of information in the form of computable functions, computable predicates, and equality relationships. It is not hard to augment-resolution to handle this sort of knowledge. Figure 3.12 shows a resolution proof of the statement

-'alive(Marcus, now)

Axioms in clause form:

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\neg man(x_1) \vee mortal(x_1)$
5. $\neg Pompeian(x_2) \vee died(x_2, 79)$
6. $erupted(volcano, 79)$
7. $\neg mortal(x_3) \vee \neg born(x_3, t_1) \vee \neg gt(t_2 - t_1, 150) \vee dead(x_3, t_2)$
8. $now = 1991$

9a. $\neg alive(x_4, t_3) \vee \neg dead(x_4, t_3)$

9b. $dead(x_5, t_4) \vee alive(x_5, t_4)$

10. $\neg died(x_6, t_5) \vee \neg gt(t_6, t_5) \vee dead(x_6, t_6)$

Prove: $\neg alive(Marcus, now)$

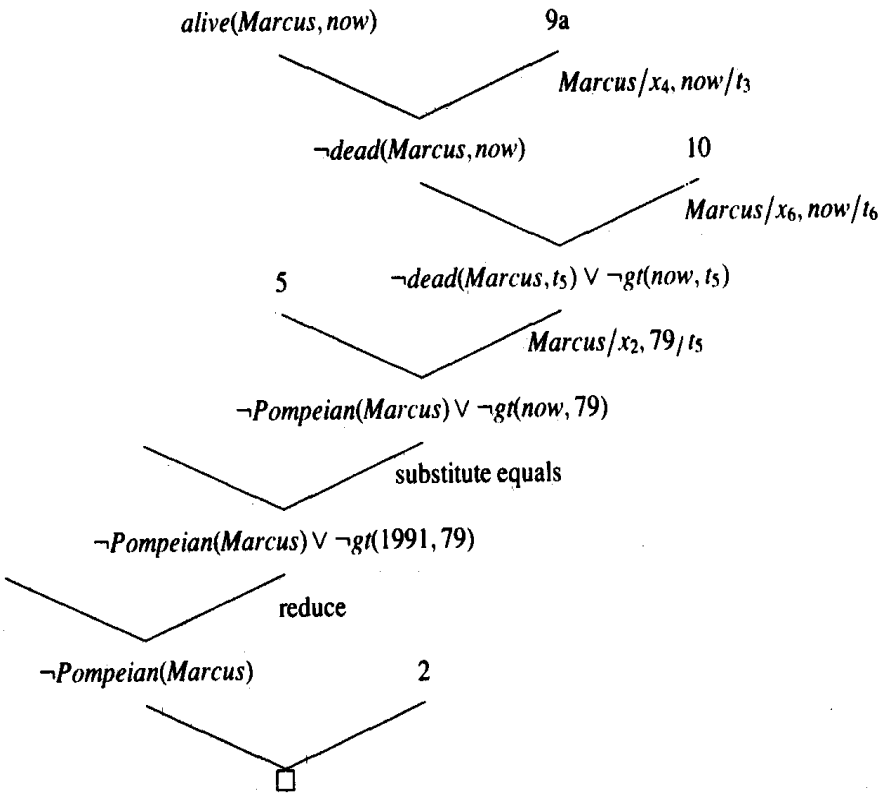


Figure 3.12: Using Resolution with Equality and Reduce

based on the statements given in Section 3.3. We have added two ways of generating new clauses, in addition to the resolution rule:

- Substitution of one value for another to which it is equal.

- Reduction of computable predicates. If the predicate evaluates to FALSE, it can simply be dropped, since adding V FALSE to a disjunction cannot change its truth value. If the predicate evaluates to TRUE, then the generated clause is a tautology and cannot lead to a contradiction.

3.4.6 The Need to Try Several Substitutions

Resolution provides a very good way of finding a refutation proof without actually trying all the substitutions that Herbrand's theorem suggests might be necessary. But it does not always eliminate the necessity of trying more than one substitution. For example, suppose we know, in addition to the statements in Section 3.1, that

hate(Marcus, Paulus)
hate(Marcus, Julian)

Now if we want to prove that Marcus hates some ruler, we would be likely to try each substitution shown in Figure 3.13(a) and *(b)* before finding the contradiction shown in *(c).* Sometimes there is no way short of very good luck to avoid trying several substitutions.

3.4.7 Question Answering

Very early in the history of AI it was realized that theorem-proving techniques could be applied to the problem of answering questions. As we have already suggested;

this seems natural since both deriving theorems from axioms and deriving neyv facts (answers) from old facts employ the process of deduction. We have already shown how resolution can be used to answer yes-no questions, such as "Is Marcus alive?" In this section, we show how resolution can be used to answer fill-in-the-blank questions, such as "When did Marcus die?" or "Who tried to assassinate a ruler?" Answering these questions involves finding a known statement that matches the

terms given in the question and then responding with another piece of that same statement that fills the slot demanded by the question. For example, to answer the question "When did Marcus die?" we need a statement of the form

died(Marcus, *??*)

with ?? actually filled in by some particular year. So, since we can prove the statement *died(Marcus,* 79)

we can respond with the answer 79.

It turns out that the resolution procedure provides an easy way of locating just the statement we need and finding a proof for it. Let's continue with the example question

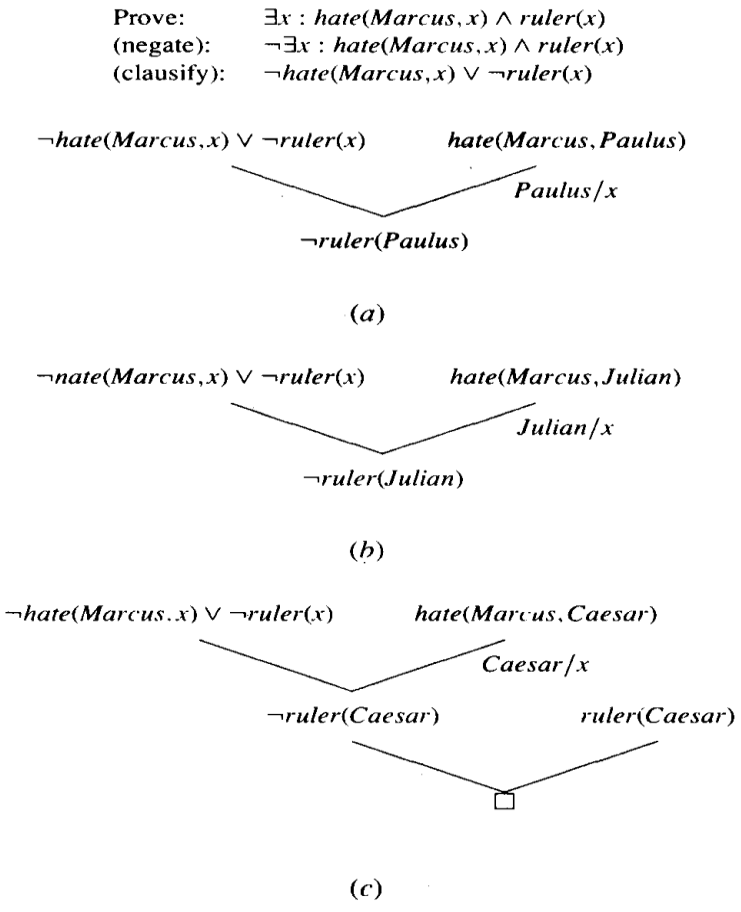


Figure 3.13: Trying Several Substitutions

'When did Marcus die?" In order to be able to answer this question, it must first be true that Marcus died. Thus it must be the case that

*3t**: died(Marcus,** t)*

A reasonable first step then might be to try to prove this. To do so using resolution, we attempt to show that

-i3r**: *died(Marcus,** t)*

produces a contradiction. What does it mean for that statement to produce a contradiction? Either it conflicts with a statement of the form

Vr: *died(Marcus,t)*

where *t* is a variable, in which case we can either answer the question by reporting that there are many times at which Marcus died, or we can simply pick one such time and respond with it. The other possibility is that we produce a contradiction with one or more specific statements of the form

died(Marcus, date)

for some specific value of *date.* Whatever value of date we use in producing that contradiction is the answer we want. The value that proves that there is a value (and thus the inconsistency of the statement that there is no such value) is exactly the value we want.

Figure 3.14(a) shows how the resolution process finds the statement for which we are looking. The answer to the question can then be derived from the chain of unifications that lead back to the starting clause. We can eliminate the necessity for this final step by adding an additional expression to the one we are going to use to try to find a contradiction. This new expression will simply be the one we are trying to prove true (i.e., it will be the negation of the expression that is actually used in the resolution). We can tag it with a special marker so that it will not interfere with the resolution process. (In the figure, it is underlined.) It will just get carried along, but each time unification is done, the variables in this dummy expression will be bound just as are the ones in the clauses

that are actively being used. Instead of terminating on reaching the nil clause, the resolution procedure will terminate when all that is left is the dummy expression. The bindings of its variables at that point provide the answer to the question. Figure 3.14(fc) shows how this process produces an answer to our question.

Unfortunately, given a particular representation of the facts in a system, there will usually be some questions that cannot be answered using this mechanism. For example, suppose that we want to answer the question "What happened in 79 A.D.?" using the statements in Section 3.3. In order to answer the question, we need to prove that something happened in 79. We need to prove

Sr:* event(x, *79)

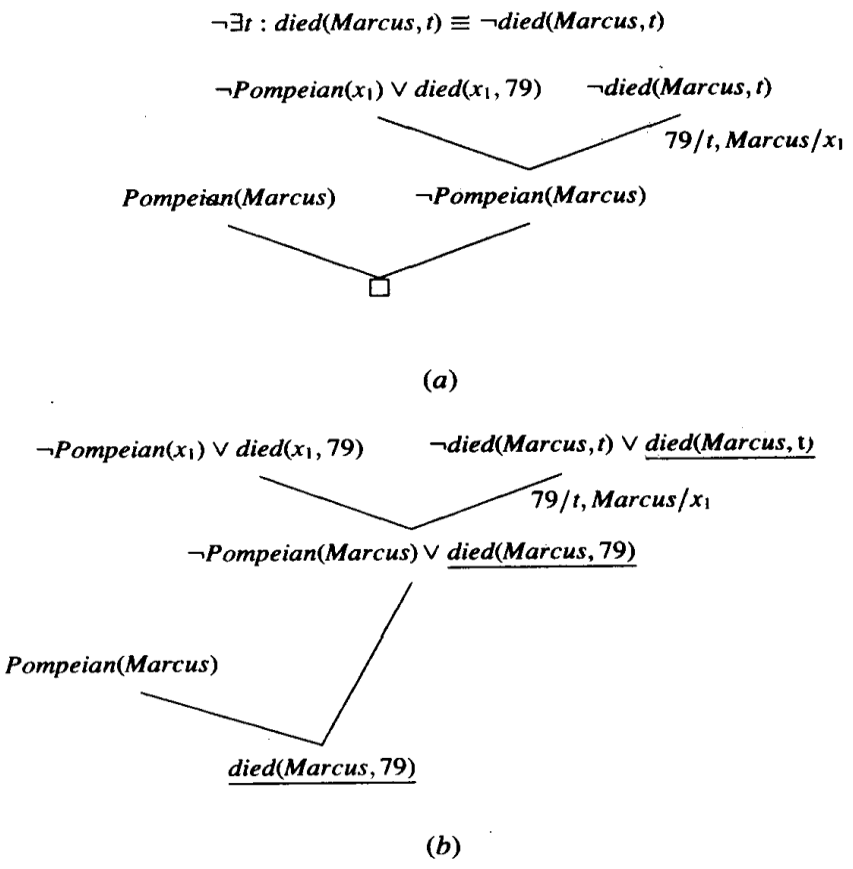


Figure 3.14: Answer Extraction Using Resolution

and to discover a value for *x.* But we do not have any statements of the form *event(x, y).* We can, however, answer the question if we change our representation. Instead of saying

erupted(volcano, 79) we can say

event(erupted(volcano), *79)*

Then the simple proof shown in Figure 3.15 enables us to answer the question.

This new representation has the drawback that it is more complex than the old one. And it still does not make it possible to answer all conceivable questions. In general, it is necessary to decide on the kinds of questions that will be asked and to design a representation appropriate for those questions.

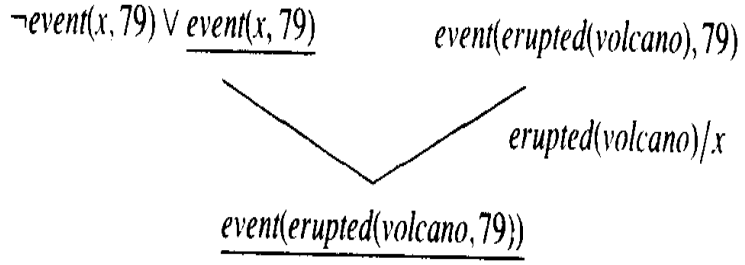


Figure 3.15: Using the New Representation

Of course, yes-no and fill-in-the-blank questions are not the only kinds one could ask. For example, we might ask how to do something. So we have not yet completely solved the problem of question answering. In later chapters, we discuss some other methods for answering a variety of questions. Some of them exploit resolution; others do not.]|

3.5 Natural Deduction

In the last section, we introduced resolution as an easily implementable proof procedure that relies for its simplicity on a uniform representation of the statements it uses. Unfortunately, uniformity has its price—everything looks the same. Since everything looks the same, there is no easy way to select those statements that are the most likely to be useful in solving a particular problem. In converting everything to clause form, we often lose valuable heuristic information that is contained in the original representation of the facts. For example, suppose we believe that all judges who are not crooked are well-educated, which can be represented as

VJC : *judge(x)* A -^£*rooked(x)* —) *educated(x)*

In this form, the statement suggests a way of deducing that someone is educated. But when the same statement is convened to clause form,

-ijudge(x) V *crooked(x)* V *educated(x)*

it appears also to be a way of deducing thai someone is not a judge by showing that he is not crooked and not educated. Of course, in a logical sense, it is. But it is almost certainly not the best way, or even a very good way, to go about showing that someone is not a judge. The heuristic information contained in the original statement has been lost in the transformation.

Another problem with the use of resolution as the basis of a theorem-proving system is that people do not think in resolution. Thus it is very difficult for a person to interact with a resolution theorem prover, either to give it advice or to be given advice by it. Since proving very hard things is something that computers still do poorly, it is important from a practical standpoint that such interaction be possible. To facilitate it, we are forced to look for a way of doing machine theorem proving that corresponds more closely to the

processes used in human theorem proving. We are thus led to what we call, mostly by definition, *natural deduction.*

Natural deduction is not a precise term. Rather it describes a melange of techniques, used in combination to solve problems that are not tractable by any one method alone. One common technique is to arrange knowledge, not by predicates, as we have been doing, but rather by the objects involved in the predicates. Another technique is to use a set of rewrite rules that not only describe logical implications but also suggest the way that those implications can be exploited in proofs.

For a good survey of the variety of techniques that can be exploited in a natural deduction system. Although the emphasis in that paper is on proving mathematical theorems, many of the ideas in it can be applied to a variety of domains in which it is necessary to deduce new statements from known ones. For another discussion of theorem proving using natural mechanisms, which describes a system for reasoning about programs. It places particular emphasis on the use'of mathematical induction as a proof technique. ^

3.6 Summary

In this chapter we showed how predicate logic can be used as the basis of a technique for knowledge representation. We also discussed a problem-solving technique, resolution, that can be applied when knowledge is represented in this way. The resolution procedure is not guaranteed to halt if given a nontheorem to prove. But is it guaranteed to halt and find a contradiction if one exists? This is called the *completeness* question. In the form in which we have presented the algorithm, the answer to this question is no. Some small changes, usually not implemented in theorem-proving systems, must be made to guarantee completeness. But, from a computational point of view, completeness is not the only important question. Instead, we must ask whether a proof can be found in the limited amount of time that is available. There are two ways to approach achieving this computational goal. The first is to search for good heuristics that can inform a theorem-proving program. Current theorem-proving research attempts to do this. The other approach is to change not the program but the data given to the program. In this approach,

we recognize that a knowledge base that is just a list of logical assertions possesses no structure. Suppose an information-bearing structure could be imposed on such a knowledge base. Then that additional information could be used to guide the program that uses the knowledge. Such a program may not look a lot like a theorem prover, although it will still be a knowledge-based problem solver.

A second difficulty with the use of theorem proving in AI systems is that there are some kinds of information that are not easily represented in predicate logic. Consider the following examples:

- "It is very hot today." How can relative degrees of heat be represented?
- "Blond-haired people often have blue eyes." How can the amount of certainty be represented?
- "If there is no evidence to the contrary, assume that any adult you meet knows how to read." How can we represent that one fact should be inferred from the absence of another?
- "It's better to have more pieces on the board than the opponent has." How can we represent this kind of heuristic information?
- "I know Bill thinks the Giants will win, but I think they are going to lose." How can several different belief systems be represented at once?

These examples suggest issues in knowledge representation that we have not yet satisfactorily addressed. They deal primarily with the need to make do with a knowledge base that is incomplete, although other problems also exist, such as the difficulty of representing continuous phenomena in a discrete system. Some solutions to *these* problems are presented in the remaining chapters in this part of the book.

3.7. Model Questions

1. trace the operation of the unification algorithm on each of the following pairs of literals.

   a. f(marcus) and f(caesar)

   b. f(x) and f(g(x))

   c. f(marcus,g(x,y)) and f(x,g(caesar,marcus))

2. consider the following sentences:

   1. john likes all kinds of food

   2. apples are food

   3. chicken is food

   4. anything anyone eats and isn't killed by is food.

   5. bill eats peanuts and is still alive.

   6. sue eats everything bill eats.

   a) Translate these sentences info formulas in predicate logic

   b) Prove that john likes peanuts using backware chaining

   c) Convert the formulas of part a into clause form

   d) Prove that jhon likes peanuts using resolution.

   e) Use resolution to answer the question. "what food does sue cat?"

3. assume the following facts:

a) steve only likes easy courses.

b) science courses are hard.

c) all the courses in the basketweaving department are easy.

d) BK301 is a basketweaving course.

4. suppose that we are attempting to resolve the following clauses.

Loves(father(a),a)

¬loves(y,x) V loves(x,y)

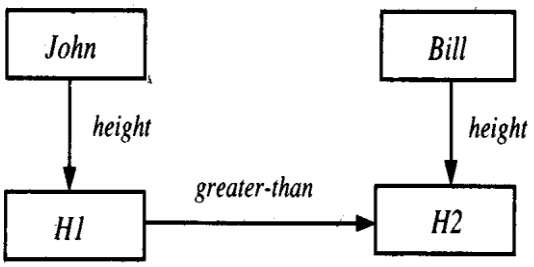
a) What will be the result of the unification algorithm when applied to clause 1 and the first term of clause 2?

b) What must be generated as a result of resolving these two clauses?

c) What does this example show about the order in which the substitutions determined by the unification procedure must be performed?

5. what is wrong with the following argument

a) men are widely distributed over the earth

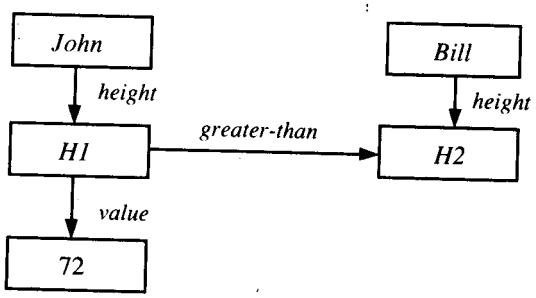


The nodes *HI* and *HI* are new concepts representing John's height and Bill's height, respectively. They are defined by their relationships to the nodes *John* and *Bill.* Using these defined concepts, it is possible to represent such facts as that John's height increased, which we could not do before. (The number 72 increased?)

Sometimes it is useful to introduce the arc *value* to make this distinction clear. Thus we might use the following net to represent the fact that John is 6 feet tall and that he is

^*The* node labeled *BK23* represents the particular book that was referred to by the phrase "the book." Discovering which particular book was meant by that phrase is similar to the problem of deciding on the correct referent for a pronoun, and it can be a very hard problem..

taller than Bill:



The procedures that operate on nets such as this can exploit the fact that some arcs, such as *height,* define new entities, while others, such as *greater-than* and *value,* merely describe relationships among existing entities.

Another example of an important distinction we have missed is the difference between the properties of a node itself and the properties that a node simply holds and passes on to its instances. For example, it is a property of the node *Person* that it is a subclass of the node *Mammal.* But the node *Person* does not have as one of its parts a nose. Instances of the node *Person* do, and we want them to inherit it.

It is difficult to capture these distinctions without assigning more structure to our notions of node, link, and value. In the next section, when we talk about frame systems, we do that. But first, we discuss a network-oriented solution to a simpler problem;

this solution illustrates what can be done in the network model but at what price in complexity.

UNIT – IV

REPRESENTING KNOWLEDGE USING RULES

In this chapter, we discuss the use of rules to encode knowledge. This is a particularly important issue since rule-Dased reasoning systems have played a very important role in the evolution of AI from a purely laboratory science into a commercially significant one.

We have already talked about rules as the basis for a search program. But we gave little consideration to the way knowledge about the world was represented in the rules (although we can see a simple example of this in Section 4.2). In particular, we have been assuming that search control knowledge was maintained completely separately from the rules themselves. We will now relax that assumption and consider a set of rules to represent both knowledge about relationships in the world, as well as knowledge about how to solve problems using the content of the rules.

4.1 Procedural versus Declarative Knowledge

Since our discussion of knowledge representation has concentrated so far on the use of logical assertions, we use logic as a starting point in our discussion of rule-based systems.

In the previous chapter, we viewed logical assertions as declarative representations of knowledge. A *declarative representation* is one in which knowledge is specified, but the use to which that knowledge is to be put is not given. To use a declarative representation, we must augment it with a program that specifies what is to be done to the knowledge and how.

For example, a set of logical assertions can be combined with a resolution theorem prover to give a complete program for solving problems. There is a different way, though, in which logical assertions can be viewed, namely as a *program,* rather than as *data* to a program. In this view, the implication statements define the legitimate reasoning paths and the atomic assertions provide the starting points (or, if we reason backward, the ending points) of those paths. These reasoning paths define the possible execution paths of the program in much the same way that traditional control constructs, such as *if-then-else.* define the execution paths through traditional programs. In other words, we could view logical assertions as procedural representations of knowledge. A *procedural representation* is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself. To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.

Actually, viewing logical assertions as code is not a very radical idea, given that all programs are really data to other programs that interpret (or compile) and execute them. The real difference between the declarative and the procedural views of knowledge lies in where control information resides. For example, consider the knowledge base:

man(Marcus)

man(Caesar)

person(Cleopatra)

VJC : *man(x) —» person(x)*

Now consider trying to extract from this knowledge base the answer to the question *3y: person(y)*

We want to bind *y* to a particular value for which *person* is true. Our knowledge base justifies any of the following answers:

y = Marcus y =
Caesar y =
Cleopatra

Because there is more than one value that satisfies the predicate, but only one value is needed, the answer to the question will depend on the order in which the assertions are examined during the search for a response. If we view the assertions as declarative, then they do not themselves say anything about how they will be examined. If we view them as procedural, then they do. Of course, nondeterministic programs are possible— for example, the concurrent and parallel programming constructs described in Dijkstra [1976], Hoare [1985], and Chandy and Misra [1989]. So, we could view these assertions as a nondeterministic program whose output is simply not defined. If we do this, then we have a "procedural" representation that actually contains no more information than does the "declarative" form. But most systems that view knowledge as procedural do not do this. The reason for this is that, at least if the procedure is to execute on any sequential or on most existing parallel machines, some decision must be made about the order in which the assertions will be examined. There is no hardware support for randomness. So if the interpreter must have a way of deciding, there is no real reason not to specify it as part of the definition of the language and thus to define the meaning of any particular program in the language. For example, we might specify that assertions will be examined in the order in which they appear in the program and that search will proceed depth-first, by which we mean that if a new subgoal is established then it will be pursued immediately and other paths will only be examined if the new one fails. If we do that, then the assertions we gave above describe a program that will answer our question with

}' = *Cleopatra*

To see clearly the difference between declarative and procedural representations, consider the following assertions:

man(Marcus)

man(Caesar)

V.v : *man(x) —> person(x)*

person{CleOpatra)

Viewed declaratively, this is the same knowledge base that we had before. All the same answers are supported by the system and no one of them is explicitly selected. But viewed proceduratly, and using the control model we used to get *Cleopatra* as our answer before, this is a different knowledge base since now the answer to our question is *Marcus.* This happens because the first statement that can achieve the *person* goal is the inference rule *1x : man(x) —> person(x).* This rule sets up a subgoal to find a man. Again the statements are examined from the beginning, and now *Marcus* is found to satisfy the subgoal and thus also the goal. So *Marcus* is reported as the answer.

It is important to keep in mind that although we have said that a procedural representation encodes control information in the knowledge base, it does so only to the extent that the interpreter for the knowledge base recognizes that control information. So we could have gotten a different answer to the *person* question by leaving our original knowledge base intact and changing the interpreter so that it examines statements from last to first (but still pursuing depth-first search). Following this control regime, we report *Caesar* as our answer.

There has been a great deal of controversy in AI over whether declarative or procedural knowledge representation frameworks are better. There is no clearcut answer to the question. As you can see from this discussion, the distinction between the two forms is often very fuzzy. Rather than try to answer the question of which approach is better, what we do in the rest of this chapter is to describe ways in which rule formalisms and interpreters can be combined to solve problems. We begin with a mechanism called *logic programming,* and then we consider more flexible structures for rule-based systems.

4.2 Logic Programming

Logic programming is a programming language paradigm in which logical assertions are viewed as programs, as described in the previous section. There are several logic programming systems in use today, the most popular of which is PROLOG A PROLOG program is described as a series of logical assertions, each *of which is & Horn clause.*[1] A Horn clause is a clause (as

defined in Section 5.4.1) that has at most one positive literal. Thus *p,* -7? V *q,* and *p* —> *q* are all Horn clauses. The last of these does not look like a clause and it appears to have two positive literals. Any logical expression can be converted to clause form. If we do that for this example, the resulting clause is -'*p* V *q,*

' Programs written in pure PROLOG are composed only of Horn clauses. PROLOG, as an actual programming language, however, allows departures from Horn clauses. In the rest of this section, we limit our discussion to pure PROLOG.

$$\forall x : pet(x) \wedge small(x) \rightarrow apartmentpet(x)$$
$$\forall x : cat(x) \vee dog(x) \rightarrow pet(x)$$
$$\forall x : poodle(x) \rightarrow dog(x) \wedge small(x)$$
$$poodle(fluffy)$$

A Representation in Logic

```
apartmentpet(X) :- pet(X), small(X).
pet(X) :- cat(X).
pet(X) :- dog(X).
dog(X) :- poodle(X).
small(X) :- poodle(X).
poodle(fluffy).
```

A Representation in PROLOG

Figure 4.1:A Declarative and a Procedural Representation

which is a well-formed Horn clause. As we will see below, when Horn clauses are written in PROLOG programs, they actually look more like the form we started with (an implication with at most one literal on the right of the implication sign) than the clause form we just produced. Some examples of PROLOG Horn clauses appear below.

The fact that PROLOG programs are composed only of Horn clauses and not of arbitrary logical expressions has two

important consequences. The first is" that because of the uniform representation a simple and efficient interpreter can be written. The second consequence is even more important. The logic of Horn clause systems is decidable (unlike that of full first-order predicate logic).

The control structure that is imposed on a PROLOG program by the PROLOG interpreter is the same one we used at the beginning of this chapter to find the answers *Cleopatra* and *Marcus.* The input to a program is a goal to be proved. Backward reasoning is applied to try to prove the goal given the assertions in the program. The program is read top to bottom, left to right and search is performed depth-first with backtracking.

Figure 4.1 shows an example of a simple knowledge base represented in standard logical notation and then in PROLOG. Both of these representations contain two types of statements, *facts,* which contain only constants (i.e., no variables) and *rules,* which do contain variables. Facts represent statements about specific objects. Rules represent statements about classes of objects.

Notice that there are several superficial, syntactic differences between the logic and the PROLOG representations, including:

1. In logic, variables are explicitly quantified. In PROLOG, quantification is pro-, vided implicitly by the way the variables are interpreted (see below). The distinction between variables and constantsTs made in PROLOG by having all variables

begin with upper case letters and all constants begin with lower case letters or numbers.

2. In logic, there are explicit symbols for *and* (A) and *or* (V). In PROLOG, there is an explicit symbol for *and* (,), but there is none for *or.* Instead, disjunction must be represented as a list of alternative statements, any one of which may provide the basis for a conclusion.

3. In logic, implications of the form"/? implies^" are writtenas/? —» *q*. InPROLOG, the same implication is written "backward," as q : - p. This form is natural in PROLOG because the interpreter always works backwards from a goal, and this form causes every rule to begin with the component that must therefore be matched first. This first component is called the *head* of the rule.

The first two of these differences arise naturally from the fact that PROLOG programs are actually sets of Horn clauses that have been transformed as follows:

1. If the Horn clause contains no negative literals (i.e., it contains a single literal which is positive), then leave it as it is.

2. Otherwise, rewrite the Horn clause as an implication, combining all of the negative literals into the antecedent of the implication and leaving the single positive literal (if there is one) as the consequent.

This procedure causes a clause, which originally consisted of a disjunction of literals (all but one of which were negative), to be transformed into a single implication whose antecedent is a conjunction of (what are now positive) literals. Further, recall that in a clause, all variables are implicitly universally quantified. But, when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent (the head) are still universally quantified. For example, the PROLOG clause

P(x) :- Q(x, y) is **equivalent to**
the logical expression
^x:3y:Q(x,y)-)P(x)

A key difference between logic and the PROLOG representation is that the PROLOG interpreter has a fixed control strategy, and so the assertions in the PROLOG program define a particular search path to an answer to any question. In contrast, the logical assertions define only the set of answers that they justify; they themselves say nothing about how to choose among those answers if there are more than one.

The basic PROLOG control strategy outlined above is simple. Begin with a problem statement, which is viewed as a goal to be proved. Look for assertions that can prove the goal. Consider facts, which prove the goal directly, and also consider any rule whose head matches the goal. To decide whether a fact or a rule can be applied to the current problem, invoke a standard unification procedure. Reason backward from that goal until a path is found that terminates with assertions in the program. Consider paths using a depth-first search strategy and using backtracking. At each choice point, consider options in the order in which they appear in the program. If a goal has more than one conjunctive part, prove the pans in the order in which they appear, propagating variable bindings as they are determined during unification. We can illustrate this strategy with a simple example.

Suppose the problem we are given is to find a value of X that satisfies the predicate apartmentpet (X). We state this goal to PROLOG as

```
?- apartmentpet(X) .
```

Think of this as the input to the program. The PROLOG interpreter begins looking for a fact with the predicate apartmentpet or a rule with that predicate as its head. Usually PROLOG programs are written with the facts containing a given predicate coming before the rules for that predicate so that the facts can be used immediately if they are appropriate and the rules will only be used when the desired fact is not immediately available. In this example, there are no facts with this predicate, though, so the one rule there is must be used. Since the rule will succeed if both of the clauses on its right-hand side can be satisfied, the next thing the interpreter does is to try to prove each of them. They will be tried in the order in which they appear. There are no facts with the predicate pet but again there are rules with it on the right-hand side. But this time there are two such rules, rather than one. All that is necessary for a proof though is that one of them succeed. They will be tried in the order in which they occur. The first will fail because there are no assertions about the predicate cat in the program. The second will eventually lead to success, using the rule about dogs and poodles and using the fact poodle (fluffy). This results in the variable X being bound to fluffy. Now the

second clause smal 1 (X) of the initial rule must be checked. Since X is now bound to fluffy, the more specific goal, small (fluffy), must be proved. This too can be done by reasoning backward to the assertion poodle (fluffy) . The program. then halts with the result apartmentpet (fluffy).

Logical negation (-1) cannot be represented explicitly in pure PROLOG. So, for example, it is not possible to encode directly the logical assertion

V-v : **dos(x) —> -icat(x)**

Instead, negation is represented implicitly by the lack of an assertion. This leads to the problem-solving strategy called *negation as failure* If the PROLOG program of Figure 4.1 were given the goal

?- cat(fluffy).

it would return FALSE because it is unable to prove that Fluffy is a cat. Unfortunately, this program returns the same answer when given the goal

even though the program knows nothing about Mittens and specifically knows nothing that might prevent Mittens from being a cat. Negation by failure requires that we make what is called the *closed world assumption,* which states that ail relevant, true assertions are contained in our knowledge'-base or are derivable from assertions that are so contained. Any assertion that is not present can therefore be assumed to be false. This assumption, while often justified, can cause serious problems when knowledge bases are incomplete.

There is much to say on the topic ot'PROLOG-style versus LISP-style programming. A great advantage of logic programming is that the programmer need only specify rules and facts since a search engine is built directly into the language. The disadvantage is that the search control is fixed. Although it is possible to write PROLOG code that uses search strategies other than depth-first with backtracking, it is difficult to do so. It is even more difficult to apply domain knowledge to constrain a search. PROLOG does allow for rudimentary control of search through a non-logical operator

called *cut.* A cut can be inserted into a rule to specify a point that may not be backtracked over.

More generally, the fact tliat PROLOG programs must be composed of a restricted set of logical operators can be viewed as a limitation of the express! veness of the language. But the other side of the coin is that it is possible to build PROLOG compilers that produce very efficient code.

In the rest of this chapter, we retain the rule-based nature *of* PROLOG, but we relax a number of PROLOG'S design constraints, leading to more flexible rule-based architectures.

4.3 Forward versus Backward Reasoning

The object of a search procedure is to discover a path through a problem space from an initial configuration to a goal state. While PROLOG only searches from a goal state, there are actually two directions in which such a search could proceed:

- Forward, from the start slates
- Backward, from the goal states

The production system model of the search process provides an easy way of viewing forward and backward reasoning as symmetric processes. Consider the problem of solving a particular instance of the 8-puzzle. The rules to be used for solving the puzzle can be written as shown in Figure 4.2. Using those rules we could attempt to solve the puzzle:

- *Reason forward from the initial states.* Begin building a tree of move sequences that might be solutions by starting with the initial configuration(s) at the root of the tree. Generate the next level of the tree by findingall the rules whose *left* sides match the root node and using their right sides to create the new configurations. Generate the next level by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue until a configuration that matches the goal state is generated.

Assume the areas of the tray are numbered:

1	2	3
4	5	6
7	8	9

Square 1 empty *Vnd* Square 2 contains tile *n* *-)*

Square 2. empty and Square 1 contains tile *n* Square 1 empty and Square 4contains tile *n* —>

Square 4 empty and Square 1 contains tile *n* Square 2 empty and Square 1 contains tile *n* —)

Square 1 empty and Square 2 contains tile *n*

Figure 4.2: A Sample of the Rules for Solving the 8-Puzzle

• *Reason backward from the goal states.* Begin building a tree of move sequences that might be solutions by starting with the goal configuration(s) at the root of the tree. Generate the next level of the tree by finding all the rules whose *right* sides match the root node. These are all the rules that, if only we could apply them, would generate the state we want. Use the left sides of the rules to generate the nodes at this second level of the tree. Generate the next level of the tree by taking each node at the previous level and finding all the rules whose right sides match it. Then use the corresponding left sides to generate the new nodes. Continue until a node that matches the initial state is generated. This method of reasoning backward from the desired final state is often called *goal-directed reasoning.*

Notice that the same rules can be *vised* both to reason forward from the initial state and to reason backward from the goal state. To reason forward, the left sides (the preconditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current node and the left sides are used to generate new nodes representing new goal states to be-achieved. This

continues until one of these goal states is matched by an initial state.

In the case of the 8-puzzle, it does not make much difference whether we reason forward or backward; about the same number of paths will be explored in either case. But this is not always true. Depending on the topology of the problem space, it may be significantly more efficient to search in one direction rather than the other.

Four factors influence, the question of whether it is better to reason forward or backward:

• Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.

• In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor..

• Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.

• What kind of event is going to trigger a problem-solving episode? If it 4s the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

A few examples make these issues clearer. It seems easier to drive from an unfamiliar place home than from home to an unfamiliar place. Why is this? The branching factor is roughly the same in both directions (unless one-way streets are laid out very strangely). But for the purpose of finding our way around, there are many more locations that count as being home than there are locations that count as the unfamiliar target place. Any place from which we know how to get home can be considered as equivalent to home. If we can get to any such place, we can get home easily. But in order to find a route

from where we are to an unfamiliar place, we pretty much have to be already attbe unfamiliar place. So in going toward the unfamiliar place, we are aiming at a much smaller target than in going home. This suggests that if our starting position is home and our goal position is the unfamiliar place, we should plan our route by reasoning backward from We unfamiliar place.

On the other hand, consider the problem of symbolic integration. The problem space is the set of formulas, some of which contain integral expressions. The start state is a particular formula containing some integral expression. The desired goal state is a formula that is equivalent to the initial one and that does not contain any integral expressions. So we begin with a single easily identified start state and a huge number of possible goal states. Thus to solve this problem, if is better to reason forward using the rules for integration to try to generate an Sntegral-free, expression than to start with arbitrary integral-free expressions, use the rules for differentiation, and try to generate the particular integral we are trying to solve. Again we want to headloward the largest target; this time that means chaining forward.

These two examples have illustrated the importance of die relative number of start states to goal states in determining the optimal" direction ifl which to search when the branching factor is approximately the same in both directions. When the branching factor is not the same, however, it must also be taken into account.

Consider again the problem of proving theorems in xome particular domain of mathematics. Our goal state is the particular theorem to 1x proved. Our initial states are normally a small set of axioms. Neither of these^s is significantly bigger than the other. But consider the btanching factor in each of the two directions. From a small set of axioms we can derive a very large number of theorems. On the other hand, this large number of theorems must go back to the small set of axioms. So the branching factor is significantly greater going forward from the axioms to the theorems than it is going backward from theorems to axioms. This suggests that it would-be much better to reason backward when trying to prove theorems.

Mathematicians have long realized this as have the designers of theorem-proving programs.

The third factor that determines the direction in which search should proceed is the need to generate coherent justifications of the reasoning process as it proceeds. This is often crucial for the acceptance of programs for the performance of very important tasks. For example, doctors are unwilling to accept the advice of a diagnostic program that cannot explain its reasoning to the doctors' satisfaction. This issue was of concern to the designers of MYCIN , a program that diagnoses infectious diseases. It reasons backward from its goal of determining the cause of a patient's illness. To do that, it uses rules that tell it such things as "If the organism has the following set of characteristics as determined by the lab results, then it is likely that it is organism *x."* By reasoning backward using such rules, the program can answer questions like "Why should I perform that test you just asked for?" with such answers as "Because it would .help to determine whether organism *x* is present."

Most of the search techniques described can be used to search either forward or backward. By describing the search process as the application of a set of production rules, it is easy to describe the specific search algorithms without reference to the direction of the search.

V/e can also search both forward from the start state and backward from the goal simultaneously until two paths meet somewhere in between. This strategy is called *bidirectional search.* It seems appealing if the number of nodes at each step grows exponentially with the number of steps that have been taken. Empirical results suggest that for blind search, this divide-and-conquer strategy is indeed effective. Unfortunately, other results suggest that for informed, heuristic search it is much less likely to be so. Figure 4.3 shows why bidirectional search may be ineffective. The two searches may pass each other, resulting in more work than it would have taken for one of them, on its own, to have finished. However, if individual forward and backward steps are performed as specified by a program that has been carefully constructed to exploit each in

exactly those situations where it can be the most profitable, the results can be more encouraging. In fact, many successful AI applications have been written using a combination of forward and backward reasoning, and most AI programming environments provide explicit support for such hybrid reasoning.

Although in principle the same set of rules can be used for both forward and backward reasoning, in practice it has proved useful to define two classes of rules, each of which

encodes a particular
kind of knowledge. *i*

- Forward rules, which encode knowledge about how to respond to certain input configurations.

- Backward rules, which encode knowledge about how to achieve particular goals.

By separating rules into these two classes, we essentially add to each rule an additional piece of information, namely how it should be used in problem solving. In the next three sections, we describe in more detail the two kinds of rule systems and how they can be combined.

"One exception to this is the means-ends analysis technique, which proceeds not by making successive steps in a single direction but by reducing differences between the current and the goal states, and, as a result, sometimes reasoning backward and sometimes forward.

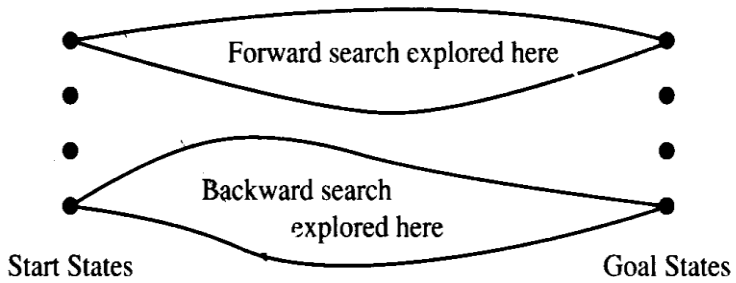


Figure 4.3: A Bad Use of Heuristic Bidirectional Search

4.3.1 Backward-Chaining Rule Systems

Backward-chaining rule systems, of which PROLOG is an example, are good for goal-directed problem solving. For example, a query system would probably use backward chaining to reason about and answer user questions.

In PROLOG, rules are restricted to Horn clauses. This allows for rapid indexing because all of the rules for deducing a given fact share the same rule head. Rules are matched with the unification procedure. Unification tries to find a set of bindings for variables to equate a (sub)goal with the head of some rule. Rules in a PROLOG program are matched in the order in which they appear.

Other backward-chaining systems allow for more complex rules. In MYCIN, for example, rules can be augmented with probabilistic certainty factors to reflect the fact that some rules are more reliable than others.

4.3.2 Forward-Chaining Rule Systems

Instead of being directed by goals, we sometimes want to be directed by incoming data. For example, suppose you sense searing heat near your hand. You are likely to jerk your hand away. While this could be construed as goal-directed behavior, it is modeled more naturally by the recognize-act cycle characteristic of forward-chaining rule systems. In forward-chaining systems, left sides of rules are matched against the. state description. Rules that match dump their right-hand side assertions into the state, and the process repeats.

Matching is typically more complex for forward-chaining systems than backward ones. For example, consider a rule that checks for some condition in the state description and then adds an assertion. After the rule fires, its conditions are probably still valid, so it could fire again immediately. However, we will need some mechanism to prevent repeated firings, especially if the state remains unchanged.

While simple matching and control strategies are possible, most forward-chaining systems implement highly efficient matchers and supply several mechanisms for preferring one

rule over another. We discuss matching in more detail in the next section.

4.3.3 Combining Forward and Backward Reasoning

Sometimes certain aspects of a problem are best handled via forward chaining and other aspects by backward chaining. Consider a forward-chaining medical diagnosis program. It might accept twenty or so facts about a patient's condition, then forward chain on those facts to try to deduce the nature and/or cause of the disease. Now suppose that at some point, the left side of a rule was *nearly* satisfied—say, nine out of ten of its preconditions were met. It might be efficient to apply backward reasoning to satisfy the tenth precondition in a directed manner, rather than wait for forward chaining to supply the fact by accident. Or perhaps the tenth condition requires further medical tests. In that case, backward chaining can be used to query the user.

Whether it is possible to use the same rules for both forward- and backward reasoning also depends on the form of the rules themselves. If both left sides and right sides contain pure assertions, then forward chaining can match assertions on the left side of a rule and add to the state description the assertions on the right side. But if arbitrary procedures are allowed as the right sides of rules, then the rules will not be reversible. Some production languages allow only reversible rules; others do not. When irreversible rules are used, then a commitment to the direction of the search must be made at the time the rules are written. But, as we suggested above, this is often a useful thing to do anyway because it allows the rule writer to add control knowledge to the rules themselves:

4.4 Matching

So far, we have described the process of using search to solve problems as the application of appropriate rules to individual problem states to generate new states to which the rules can then be applied, and so forth, until a solution is found. We have suggested that clever search involves choosing from among the rules that can be applied at a particular point, the

ones that are most likely to lead to a solution. But we have said little about how we extract from the entire collection of rules those that can be applied at a given point. To do so requires some kind *of matching* between the current state and the preconditions of the rules. How should this be done? The answer to this question can be critical to the success of a rule-based system. We discuss a few proposals below.

4.4.1 Indexing

One way tQJselect applicable rules is to do a simple search through all the rules, comparing each one's preconditions to the current state and extracting all the ones that match. But there are two problems with this simple solution:

- In order to solve very interesting problems, it will be necessary to use a large number of rules. Scanning through all of them at every step of the search would be hopelessly inefficient.

- It is not always immediately obvious whether a rule's preconditions are satisfied by a particular state.

Sometimes there are easy ways to deal with the first of these problems. Instead of searching through the rules, use the current state as an index into the rules and select the

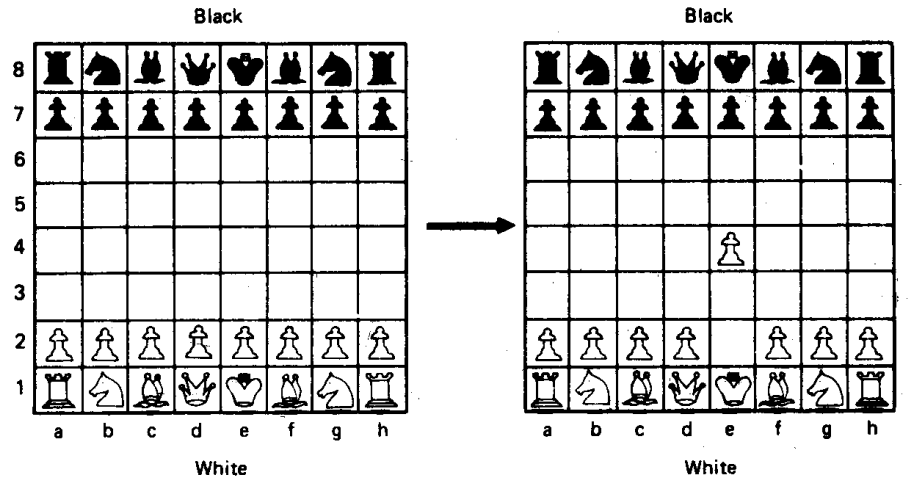


Figure 4.4: One Legal Chess Move

White pawn at
Square(file e, rank 2)
AND
Square(file e, rank 3)
is empty
AND
Square(file e, rank 4)
is empty

→

move pawn from
Square(file e, rank 2)
to Square(file e, rank 4)

Figure 4.5: Another Way to Describe Chess Moves

matching ones immediately. For example, consider the legal-move generation rule for chess shown in in Figure 4.4. To be able to access the appropriate rules immediately, all we need do is assign an index to each board position. This can be done simply by treating the board description as a large number. Any reasonable hashing function can then be used to treat that number as an index into the rules. All the rules that describe a given board position will be stored under the same key and so will be found together. Unfortunately, this simple indexing scheme only works because preconditions of rules match exact board configurations. Thus the matching process* is easy but at the price of complete lack of generality in the statement of the rules. It is often better to write rules in a more general form, such as that shown in Figure 4.5. When this is done, such simple indexing is not possible. In fact, there is often a trade-off between the ease of writing rules (which is increased by the use of high-level descriptions) and the simplicity of the matching process (which is decreased by such descriptions).

All of this does not mean that indexing cannot be helpful even when the preconditions of rules are stated as fairiy high-level predicates. In PROLOG and many theorem-proving systems, for example, rules are indexed by the predicates they contain, so all the rules that could be applicable to proving a particular fact can be accessed fairly quickly.

In the chess example, rules can be indexed by pieces and their positions. Despite some limitations of this approach, indexing

in some form i* very important in the efficient operation of rule-based systems.

4.4.2 Matching with Variables

The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties (of varying complexity) that the situations must have. It often turns out that discovering whether there is a match between a particular situation and the preconditions of a given rule must itself involve a significant search process.

If we want to match a single condition against a single element in a state description, then the unification procedure will suffice. However, in many rule-based systems, we need to compute the whole set of rules that match the current state description. Backward-chaining systems usually use depth-first backtracking to select individual rules, but forward-chaining systems generally employ sophisticated *conflict resolution strategies* to choose among the applicable rules.[3] While it is possible to apply unification repeatedly over the cross product of preconditions and state description elements, it is more efficient to consider the *many-many* match problem, in which many rules are matched against many elements in the state description simultaneously.

One efficient many-many match algorithm is RETE, which gains efficiency from three major sources:

- The temporal nature of data. Rules usually do not alter the state description radically. Instead, a rule will typically add one or two elements, or perhaps delete one or two, but most of the state description remains the same. (Recall our discussion of this as part of our treatment of the frame problem in Section 4.4.) If a rule did not match in the previous cycle, it will most likely fail to apply in the current cycle. RETE maintains a network of rule conditions, and it uses changes in the state description to determine which new rules might apply (and which rules might no longer apply). Full matching is only pursued for candidates that could be affected by incoming or outgoing data.

• Structural similarity in rules. Different rules may share a large number of preconditions. For example, consider rules for identifying wild animals. One rule concludes *jaguar(x)* if *mammal(x), feline{x), carnivorous{x),* and *has-spots(x).* Another rule concludes *tiger(x)* and is identical to the first rule except that it replaces *has-spots* with *has-stripes.* If we match the two rules independently, we will repeat a lot of work unnecessarily. RETE stores the rules so that they share structures in memory; sets of conditions that appear in several rules are matched (at most) once per cycle.

• Persistence of variable binding consistency. While all the individual preconditions of a rule might be met, there may be variable binding conflicts that prevent the rule from firing. For example, suppose we know the facts *son(Mary, Joe)* and *son(Bill, Bob).* The individual preconditions of the rule

•Conflict resolution is discussed in the next section.

son(x,y) A *son(y, z) —> grandparent(x, z)*

can be matched, but not in a manner that-satisfies the constraint imposed by the variable *y.* Fortunately, it is not necessary to compute binding consistency from scratch every time a new condition is satisfied. RETE remembers its previous calculations and is able to merge new binding information efficiently.

Other matching algorithms take different stands on how much time to spend on saving state information between cycles. They can be more or less efficient than RETE, depending on the types of rules written for the. domain and on the degree of hardware parallelism available.

4.4.3 Complex and Approximate Matching

A more complex matching process is required when the preconditions or a rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.

An even more complex matching process is required if rules should be applied if their preconditions *approximately* match the current situation. This is often the case in situations involving physical descriptions of the world. For example, a speech-understanding program must contain rules that map from a description of a physical waveform to phones (instances of English phonemes, such as *p* or *d).* There is so much variability in the physical signal, as a result of background noise, differences in the way individuals speak, and so forth, that one can hope to find only an approximate match between the rule that describes an ideal sound and the input that describes an unideal world. Approximate matching is particularly difficult to deal with because as we increase the tolerance allowed in the match, we also increase the number of rules that will match, thus increasing the size of the main search process. But approximate matching is nevertheless superior to exact matching in situations such as speech understanding, where exact matching may often result in no rules being matched and the search process coming to a grinding halt. Although symbolic techniques for approximate matching exist, there is another, very different approach that can be used to solve this problem. We describe connecUonist systems (also called neural nets).

For some problems, almost all the action is in the matching of the rules to the problem state. Once that is done, so few rules apply that the remaining search is trivial. This was the case, for example, in ELIZA an early AI program that simulated the behavior .of a Rogerian therapist. A fragment of a dialogue between ELIZA and a user is shown in Figure 4.4. ELIZA's knowledge about both English and psychology was coded in a set of simple rules. Figure 4.7 shows some ELIZA-like rules.

ELIZA operated by matching the left sides of the rules against the user's last sentence and using the appropriate right side to generate a response. For example, if the user typed "My brother is mean to me," ELIZA might respond, "Who else in your family is mean to you?" or "Tell me more about your family." The rules were indexed by keywords so only a few had actually to be matched against a canicular sentence. Some

Person: Men are all alike.
ELIZA: In what way?
Person: They're always bugging us about something or other.
ELIZA: Can you think of a specific example?
Person: Well, my boyfriend made me come here.
ELIZA: Your boyfriend made you come here.
Person: He says I'm depressed much of the time
ELIZA: I am sorry to hear you are depressed.
Person: It's true. I am unhappy.
ELIZA: Do you think coming here will help you not to be unhappy?
Person: I need some help, that much seems certain.
ELIZA: What would it mean to you if you got some help?
Person: Perhaps I could learn to get along with my mother.
ELIZA: Tell me more about your family.

Figure 4.6: A Bit of a Dialogue with ELIZA

(X me Y)	→	(X you Y)
(I remember X)	→	(Why do remember X just now?)
(My {family-member} is Y)	→	(Who else in your family is Y?)
(X {family-member} Y)	→	(Tell me more about your family)

Figure 4.7: Some ELIZA-like rules

of the rules had no left side, so the rule could apply anywhere. These rules were used if no other rules matched and they generated replies such as "Tell me more about that." Notice that the rules themselves cause a form of approximate matching to occur. The patterns ask about specific words in the user's sentence. They do not need to match entire sentences. Thus a great variety of sentences can be matched by a single rule, and the grammatical complexity of English is pretty much ignored. This accounts both for ELIZA's major strength, its ability to say something fairly reasonable almost all of the time, and its major weakness, the superficiality, of its understanding and its ability to be led completely astray. Approximate matching can easily lead to both these results.

As if the matching process were not already complicated enough, recall the frame problem mentioned. One way of dealing with the frame problem is to avoid storing entire state descriptions at each node but instead to store only the changes from the previous node. If this is done, the matching process will have to be modified to scan backward from a node through its predecessors, looking for the required objects.

4.4.4 Conflict Resolution

The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable bindings were generated by the matching process. It is the job of the search method to decide on the order in which rules will be applied. But sometimes it is useful to incorporate some of that decision making into the matching process. This phase of the matching process is then called *conflict resolution.*

There are three basic approaches to the problem of conflict resolution in a production system:

- Assign a preference based on the rule that matched.
- Assign a preference based on the objects that matched.
- Assign a preference based on the action that the matched rule would perform.

Preferences Based on Rules

There are two common ways of assigning a preference based on the rules themselves. The first, and simplest, is to consider the rules to have been specified in a particular order, such as the physical order in which they are presented to the system. Then priority is given to the rules in the order in which they appear. This is the scheme used in PROLOG.

The other common rule-directed preference scheme is to give priority to special case rules over rules that are more general. In the case of the water jug problem, recall that rules 11 and 12 were special cases of rules 9 and 5, respectively. The

purpose of such specific rules is to allow for the kind of knowledge that expert problem solvers use when they solve problems directly, without search. If we consider all rules that match, then the addition of such special-purpose rules will increase the size of the search rather than decrease it. In order to prevent that, we build the matcher so that it rejects rules that are more general than other rules that also match. How can the matcher decide that one rule is more general than another'? There are a few easy ways:

- If the set of preconditions of one rule contains all the preconditions of another (plus some others), then the second rule is more general than the first.

- 'If the preconditions of one rule are the same as those of another except that in the first case variables are specified where in the second there are constants, then the first rule is more general than the second.

Preferences Based on Objects

Another way in which the matching process can ease the burden on the search mechanism is to order the matches it finds based on the importance of the objects that are matched. There are a variety of ways this can happen. Consider again ELIZA, which matched patterns against a user's sentence in order to find a rule to generate a reply. The patterns looked for specific combinations of important keywords. Often an input sentence contained several of the keywords that ELIZA knew. If that happened, then ELIZA made use of the fact that some keywords had been marked as being more significant than others. The pattern matcher returned the match involving the highest priority keyword. For example, ELIZA knew the word "I" as a keyword. Matching the input sentence "I know everybody laughed at me" by the keyword "I" would have enabled it to respond, "You say you know everybody laughed at you." But ELIZA also knew the word "everybody" as a keyword. Because "everybody" occurs more rarely than "I," ELIZA knows it to be more semantically significant and thus to be the clue to which it should respond. So it will produce a response such as "Who in particular are you thinking of?" Notice that priority matching such as this is particularly important if only one of the choices will ever be tried. This was true for ELIZA and

would also be true, say, for a person who, when leaving a fast-burning room, must choose between turning off the lights (normally a good thing to do) and grabbing the baby (a more important thing to do).

Another form of priority matching can occur as a function of the position of the matchable objects in the current state description. For example, suppose we want to model the behavior of human short-term memory (STM). Rules can be matched against the current contents of STM and then used to generate actions, such as producing output to the environment or storing something in long-term memory. In this situation, we might like to have the matcher first try to match against the objects that have most recently entered STM and only compare against older elements if the newer elements do not trigger a match. For a discussion of this method as a conflict resolution strategy in a production system.

Preferences Based on States

Suppose that here are several rules waiting to fire. One way of selecting among them is to fire all of them temporarily and to examine the results of each. Then, using a heuristic function that can evaluate each of the resulting states, compare the merits of the results, and select the preferred one. Throw away (or maybe keep for later if necessary) the remaining ones.

This approach should look familiar—it is identical to the best-first search procedure. Although conceptually this approach can be thought of as a conflict resolution strategy, it is usually implemented as a search control technique that operates on top of the states generated by rule applications. The drawback to this design is that 'LISP-coded search control knowledge is procedural and therefore difficult to modify. Many AI search programs, especially ones that leam from their experience, represent their control strategies declaratively. The next section describes some methods for capturing knowledge about control using rules.

4.5 Control Knowledge

A major theme of this book is that while intelligent programs require search, search is computationally intractable unless it

is constrained by knowledge about the world. In large knowledge bases that contain thousands of rules, the intractability of search is an overriding concern. When there are many possible paths of reasoning, it is critical that

```
Under conditions A and B,
Rules that do {not} mention X
      {at all,
       in their left-hand side,
       in their right-hand side}
will
      {definitely be useless,
       probably be useless
         ...
       probably be especially useful
       definitely be especially useful}
```

Figure 4.8: Syntax for a Control Rule

fruitless ones not be pursued. Knowledge about which paths are most likely to lead quickly to a goal state is often called *search control knowledge.* It can take many forms:

1. Knowledge about which states are more preferable to others.

2. Knowledge about which rule to apply in a given situation.

3. Knowledge about the order in which to pursue subgoals.

4. Knowledge about useful sequences of rules to apply.

The first type of knowledge could be represented with heuristic evaluation functions. There are many ways of representing the other types of control knowledge. For example, rules can be labeled and partitioned. A medical diagnosis system might have one set of rules for reasoning about bacteriological diseases and another set for immunological diseases. If the system is trying to prove a particular fact by backward chaining, it can probably eliminate one of the two rule sets, depending on what the fact is. Another method is to assign cost and probability-of-success measures to rules. The

problem solver can then use probabilistic decision analysis to choose a cost-effective alternative at each point in the search.

By now it should be clear that we are discussing how to represent knowledge about knowledge. For this reason, search control knowledge is sometimes called *metaknowledge.* Davis [1980] first pointed out the need for meta-knowledge, and" suggested that it be represented declaratively using rules. The syntax for one type of control rule is shown in Figure 4.8.

A number of AI systems represent their control knowledge with rules. We look briefly at two such systems, SOAR and PRODIGY.

SOAR [Laird *et al.,* 1987] is a general architecture for building intelligent systems, SOAR is based on a set of specific, cognitively motivated hypotheses about the structure of human problem solving. These hypotheses are derived from what we know about short-term memory, practice effects, etc. In SOAR:

1. Long-term memory is stored as a set of productions (or, rules).

2. Short-term memory (also called *working memory)* is a buffer that is affected by perceptions and' serves as a storage area for facts deduced by rules in long-term memory. Working memory is analogous to the state description in problem solving.

3. All problem-solving activity takes place as state space traversal. There are several classes of problem-solving activities, including reasoning about which states to explore, which rules to apply in a given situation, and what effects those rules will have.

4. All intermediate and final results of problem solving are remembered (or, *chunked)* for future reference.

The third feature is of most interest to us here. When SOAR is given a start state and a goal state, Tt sets up an initial problem space. In order to take the first step in that space, it must choose a rule from the set of applicable ones. Instead of employing a fixed conflict resolution strategy, SOAR considers that choice of rules to be a substantial problem in its own

right, and it actually sets up another, auxiliary problem space. The rules that apply in this space look something like the rule shown in Figure 4.8. Operator preference rules may be very general, such as the ones described in the previous section on conflict resolution, or they may contain domain-specific knowledge.

SOAR also has rules for expressing a preference for applying a whole sequence of rules in a given situation. In learning mode, SOAR can take useful sequences and build frorti them more complex productions that it can apply in the future.

We can also write rules based on preferences for some states over others. Such rules can be used to implement the basic search strategies. For example, if we always prefer to work from the state we generated last, we will get depth-first behavior. On the other hand, if we prefer states that were generated earlier in time, we will get breadth-first behavior. If we prefer any state that looks better than the current state (according to some heuristic function), we will get hill climbing. Best-first search results when state preference rules prefer the state with the highest heuristic score. Thus we see that all of the weak methods are subsumed by an architecture that reasons with explicit search control knowledge. Different methods may be employed for different problems, and specific domain knowledge can override the more general strategies.

PRODIGY [Minton *et* a/., 1989] is a general-purpose problem-solving system that incorporates several different learning mechanisms. A good deal of the learning in PRODIGY is directed at automatically constructing a set of control rules to improve search in a particular domain. We return to PRODIGY'S learning methods, but we mention here a few facts that bear on the issue of search control rules. PRODIGY can acquire control rules in.a number of ways:

- Through hand coding by programmers.
- Through a static analysis of the domain's operators.
- Through looking at traces of its own problem-solving behavior.

PRODIGY leams control rules from its experience, but unlike SOAR it also leams from its failures. If PRODIGY pursues an unfruitful path, it will try to come up with an explanation of why that path failed. It will. then use that explanation to build control knowledge that will help it avoid fruitless search paths in the future.

One reason why a path may lead to difficulties is that subgoals can interact with one another. In the process of solving one subgoal, we may undo our solution of a previous subgoal. Search control knowledge can tell us something about the order in which we should pursue our subgoals. Suppose we are faced with the problem of building a piece of wooden furniture. The problem specifies that the wood must be sanded, sealed, and painted. Which of the three goals do we pursue first? To humans who have knowledge about this sort of thing, the answer is clear. An AI program, however, might decide to try painting first, since any physical object can be painted, regardless of whether it has been sanded. However, as the program plans further, it will realize that one of the effects of the sanding process is to remove the paint. The program will then be forced to plan a repainting step or else backtrack and try working on another subgoal first. Proper search control knowledge can prevent this wasted computational effort. Rules we might consider include:

- If a problem's subgoals include sanding and painting, then we should solve the sanding subgoal first.
- If subgoals include sealing and painting, then consider what the object is made of. If the object is made of wood, then we should seal it before painting it.

Before closing this section, we should touch on a couple of seemingly paradoxical issues concerning control rules. The first issue is called the *utility problem* .As we add more and

more control knowledge to a system, the system is able to search more judiciously. This cuts down on the number of nodes it expands. However, in deliberating about which step to take next in the search space, the system musi consider all the control rules. If there are many control rules, simply matching them all can be very time-consuming. It is easy to reach a situation (especially in systems that generate control knowledge automatically) in which the system's problem-solving efficiency, as measured in CPU cycles, is worse with the control rules than without them. Different systems handle this problem in different ways.

The second issue concerns the complexity of the production system interpreter. As this chapter has progressed, we have seen a trend toward explicitly representing more and more knowledge about how search should proceed. We have found it useful to create meta-rules that talk about when to apply other rules. Now, a production system interpreter must know how to apply various rules and meta-rules, so we should expect that our interpreters will have to become more complex as we progress away from simple backward-chaining systems like PROLOG. And yet, moving to a declarative representation for control knowledge means that previously hand coded LISP functions can be eliminated from the interpreter. In this sense, the interpreter becomes more streamlined.

4.6 Summary

In this chapter, we have seen how to represent knowledge declaratively in rule-based systems and how to reason with that knowledge. We began with a simple mechanism, logic programming, and progressed to more complex production system models that can reason both forward and backward, apply sophisticated and efficient matching techniques, and represent their search control knowledge in rules.

In later chapters, we expand further on rule-based systems. The use of rules that allow default reasoning to occur in the absence of specific counter evidence. The idea of attaching probabilistic measures to rules. The rule-based systems are being used to solve complex, real-world problems.

The book *Pattern-Directed Inference Systems* is a collection of papers describing the wide variety of uses to which production systems have been put in AI. Its introduction provides a good overview of the subject as a introduction to programming in production rules, with an emphasis on the OPS5 programming language.

4.7 Exercises

1. Consider the following knowledge base:

VJC : Vy : **cat(x) /\fish(y) —> likes —to— eat(x,y)**

1x : calico(x) —> cat(x)

1x: tuna(x) —>fish(x)

tuna(Charlie)

tuna(Herb)

calico(Puss)

(a) Convert these wff's into Horn clauses.

(b) Convert *he Horn clauses into a PROLOG program.

(c) Write a PROLOG query corresponding to the question, "What does Puss like to eat?" and show how it will be answered by your program.

(d) Write another PROLOG program that corresponds to the same set of wff's but returns a different answer to the same query.

2. A problem-solving search can proceed either forward (from a known start state to a desired goal state) or backward (from a goal state to a start state). What factors determine the choice of direction for a particular problem?

3. If a problem-solving search program were to be written to solve each of the following types of problems, determine whether the search should proceed forward or backward:

(a) water jug problem

(b) blocks world

(c) natural language understanding

4. Program the interpreter for a production system. You will need to build a table that holds the rules and a matcher that compares the current state to the left sides of the rules. You will also need to provide an appropriate control strategy to select among competing rules. Use your interpreter as the basis of a program that solves water jug problems.

UNIT - V

WEAK SLOT-AND-FILLER STRUCTURES, STRONG SLOT-AND-FILLER STRUCTURES

In this chapter, we continue the discussion of slot-and-filler structures. Recall that we originally introduced them as a device to support property 'inheritance along *isa* and *instance* links. This is an important aspect of these structures. Monotonic inheritance can be performed substantially more efficiently with such structures than with pure logic, and nonmonotonic inheritance is easily supported. The reason that inheritance is easy is that the knowledge in slot-and-filler systems is structured as a set of entities and their attributes. This structure turns out to be a useful one for other reasons besides the support of inheritance, though, including:

- It indexes assertions by the entities they describe. More formally, it indexes binary predicates [such as *team(Three-Finger-Brown, Chicago-Cuhs)] by* their first argument. As a result, retrieving the value for an attribute of an entity is fast.

- It makes it easy to describe properties of relations. To do this in a purely logical system requires some higher-order mechanisms.

- It is a form of object-oriented programming and has the advantages that such systems normally have, including modularity and ease of viewing by people.

We describe two views of this kind of structure: semantic nets and frames. We talk about the representations themselves and about techniques for reasoning with them. We do not say much, though, about the specific knowledge that the structures should contain. We call these "knowledge-poor" structures "weak," by analogy with the weak methods for

problem solving that we discussed in Chapter 3. In the next chapter, we expand this discussion to include "strong" slot-and-filler structures, in which specific commitments to the content of the representation are made.

The slot-and-filler structures described in the previous chapter are very general, Indi vidual semantic networks and frame systems may have specialized links and inference procedures, but there are no hard and fast rules about what kinds of objects and links are good in general for knowledge representation. Such decisions are left up to the builder of the semantic network or frame system.

The three structures discussed in this chapter, *conceptual dependency, scripts,* and *CYC,* on the other hand, embody specific notions of what types of objects and relations are permitted. They stand for powerful theories of how AI programs can represent and use knowledge about common situations.

5.1. Semantic Nets

The main idea behind semantic nets is that the meaning of a concept comes,, from the ways in which it is connected to other concepts. In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled arcs, which

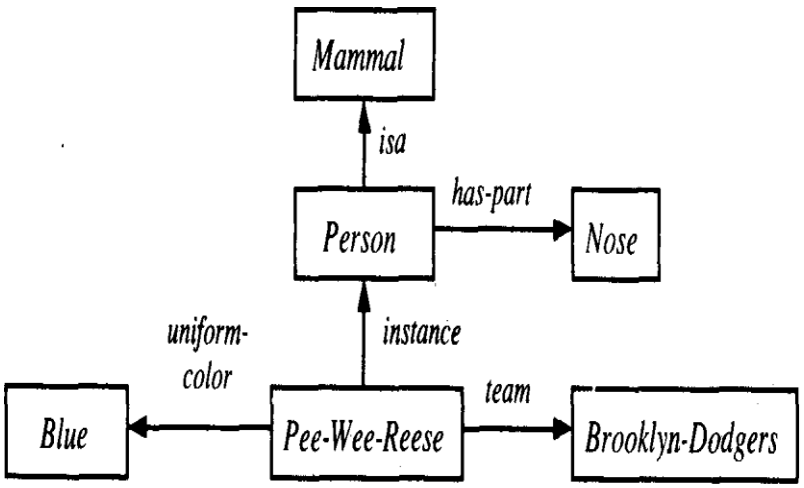


Figure 5.1: A Semantic Network

represent relationships among the nodes. A fragment of a typical semantic net is shown in Figure 5.1.

This network contains examples of both the *isa* and *instance* relations, as well as some other, more domain-specific relations like *team* and *uniform-color.* In this network, we could use inheritance to derive the additional relation

has-part(Pee-Wee-Reese, Nose)

5.1.1 Intersection Search

One of the early ways that semantic nets were used was to find relationships among objects by spreading activation out from each of two nodes and seeing where the activation meL This process is called *intersection search* . Using this process, it is possible to use the network of Figure 5.1 to answer questions such as "What is the connection between the Brooklyn Dodgers and blue?"' This kind of reasoning exploits one of the important advantages that slot-and-filler structures have over purely logical representations because it takes advantage of the entity-based organization of knowledge that slot-and-fiiler representations provide.

To answer more structured questions, however, requires networks that are themselves more highly structured. In the next few sections we expand and refine our notion of a network in order to support more sophisticated reasoning.

5.1.2 Representing Nonbinary Predicates

Semantic nets are a natural way to represent relationships that would appear as ground instances of binary predicates in predicate logic For example, some of the arcs from Figure 5.1 could be represented in logic as

Actually, to do this we need to assume that the inverses of the links we have shown also exist.

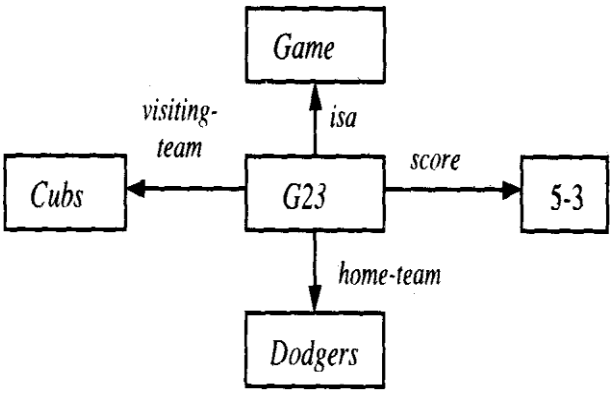


Figure 5.2: A Semantic Net for an w-Place Predicate

isa(Person, Mammal)

instance(Pee- Wee-Reese, Person)

team{Pee-Wee-Reese, Brooklyn-Dodgers)

uniform-color(Pee-Wee-Reese,Blue)

But the knowledge expressed by predicates of other arities can also be expressed in semantic nets. We have already seen that many unary predicates in logic can be thought of as binary predicates using some very general-purpose predicates, such as *isa* and *instance. So,* for ex-ample,

man(Marcus) could be
rewritten as

instance(Marcus, Man)

thereby making it easy to represent in a semantic net.

Three or more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationship to this new object of each of the original arguments. For example, suppose we know that

score(Cuhs, Dodgers, 5-3)

This can be represented in a semantic net by creating a node to represent the specific game and then relating each of the three pieces of information to it. Doing this produces the network shown in Figure 5.2.

This technique is particularly useful for representing the contents of a typical declarative sentence that describes several aspects of a particular event. The sentence

John gave the book to Mary.

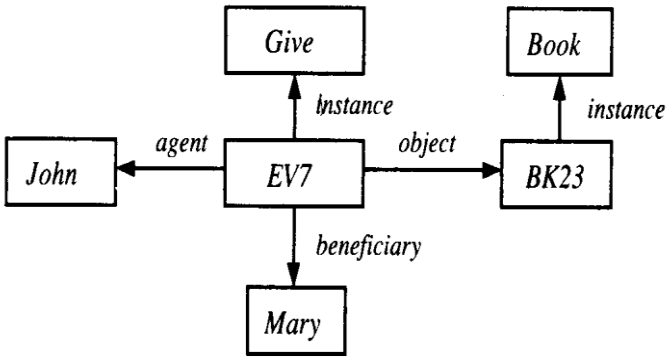
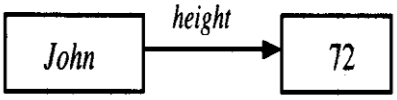


Figure 5.3: A Semantic Net Representing a Sentence

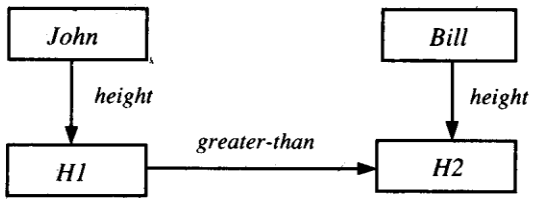
could be represented by the network shown in Figure 5.3.[2] In fact, several of the earliest uses of semantic nets were in English-understanding programs.

5.1.3 Making Some Important Distinctions

In the networks we have described so far, we have glossed over some distinctions that are important in reasoning. For example, there should be a difference between a link that defines a new entity and one that relates two existing entities. Consider the net



Both nodes represent objects that exist independently of their relationship to each other. But now suppose we want to represent the fact that John is taller than Bill, using the net

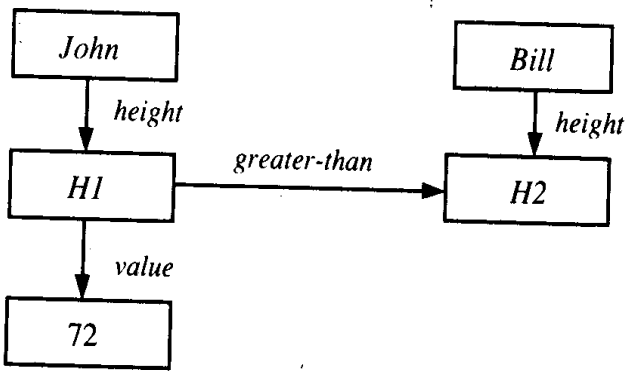


The nodes *HI* and *HI* are new concepts representing John's height and Bill's height, respectively. They are defined by their relationships to the nodes *John* and *Bill.* Using these defined concepts, it is possible to represent such facts as that John's height increased, which we could not do before. (The number 72 increased?)

Sometimes it is useful to introduce the are *value* to make this distinction clear. Thus we might use the following net to represent the fact that John is 6 feet tall and that he is

The node labeled *BK23* represents the particular book that was referred to by the phrase "the book." Discovering which particular book was meant by that phrase is similar to the problem of deciding on the correct referent for a pronoun, and it can be a very hard problem.

taller than Bill:



The procedures that operate on nets such as this can exploit the fact that some arcs, such as *height,* define new entities, while others, such as *greater-than* and *value,* merely describe relationships among existing entities.

Another example of an important distinction we have missed is the difference between the properties of a node itself and the properties that a node simply holds and passes on to its instances. For example, it is a property of the node *Person* that it is a subclass of the node *Mammal.* But the node *Person* does not have as one of its parts a nose. Instances of the node *Person* do, and we want them to inherit it.

It is difficult to capture these distinctions without assigning more structure to our notions of node, link, and value. In the next section, when we talk about frame systems, we do that. But first, we discuss a network-oriented solution to a simpler problem;

this solution illustrates what can be done in the network model but at what price in complexity.

5.1.4 Partitioned Semantic Nets

Suppose we want to represent simple quantified expressions in semantic nets. One way to do this is to *partition* the semantic

net into a hierarchical set *of spaces,* each of which corresponds to the scope of one or more variables. To see how this works, consider first the simple net shown in Figure 5.4(a). This net corresponds to the statement

The dog bit the mail carrier.

The nodes *Dogs, Bite,* and *Mail-Carrier* represent the classes of dogs, bitings, and mail carriers, respectively, while the nodes *a, h,* and *m* represent a particular dog, a particular biting, and a particular mail carrier. This fact can easily be represented by a single net with no partitioning.

But now suppose that we want to represent the fact

Every dog has bitten a mail carrier. or, in logic:

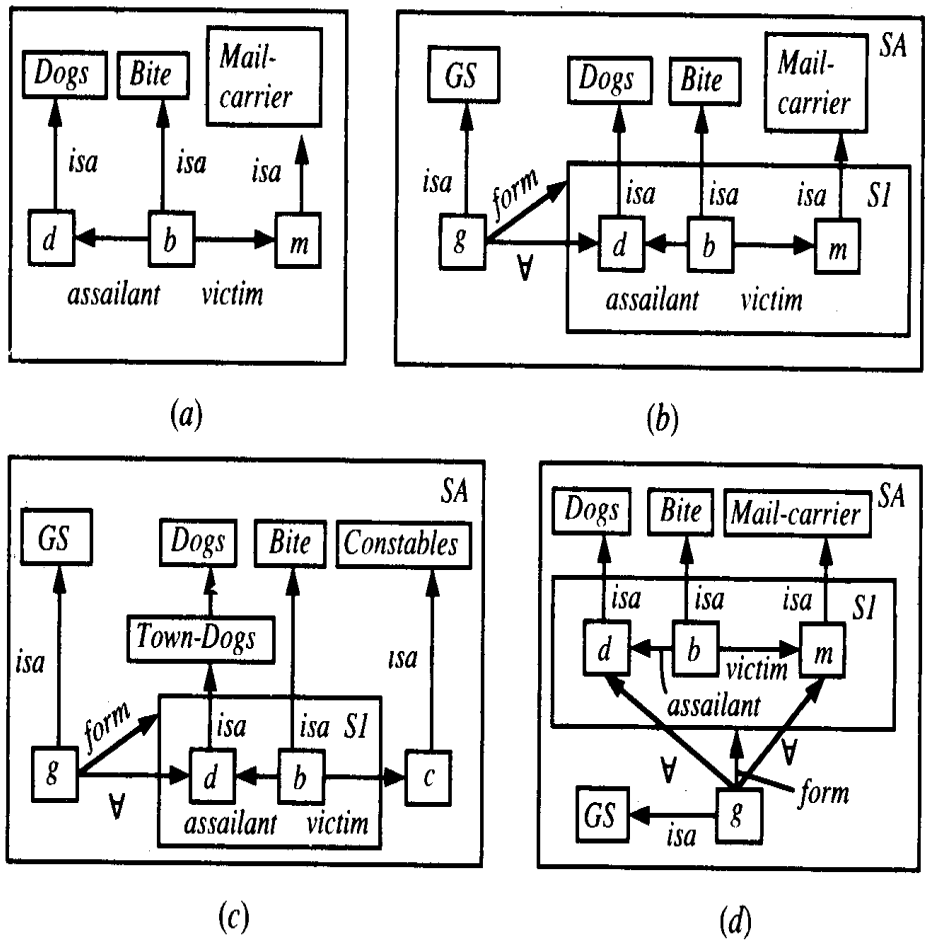


Figure 5.4: Using Partitioned Semantic Nets

V.v : *Dog(x) —> 3y : Mail-Carrier^)* A *Bite(x,y)*

To represent this fact, it is necessary to encode the scope of the universally quantified variable *x.* This can be done using partitioning as shown in Figure 5.*4(h).* The node *g* stands for the assertion given above. Node *g* is an instance of the special class *GS of* general statements about the world (i.e., those with universal quantifiers). Every element of GS has at least two attributes: *a form,* which states the relation that is being asserted, and one or more V connections, one for each of the universally quantified variables. In this example, there is only one such variable *d,* which can stand for any element of the class *Dogs.* The other two variables in the form, *h* and *m.* are understood to be existentially quantified. In other words, for every dog *d,* there exists a biting event *h,* and a mail carrier *m,* such that *d* is the assailant of *b* and *m* is the victim.

To see how partitioning makes variable quantification explicit, consider next the similar sentence:

Every dog in town has bitten the constable.

The representation of this sentence is shown in Figure 5.4(c). In this net, the node c representing the victim lies outside the form of the general statement. Thus it is not viewed as an existentially quantified variable whose value may depend on the value of *d.* Instead it is interpreted as standing for a specific entity (in this case, a particular

constable), just as do other nodes in a standard, nonpartitioned net. Figure 5.4*(d}* shows how yet another similar sentence:

Every dog has bitten every mail carrier.

would be represented. In this case, *g* has two V links, one pointing to *d,* which represents any dog, and one pointing to *m,* representing any mail carrier.

The spaces of a partitioned semantic net are related to each other by an inclusion hierarchy. For example, in Figure 5.4(*d),* space Sl is included in space *SA.* Whenever a search process operates in a partitioned semantic net, it can explore nodes and arcs in the space from which it starts and in other spaces

that contain the starting point, but it cannot go downward, except in special circumstances, such as when *a form* arc is being traversed. So, returning to Figure 5.4*(d),* from node *d* it can be determined that *d* must be a dog. But if we were to start at the node *Dogs* and search for all known instances of dogs by traversing *isa* links, we would not find *d* since it and the link to it are in the space Sl, which is at a lower level than space *SA,* which contains *Dogs.* This is important, since *d* does not stand for a particular dog; it is merely a variable that can be instantiated with a value that represents a dog.

5.1.5 The Evolution into Frames

The idea of a semantic net started out simply as a way to represent labeled connections among entities. But, as we have just seen, as we expand the range of problem-solving tasks that the representation must support, the representation itself necessarily begins to become more complex. In particular, it becomes useful to assign more structure to nodes as well as to links. Although there is no clear distinction between a semantic net and a frame system, the more structure the system has, the more likely it is to be termed a frame system. In the next section we continue our discussion of structured slot-and-filler representations by describing some of the most important capabilities that frame systems offer.

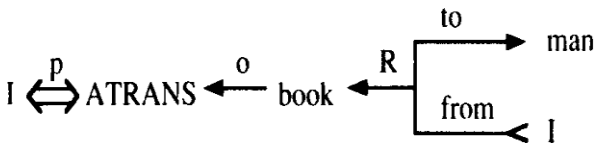
5.2. Conceptual Dependency

Conceptual dependency (often nicknamed CD) is a theory of how to represent the **kind** of knowledge about events that is usually contained in natural language sentences. **The** goal is to represent the knowledge in a way that

- Facilitates drawing inferences from the sentences.
- Is independent of the language in which the sentences were originally stated.

Because of the two concerns just mentioned, the CD representation of a sentence is built not out of primitives corresponding to the words used in the sentence, but rather out of conceptual primitives that can be combined to form the meanings of words in any particular language. The theory was

first described in Schank and was further developed in Schank It has since been implemented 'n a variety of programs that read and understand natural language text. Unlike semantic nets, which provide only a structure into which nodes representing information at any level can be placed. conceptual dependency provides both a structure and a specific set of primitives, at a particular level of granularity, out of which representations of particular pieces of information can be constructed.



where the symbols have the following meanings:

- Arrows indicate direction of dependency.
- Double arrow indicates two way link between actor and action.
- p indicates past tense.
- ATRANS is one of the primitive acts used by the theory. It indicates transfer of possession.
- o indicates the object case relation.
- R indicates the recipient case relation.

Figure 5.5: A Simple Conceptual Dependency Representation

As a simple example of the way knowledge is represented in CD, the event represented by the sentence

I gave the man a book.

would be represented as shown in Figure 5.5

In CD, representations of actions are built from a set of primitive acts. Although there are slight differences in the exact set of primitive actions provided in the various sources

on CD. a typical set is the following, taken from Schank and Abelson:

ATRANS Transfer of an abstract relationship (e.g., give)

PTRANS Transfer of the physical location of an object (e.g., go)

PROPEL Application of physical force to an object (e.g., push)

MOVE Movement of a body pan by its owner (e.g., kick)

GRASP Grasping of an object by an actor (e.g., clutch)

INGEST Ingestion of an object by an animal (e.g., eat)

EXPEL Expulsion of something from the body of an animal (e.g., cry)

MTRANS Transfer of mental information (e.g., tell)

MRUILD Building new information out of old (e.g., decide)

SPEAK Production of sounds (e.g., say)

ATTEND Focusing of a sense organ toward a stimulus (e.g., listen)

A second set of CD building blocks is the set of allowable dependencies among the conceptualizations described in a sentence. There are four primitive conceptual categories from which dependency structures can be built. These are

ACTs	Actions
PPs	Objects (picture producers)
AAs	Modifiers of actions (action aiders)
PAs	Modifiers of PPs (picture aiders)

In addition, dependency structures are themselves conceptualizations and can serve as components of larger dependency structures.

The dependencies among conceptualizations correspond to semantic relations among the underlying concepts. Figure 5.6 lists the most important ones allowed by CD.' The first column contains the rules; the second contains examples of their use; and the third contains an English version of each example. The rules shown in the figure can be interpreted as follows:

• Rule 1 describes the relationship between an actor and the event he or she causes. This is a two-way dependency sinee neither actor nor event can be considered primary. The letter p above the dependency link indicates past tense.

• Rule 2 describes the relationship between a PP and a PA that is being asserted to describe it. Many state descriptions, such as height, are represented in CD as numeric scales.

• Rule 3 describes the relationship between two PPs, one of which belongs to the set defined by the other.

• Rule 4 describes the relationship between a PP and an attribute that has already been predicated of it. The direction of the arrow is toward the PP being described.

• Rule 5 describes the relationship between two PPs, one of which provides a particular kind of information about the other. The three most common types of information to be provided in this way are possession (shown as POSS-BY), location (shown as LOC), and physical containment (shown as CONT). The direction of the arrow is again toward the concept being described.

• Rule 6 describes the relationship between an ACT and the PP that is the object of that ACT. The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.

• Rule 7 describes the relationship between an ACT and the source and the recipient of the ACT.

• Rule 8 describes the relationship between an ACT and the instrument with which it is performed. The instrument must always be a full conceptualization (i.e., it must contain an ACT), not just a single physical object.

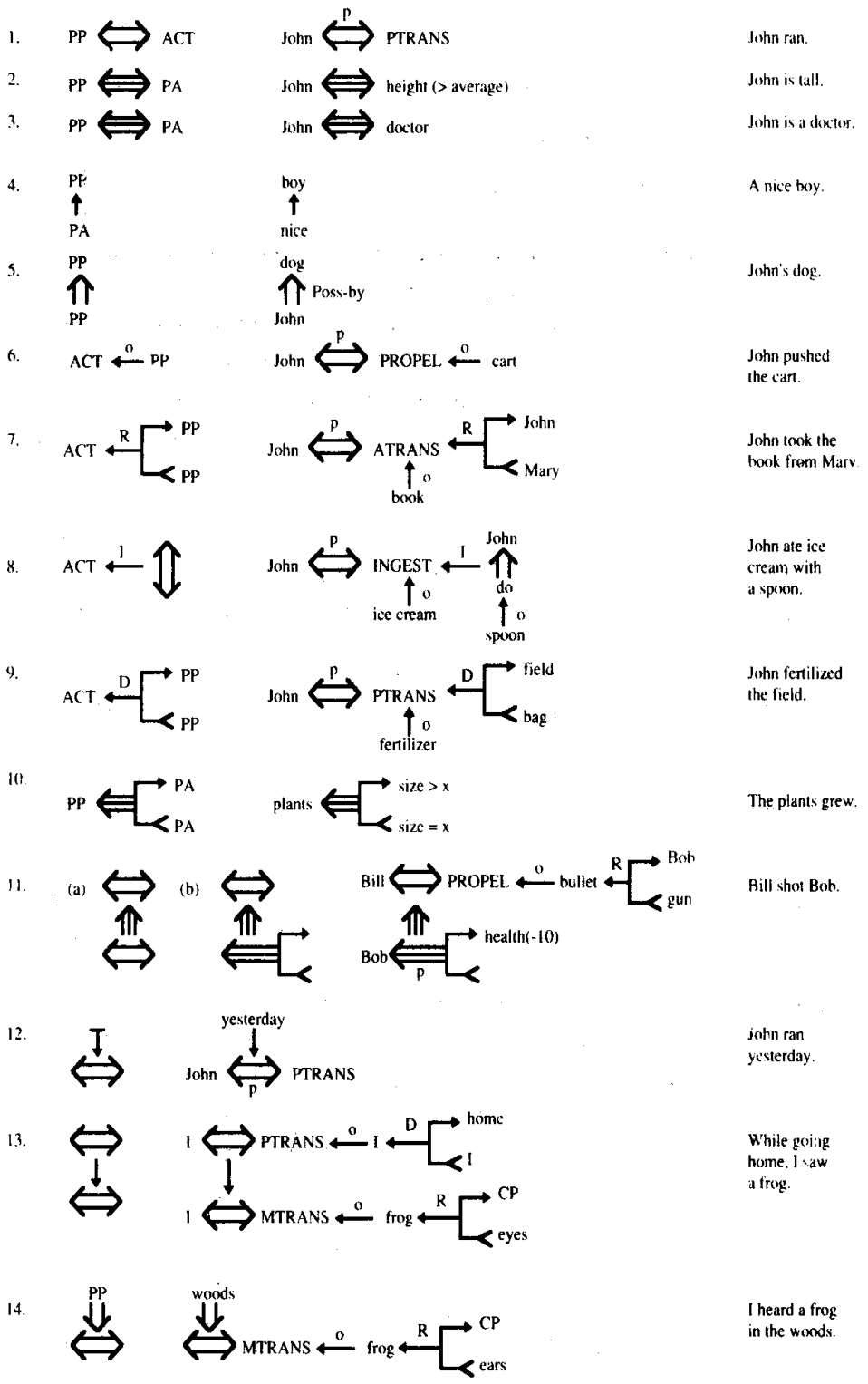


Figure 5.6: The Dependencies of CD

• Rule 9 describes the relationship between an ACT and its physical source and destination.

• Rule 10 represents the relationship between a PP and a state in which it started and another in which it ended.

• Rule 11 describes the relationship between one conceptualization and another that causes it. Notice that the arrows indicate dependency of one conceptualization on another and so point in the opposite direction of the implication arrows. The two forms of the rule describe the cause of an action and the cause of a state change.

• Rule 12 describes the relationship between a conceptualization and the time at which the event it describes occurred.

• Rule 13 describes the relationship between one conceptualization and another that is the time of the first. The example for this rule also shows how CD exploits a model of the human information processing system; *see* is represented as the transfer of information between the eyes and the conscious processor.

• Rule 14 describes the relationship between a conceptualization and the place at which it occurred.

Conceptualizations representing events can be modified in a variety of ways to supply information normally indicated in language by the tense, mood, or aspect of a verb form. The use of the modifier p to indicate past tense has already been shown. The set of conceptual tenses proposed by Schank includes

p	**Past**
f	**Future**
t	**Transition**
t_s	**Start transition**
t_f	**Finished transition**
k	**Continuing**
?	**Interrogative**
/	**Negative**
nil	**Present**
delta	**Timeless**
c	**Conditional**

As an example of the use of these tenses, consider the CD representation shown in Figure 5.7 (taken from Schank [1973]) of the sentence

Since smoking can kill you, I stopped.

The vertical causality link indicates that smoking kills one. Since it is marked c, however, we know only that smoking can kill one, not that it necessarily does. The horizontal causality link indicates that it is that first causality that made me stop smoking. The qualification t attached to the dependency between I and INGEST indicates that the smoking (an instance of INGESTING) has stopped and that the stopping happened in the past.

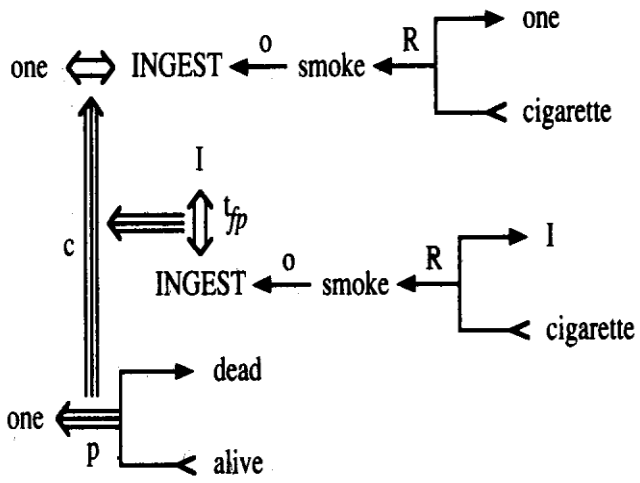


Figure 5.7: Using Conceptual Tenses

There are three important ways in which representing knowledge using the conceptual dependency model facilitates reasoning with the knowledge:

1. Fewer inference rules are needed than would be required if knowledge were not broken down into primitives.

2. Many inferences are already contained in the representation itself.

3. The initial structure that is built to represent the information contained in one sentence will have holes that need to be filled. These holes can serve as an attention focuser for the program that must understand ensuing sentences.

Each of these points merits further discussion.

The first argument in favor of representing knowledge in terms of CD primitives rather than in the higher-level terms in which it is normally described is that using the primitives makes it easier to describe the inference rules by which the knowledge can be manipulated. Rules need only be represented once for each primitive ACT rather than once for every word that describes that ACT. For example, all of the following verbs involve a transfer of ownership or an object:

- Give
- Take
- Steal
- Donate

If any of them occurs, then inferences about who now has the object and who once had the object (and thus who may know something about it) may be important. In a CD representation, those possible inferences can be slated once and associated with the primitive ACT ATRANS.

A second argument in favor of the use of CD representation is that to construct it, we must use not only the information that is stated explicitly in a sentence but also a set

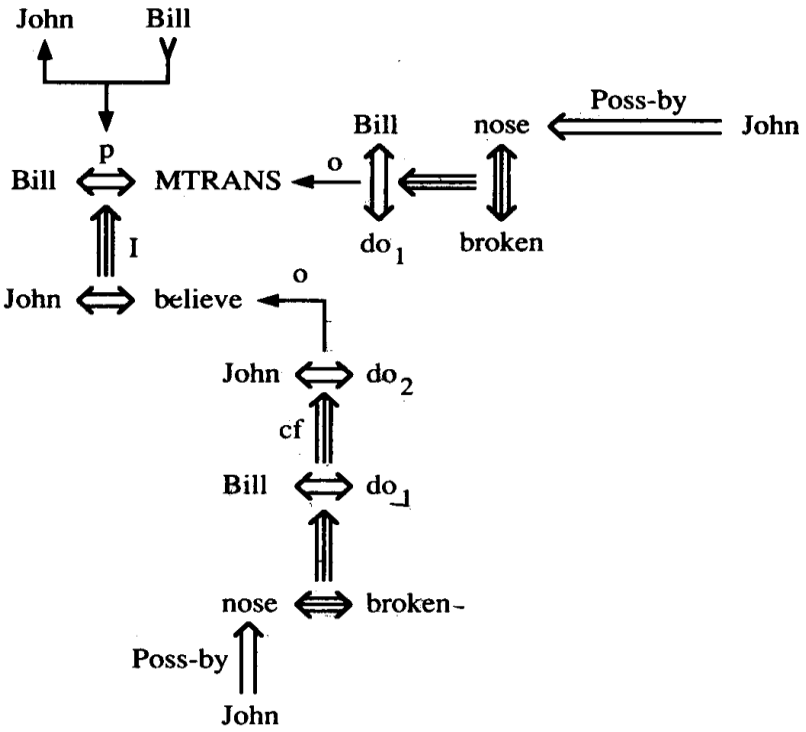


Figure 5.8: The CD Representation of a Threat

of inference rules associated with the specific information. Having applied these rules once, we store these results as part of the representation and they can be used repeatedly without the rules being reapplied. For example, consider the sentence

Bill threatened John with a broken nose.

The CD representation of the information contained in this sentence is shown in Figure 10.4. (For simplicity, *believe* is shown as a single unit. In fact, it must be represented in terms of primitive ACTs and a model of the human information processing system.) It says that Bill informed John that he (Bill) will do something to break John's nose. Bill did this so that John will believe that if he (John) does some other thing (different from what Bill will do to break his nose), then Bill will break John's nose. In this representation, the word "believe" has been used to simplify the example. But the! idea behind *believe* can be represented in CD as an MTRANS of a fact into John's memory. The actions doj and do^ are dummy placeholders that refer to some as yet unspecified actions.

A third argument for the use of the CD representation is that unspecified elements of the representation of one piece of information can be used as a focus for the understanding of later events as they are encountered. So, for example, after hearing that

Bill threatened John with a broken nose.

we might expect to find out what action Bill was trying to prevent John from performing. That action could then be substituted for the dummy action represented in Figure 5.8 as do2. The presence of such dummy objects provides clues as to what other events or objects are important for the understanding of the known event.

Of course, there are also arguments against the use of CD as a representation formalism. For one thing, it requires that all knowledge be decomposed into fairly low-level primitives. We discussed how this may be inefficient or perhaps even impossible in some situations as we put it,

> CD is a theory of representing fairly simple actions. To express, for example, "John bet Sam fifty dollars that the Mets would win the World Series" takes about two pages of CD forms. This does not seem reasonable.

Thus, although there are several arguments in favor of the use of CD as a model for representing events, it is not always completely appropriate to do so, and it may be worthwhile to seek out higher-level primitives.

Another difficulty with the theory of conceptual dependency as a general model for the representation of knowledge is that it is only a theory of the representation of events. But to represent all the information that a complex program may need, it must be able to represent other things besides events. There have been attempts to define a set of primitives, similar to those of CD for actions, that can be used to describe other kinds of knowledge. For example, physical objects, which in CD are simply represented as atomic units, have been analyzed in Lehnert . A similar analysis of social actions is provided in several other books. These theories continue the style of representation pioneered by CD, but they have not yet been subjected to the same amount of empirical investigation (i.e., use in real programs) as CD.

We have discussed the theory of conceptual dependency in some detail in order to illustrate the behavior of a knowledge representation system built around a fairly small set of specific primitive elements. But CD is not the only such theory to have been developed and used in AI programs. For another example of a primitive-based system, see Wilks [1972].

5.3. Scripts

CD is a mechanism for representing and reasoning about events. But rarely do events occur in isolation. In this section, we present a mechanism for representing knowledge about common sequences of events.

A *script* is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available. So far, this definition of a script looks very similar to that of a frame given, and at this level of detail, the two structures are identical. But now, because of the specialized role to be played by a script, we can make some more precise statements about its structure.

5.4. Model Questions

1. construct semantric net representations for the

   following:

a. Pompeian(marcus),blacksmith(marcus)

b. mary gave the green flowered vase to her favorite

   cousin.

2. suppose we want to use a semantic net discover relationships that could help in disambiguating the word “bank” in the sentence

John went downtown to deposit his money in the bank

The financial institution meaning for bank should be preferred over the river bank meaning.

a. Constructing a semantic net that contains representations for the relevant concepts.

b. Show how intersection search could be to find the connection between the correct meaning for bank and the rest of the sentence more easily than it can find a connection with the incorrect meaning.

3. construct partitioned semantic net representations for the following:

a. every batter hit a ball

b. all the batters like the pitcher.

4. show a conceptual dependency representation of the sentence

John begged mary for a pencil

How does this representation make it possible to answer the question

Did john talk to mary?

5. construct a script for going to a movie from the viewpoint of the movie goer.

6. would conceptual dependency be a good way to represent the contents of a typical issue of national geographic?

7. state where in the CYC ontology following concepts should fall:

a. cat

b. court case.

c. new York times

d. france

e. glass of water.

8. consider the following paragraph :

Jane was extremely hungry. She thought about going to her favorite restaurtant for dinner, but it was the day before payday. So instead she decided to go home and pop a frozen pizza in the oven. On the way though , she ran into her friend, judy. Judy invited jane to go out to dinner with her jane instantly agreed. When they got to their favorite place, they found a good table and relaxed over their meal.

How could the restaurant script be invoked by the contents of this story? Trace the process throughout the story. Might any other scripts also be invoked? For example, how wouyld your answer the question, “did jane pay for her dinner”?

UNIT - VI

Introduction

This lecture has two main goals:

1. To introduce Prolog's inbuilt abilities for performing arithmetic, and

2. To apply them to simple list processing problems, using accumulators.

Objectives

To be familiar with

Arithmetic in Prolog

Lists

Comparing integers

and other syntactic constructs of PROLOG

6.1 Arithmetic in Prolog

Prolog provides a number of basic arithmetic tools for manipulating integers (that is, numbers of the form ...-3, -2, -1, 0, 1, 2, 3, 4...). Most Prolog implementation also provide tools for handling real numbers (or floating point numbers) such as 1.53 or , but we're not going to discuss these, for they are not particularly useful for the symbolic processing tasks discussed in this course. Integers, on the other hand, are useful for various tasks (such as finding the length of a list), so it is important to understand how to work with them. We'll start by looking at how Prolog handles the four basic operations of addition, multiplication, subtraction, and division.

Arithmetic examples **Prolog Notation**

8 is 6+2.

12 is 6*2.

4 is 6-2.

-2 is 6-8.

3 is 6/2.

3 is 7/2.

1 is the remainder when 7 is divided by 2 1 is mod(7,2).

(Note that as we are working with integers, division gives us back an integer answer. Thus

gives 3 as an answer, leaving a reminder of 1.)

Posing the following queries yields the following responses:

?- 8 is 6+2. yes

?- 12 is 6*2. yes

?- -2 is 6-8. yes

?- 3 is 6/2. yes

?- 1 is mod(7,2). Yes

More importantly, we can work out the answers to arithmetic questions by using variables. For

example:

?- X is 6+2.

X = 8

?- X is 6*2.

X = 12

?- R is mod(7,2).

R = 1

Moreover, we can use arithmetic operations when we define predicates. Here's a simple example. Let's define a predicate add_3_and_double2/ whose arguments are both integers.

This predicate takes its first argument, adds three to it, doubles the result, and returns the number obtained as the second argument. We define this predicate as follows:

add_3_and_double(X,Y) :- Y is (X+3)*2.

And indeed, this works:

?- add_3_and_double(1,X).

X = 8

?- add_3_and_double(2,X).

X = 10

One other thing. Prolog understands the usual conventions we use for disambiguating arithmetical expressions. For example, when we write we mean and not and Prolog knows this convention:

?- X is 3+2*4.

X = 11

6.2 A closer look

That's the basics, but we need to know more. The most important to grasp is this: +, *, -, and mod do not carry out any arithmetic. In fact, expressions such as 3+2, 3-2 and 3*2 are simply terms. The functors of these terms are +, - and * respectively, and the arguments are 3 and 2. Apart from the fact that the functors go between their arguments (instead of in front

of them) these are ordinary Prolog terms, and unless we do something special, Prolog will not actually do any arithmetic. In particular, if we pose the query

?- X = 3+2

we don't get back the answer X=5. Instead we get back

X = 3+2

yes

That is, Prolog has simply bound the variable X to the complex term 3+2. It has not carried out any arithmetic. It has simply done what it usually does: performed unification Similarly, if

we pose the query

?- 3+2*5 = X

we get the response

X = 3+2*5

yes

Again, Prolog has simply bound the variable X to the complex term 3+2*5. It did not evaluate this expression to 13. To force Prolog to actually evaluate arithmetic expressions we have to use isjust as we did in our in our earlier examples. In fact, is does something very special: it sends a signal to Prolog that says `Hey! Don't treat this expression as an ordinary complex term! Call up your inbuilt arithmetic capabilities and carry out the calculations!'

In short, is forces Prolog to act in an unusual way. Normally rolog is quite happy just unifying variables to structures: that's its job, after all. Arithmetic is something extra that has been bolted on to the basic Prolog engine because it is useful. Unsurprisingly, there are some restrictions on this extra ability, and we need to know what they are.

For a start, the arithmetic expressions to be evaluated must be on the right hand side of is.

In our earlier examples we carefully posed the query

?- X is 6+2.

X = 8

which is the right way to do it. If instead we had asked

6+2 is X.

we would have got an error message saying instantiation_error, or something similar get some sort of instantiation_error message. And this makes perfect sense. Arithmetic *isn't performed using Prolog usual unification and knowledge base search mechanisms: it's done by calling up a special `black box' which knows about integer arithmetic. If we hand the black box the wrong kind of data, naturally its going to complain.*

Here's an example. Recall our `add 3 and double it' predicate.

add_3_and_double(X,Y) :- Y is (X+3)*2.

When we described this predicate, we carefully said that it added 3 to its first argument, doubled the result, and returned the answer in its second argument. For example, add_3_and_double(3,X) returns X = 12. We didn't say anything about using this predicate in the reverse direction. For example, we might hope that posing the query add_3_and_double(X,12).

would return the answer X=3. But it doesn't! Instead we get the instantiation_error message. Why? Well, when we pose the query this way round, we are asking Prolog to evaluate 12 is (X+3)*2, which it can't do as X is not instantiated.

Two final remarks. As we've already mentioned, for Prolog 3 + 2 is just a term. In fact, for Prolog, it really is the term +(3,2). The expression 3 + 2 is just a user-friendly notation that's nicer for us to use. This means that if you really want to, you can give Prolog queries like

X is +(3,2)

and Prolog will correctly reply

X = 5

Actually, you can even given Prolog the query

is(X,+(3,2))

and Prolog will respond

X = 5

This is because, for Prolog, the expression X is +(3,2) is the term is(X,+(3,2)). The expression X is +(3,2) is just user friendly notation. Underneath, as always, Prolog is just working away with terms.

Summing up, arithmetic in Prolog is easy to use. Pretty much all you have to remember is to use is to force evaluation, that stuff to be evaluated must goes to the right of is, and to take care that any variables are correctly instantiated. But there is a deeper lesson that is worth reflecting on. By `bolting on' the extra capability to do arithmetic we have further widened the distance between the procedural and declarative interpretation of Prolog processing.

6.3 Arithmetic and lists

Probably the most important use of arithmetic in this course is to tell us useful facts about data-structures, such as lists. For example, it can be useful to know how long a list is. We'll give some examples of using lists together with arithmetic capabilities.

How long is a list? Here's a recursive definition.

1. The empty list has length zero.

2. A non-empty list has length 1 + len(T), where len(T) is the length of its tail.

This definition is practically a Prolog program already. Here's the code we need:

```
len([],0).

len([_|T],N) :- len(T,X), N is X+1.
```

This predicate works in the expected way. For example:

```
?- len([a,b,c,d,e,[a,b],g],X).
```

X = 7

Now, this is quite a good program: it's easy to understand and efficient. But there is another method of finding the length of a list. We'll now look at this alternative, because it introduces the idea of accumulators, a standard Prolog technique we will be seeing lots more of.

If you're used to other programming languages, you're probably used to the idea of using variables to hold intermediate results. An accumulator is the Prolog analog of this idea.

Here's how to use an accumulator to calculate the length of a list. We shall define a predicate

accLen3/ which takes the following arguments.

accLen(List,Acc,Length)

Here List is the list whose length we want to find, and Length is its length (an integer).

What about Acc? This is a variable we will use to keep track of intermediate values for length (so it will also be an integer). Here's what we do. When we call this predicate, we are going to

give Acc an initial value of 0. We then recursively work our way down the list, adding 1 to Acc each time we find a head element, until we reach the empty list. When we do reach the empty set, Acc will contain the length of the list. Here's the code:

```
accLen([_|T],A,L) :-  Anew is A+1, accLen(T,Anew,L).

accLen([],A,A).
```

The base case of the definition, unifies the second and third arguments. Why? There are actually two reasons. The first is because when we reach the end of the list, the accumulator (the second variable) contains the length of the list. So we give this value (via unification) to the length variable (the third variable). The second is that this trivial unification gives a nice way of stopping the recursion when we reach the empty list. Here's an example trace:

?- accLen([a,b,c],0,L).

Call: (6) accLen([a, b, c], 0, _G449) ?

Call: (7) _G518 is 0+1 ?

Exit: (7) 1 is 0+1 ?

Call: (7) accLen([b, c], 1, _G449) ?

Call: (8) _G521 is 1+1 ?

Exit: (8) 2 is 1+1 ?

Call: (8) accLen([c], 2, _G449) ?

Call: (9) _G524 is 2+1 ?

Exit: (9) 3 is 2+1 ?

Call: (9) accLen([], 3, _G449) ?

Exit: (9) accLen([], 3, 3) ?

Exit: (8) accLen([c], 2, 3) ?

Exit: (7) accLen([b, c], 1, 3) ?

Exit: (6) accLen([a, b, c], 0, 3) ?

As a final step, we'll define a predicate which calls accLen for us, and gives it the initial value of 0:

leng(List,Length) :- accLen(List,0,Length).

So now we can pose queries like this:

leng([a,b,c,d,e,[a,b],g],X).

Accumulators are extremely common in Prolog programs. (We'll see another accumulator based program later in this lecture. And many more in the rest of the course.) But why is this?

In what way is accLen better than len? After all, it looks more difficult. The answer is that accLen is tail recursive while len is not. In tail recursive programs the result is all calculated once we reached the bottom of the recursion and just has to be

passed up. In recursive programs which are not tail recursive there are goals in one level of recursion which have to wait for the answer of a lower level of recursion before they can be evaluated. To understand this, compare the traces for the queries accLen([a,b,c],0,L) (see above) and len([a,b,c],0,L) (given below). In the first case the result is built while going into the recursion -- once the bottom is reached at accLen([],3,_G449) the result is there and only has to be passed up. In the second case the result is built while coming out of the recursion -- the result of len([b,c], _G481), for instance, is only computed after the recursive call of len has been completed and the result of len([c], _G489) is known.

?- len([a,b,c],L).

Call: (6) len([a, b, c], _G418) ?

Call: (7) len([b, c], _G481) ?

Call: (8) len([c], _G486) ?

Call: (9) len([], _G489) ?

Exit: (9) len([], 0) ?

Call: (9) _G486 is 0+1 ?

Exit: (9) 1 is 0+1 ?

Exit: (8) len([c], 1) ?

Call: (8) _G481 is 1+1 ?

Exit: (8) 2 is 1+1 ?

Exit: (7) len([b, c], 2) ?

Call: (7) _G418 is 2+1 ?

Exit: (7) 3 is 2+1 ?

Exit: (6) len([a, b, c], 3) ?

6.4 Comparing integers

Some Prolog arithmetic predicates actually do carry out arithmetic all by themselves (that is, without the assistance of is). These are the operators that compare integers.

Arithmetic examples Prolog Notation

X < Y.

X =< Y.

X =:= Y.

X =\= Y.

X >= Y

X > Y

These operators have the obvious meaning:

2 < 4.

yes

2 =< 4.

yes

4 =< 4.

yes

4=:=4.

yes

4=\=5.

yes

4=\=4.

no

4 >= 4.

yes

4 > 2.

Yes

Moreover, they force both their right-hand and left-hand arguments to be evaluated:

2 < 4+1.

yes

2+1 < 4.

yes

2+1 < 3+2.

yes

Note that =:= really is different from =, as the following examples show:

4=4.

yes

2+2 =4.

no

2+2 =:= 4.

yes

That is, = tries to unify its arguments; it does not force arithmetic evaluation. That's =:='s job.

For example, all the following queries lead to instantiation errors.

X < 3.

3 < Y.

X =:= X.

Moreover, variables have to be instantiated to integers. The query

X = 3, X < 4.

succeeds. But the query

X = b, X < 4.

fails.

OK, let's now look at an example which puts Prolog's abilities to compare numbers to work.

We're going to define a predicate which takes takes a list of non-negative integers as its first argument, and returns the maximum integer in the list as its last argument. Again, we'll use an accumulator. As we work our way down the list, the accumulator will keep track of the highest integer found so far. If we find a higher value, the accumulator will be updated to this new value. When we call the program, we set accumulator to an initial value of 0. Here's the code. Note that there are two recursive clauses:

```
accMax([H|T],A,Max) :-
   H > A,
   accMax(T,H,Max).


accMax([H|T],A,Max) :-
   H =< A,
   accMax(T,A,Max).


accMax([],A,A).
```

The first clause tests if the head of the list is larger than the largest value found so far. If it is, we set the accumulator to this new value, and then recursively work through the tail of the list.

The second clause applies when the head is less than or equal to the accumulator; in this case we recursively work through the tail of the list using the old accumulator value. Finally, the base clause unifies the second and third arguments; it gives the highest value we found while going through the list to the last argument. Here's how it works:

accMax([1,0,5,4],0,_5810)

accMax([0,5,4],1,_5810)

accMax([5,4],1,_5810)

accMax([4],5,_5810)

accMax([],5,_5810)

accMax([],5,5)

Again, it's nice to define a predicate which calls this, and initializes the accumulator. But wait:

what should we initialize the accumulator too? If you say 0, this means you are assuming that all the numbers in the list are positive. But suppose we give a list of negative integers as

input. Then we would have

accMax([-11,-2,-7,-4,-12],0,Max).

Max = 0

yes

This is not what we want: the biggest number on the list is -2. Our use of 0 as the initial value of the accumulator has ruined everything, because it's bigger than any number on the list.

There's an easy way around this: since our input list will always be a list of integers, simply initialize the accumulator to the ead of the list. That way we guarantee that the accumulator is initialized to a number on the list. The following predicate does this for us:

```
max(List,Max) :-
    List = [H|_],
    accMax(List,H,Max).
```

So we can simply say:

```
max([1,2,46,53,0],X).
```

X = 53

yes

And furthermore we have:

```
max([-11,-2,-7,-4,-12],X).
```

X = -2

yes

6.5 Exercises

Exercise 6.1

How does Prolog respond to the following queries?

1. X = 3*4
2. X is 3*4.
3. 4 is X.

4. X = Y.
5. 3 is 1+2.
6. 3 is +(1,2)
7. 3 is X+2.
8. X is 1+2.
9. 1+2 is 1+2
10. is(X,+(1,2))
11. 3+2 = +(3,2)
12. *(7,5) = 7*5.
13. *(7,+(3,2)) = 7*(3+2).

maximum of a list of integers. By changing the code slightly, turn this into a 3-place

14. *(7,(3+2)) = 7*(3+2).
15. *(7,(3+2)) = 7*(+(3,2)).

Exercise 6.2

Define a 2-place predicate increment that holds only when its second argument is an integer one larger than its first argument. For example, increment(4,5) should hold, but increment(4,6) should not.

2. Define a 3-place predicate sum that holds only when its third argument is

the sum of the first two arguments. For example, sum(4,5,9) should hold, but sum(4,6,12)should not.

Exercise 6.3

Write a predicate addone2/ whose first argument is a list of integers, and whose second argument is the list of integers obtained by adding 1 to each integer in the first list. For example, the query

addone([1,2,7,2],X).

should give

X = [2,3,8,3].

6.6 Practical Session (optional)

The purpose of Practical Session 5 is to help you get familiar with Prolog's arithmetic capabilities, and to give you some further practice in list manipulation. To this end, we suggest the following programming exercises:

In the text we discussed the 3-place predicate accMax which which returned the predicate accMin which returns the minimum of a list of integers.

In mathematics, an n-dimensional vector is a list of numbers of length n. For example,

[2,5,12] is a 3-dimensional vector, and [45,27,3,-4,6] is a 5-dimensional vector.

One of the basic operations on vectors is scalar multiplication. In this operation, every element of a vector is multiplied by some number. For example, if we scalar multiply the 3-dimensional vector [2,7,4] by 3 the result is the 3-dimensional vector [6,21,12].

Write a 3-place predicate scalarMult whose first argument is an integer, whose second argument is a list of integers, and whose third argument is the result of scalar multiplying the second argument by the first. For example, the query scalarMult(3,[2,7,4],Result).

should yield

Result = [6,21,12]

Another fundamental operation on vectors is the dot product. This operation combines two vectors of the same dimension and yields a number as a result. The operation is carried out as follows: the corresponding elements of the two vectors are multiplied, and the results added. For example, the dot product of [2,5,6] and [3,4,1] is 6+20

+6, that is, 32. Write a 3-place predicate dot whose first argument is a list of integers, whose second argument is a list of integers of the same length as the first, and whose

third argument is the dot product of the first argument with the second. For example,

the query

dot([2,5,6],[3,4,1],Result).

should yield

Result = 32

instead of

[the,cow,under,the,table,shoots].

[a,dead,woman,likes,he].

UNIT - 7

OTHER FEATURES OF PROLOG

Introduction

This chapter describes the basic Prolog facts. They are the simplest form of Prolog predicates, and are similar to records in a relational database. As we will see in the next chapter they can be queried like database records.

Objectives

To be able to understand and develop prolog scripts to achieve the desired effects in artificial intelligence

7.1 FACTS

The syntax for a fact is

pred(arg1, arg2, ... argN).

where

pred

The name of the predicate

arg1, ...

The arguments

N

The arity The syntactic end of all Prolog clauses

A predicate of arity 0 is simply pred.

The arguments can be any legal Prolog term. The basic Prolog terms are integer

A positive or negative number whose absolute value is less than some implementation-specific

power of 2

atom

A text constant beginning with a lowercase letter variable

Begins with an uppercase letter or underscore (_)structure Complex terms

Various Prolog implementations enhance this basic list with other data types, such as floating point

Numbers, or strings.

The Prolog character set is made up of

q Uppercase letters, A-Z

q Lowercase letters, a-z

q Symbols, + - * / \ ^ , . ~ : . ? @ # $ &

q Digits, 0-9

Integers are made from digits. Other numerical types are allowed in some Prolog implementations.

Atoms are usually made from letters and digits with the first character being a lowercase letter, such as

hello

twoWordsTogether

x14

For readability, the underscore (_), but not the hyphen (-), can be used as a separator in longer names. So

the following are legal.

a_long_atom_name

z_23

The following are not legal atoms.

no-embedded-hyphens

123nodigitsatbeginning

_nounderscorefirst

Nocapsfirst

Use single quotes to make any character combination a legal atom as follows.

'this-hyphen-is-ok'

'UpperCase'

'embedded blanks'

Do not use double quotes ("") to build atoms. This is a special syntax that causes the character string to

be treated as a list of ASCII character codes.

Atoms can also be legally made from symbols, as follows.

-->

++

Variables are similar to atoms, but are distinguished by beginning with either an uppercase letter or the

underscore (_).

Using these building blocks, we can start to code facts. The predicate name follows the rules for atoms.

The arguments can be any Prolog terms.

Facts are often used to store the data a program is using. For example, a business application might have

customer/3.

customer('John Jones', boston, good_credit).

customer('Sally Smith', chicago, good_credit).

The single quotes are needed around the names because they begin with uppercase letters and because

they have embedded blanks.

> example the arguments give the window name and coordinates of the upper left and lower right corners

window(main, 2, 2, 20, 72).

window(errors, 15, 40, 20, 78).

A medical diagnostic expert system might have disease/2.

disease(plague, infectious).

A Prolog listener provides the means for dynamically recording facts and rules in the logicbase, as well

as the means to query (call) them. The logicbase is updated by 'consult'ing or 'reconsult'ing program

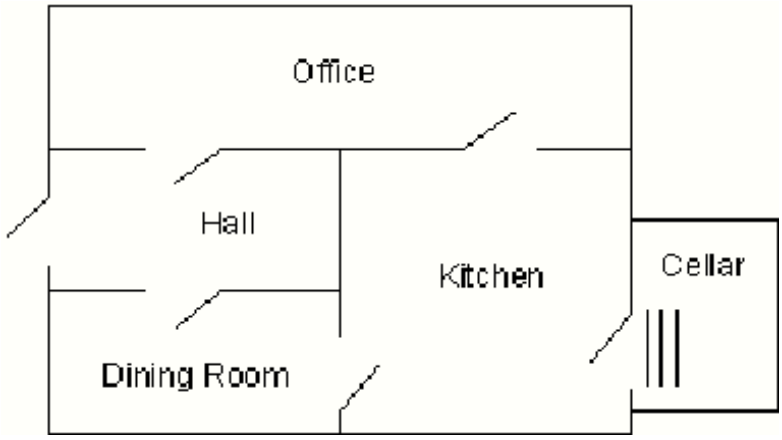
source. Predicates can also be typed directly into the listener, but they are not saved between sessions.

Nani Search

We will now begin to develop Nani Search by defining the basic facts that are meaningful for the game.

These include

- The rooms and their connections
- The things and their locations
- The properties of various things
- Where the player is at the beginning of the game



Open a new source file and save it as 'myadven.pro', or whatever name you feel is appropriate. You will make your changes to the program in that source file. (A completed version of nanisrch.pro is in the Prolog samples directory, samples/prolog/misc_one_file.)

First we define the rooms with the predicate room/1, which has five clauses, all of which are facts. They are based on the game map in figure 6.1.

room(kitchen).

room(office).

room(hall).

room('dining room').

room(cellar).

mean the thing and the second will mean its location. To begin with, we will add the following things.

location(desk, office).

location(apple, kitchen).

location(flashlight, desk).

location('washing machine', cellar).

location(nani, 'washing machine').

location(broccoli, kitchen).

location(crackers, kitchen).

location(computer, office).

The symbols we have chosen, such as kitchen and desk have meaning to us, but none to Prolog. The relationship between the arguments should also accurately reflect our meaning.

For example, the meaning we attach to location/2 is "The first argument is located in the second argument." Fortunately Prolog considers location(sink, kitchen) and location(kitchen, sink) to be different. Therefore, as long as we are consistent in our use of arguments, we can accurately represent our meaning and avoid the potentially ambiguous interpretation of the kitchen being in the sink.

We are not as lucky when we try to represent the connections between rooms. Let's start, however, with door/2, which will contain facts such as door(office, hall).

We would like this to mean "There is a connection from the office to the hall, or from the hall to the office."

accurately represent a two-way connection, we would have to define door/2 twice for each connection

door(office, hall).

door(hall, office).

The strictness about order serves our purpose well for location, but it creates this problem for connections between rooms. If the office is connected to the hall, then we would like the reverse to be true as well.

For now, we will just add one-way doors to the program; we will address the symmetry problem again in the next chapter and resolve it in chapter 5.

door(office, hall).

door(kitchen, office).

door(hall, 'dining room').

door(kitchen, cellar).

door('dining room', kitchen).

Here are some other facts about properties of things the game player might try to eat.

edible(apple).

edible(crackers).

tastes_yucky(broccoli).

Finally we define the initial status of the flashlight, and the player's location at the beginning of the game.

turned_off(flashlight).

here(kitchen).

We have now seen how to use basic facts to represent data in a Prolog program.

Exercises

During the course of completing the exercises you will develop three Prolog applications in addition to Nani Search. The exercises from each chapter will build on the work of previous chapters. Suggested solutions to the exercises are contained in the Prolog source files listed in the appendix, and are also included in samples/prolog/misc_one_file. The files are gene

A genealogical intelligent logicbase custord

A customer order entry application

Birds

An expert system that identifies birds

Not all applications will be covered in each chapter. For example, the expert system requires an understanding of rules and will not be started until the end of chapter 5.

Genealogical Logicbase

- First create a source file for the genealogical logicbase application. Start by adding a few members of your family tree. It is important to be accurate, since we will be exploring family relationships. Your own knowledge of who your relatives are will verify the correctness of your Prolog programs.

Start by recording the gender of the individuals. Use two separate predicates, male/1 and female/1. For example

male(dennis).

male(michael).

female(diana).

Remember, if you want to include uppercase characters or embedded blanks you must enclose the name in single (not double) quotes. For example male('Ghenghis Khan').

2- Enter a two-argument predicate that records the parent-child relationship. One argument represents the parent, and the other the child. It doesn't matter in which order you enter the arguments, as long as you are consistent. Often Prolog programmers adopt the convention that parent(A,B) is interpreted "A is the parent of B". For example

parent(dennis, michael).

parent(dennis, diana).

Customer Order Entry

3- Create a source file for the customer order entry program. We will begin it with three record types (predicates). The first is customer/3 where the three arguments are

arg1

Customer name

arg2

City

arg3

Credit rating (aaa, bbb, etc)

Add as many customers as you see fit.

4- Next add clauses that define the items that are for sale. It should also have three arguments

arg1

Item identification number

arg2

Item name

arg3

The reorder point for inventory (when at or below this level, reorder)

5- Next add an inventory record for each item. It has two arguments.

arg1

Item identification number (same as in the item record)

arg2

Amount in stock

7.2 Rules

We said earlier a predicate is defined by clauses, which may be facts or rules. A rule is no more than a stored query. Its syntax is

head :- body.

where

head a predicate definition (just like a fact)

the neck symbol, sometimes read as "if"

body

one or more goals (a query)

For example, the compound query that finds out where the good things to eat are can be stored as a rule with the predicate name where_food/2.

```
where_food(X,Y) :-
 location(X,Y),
 edible(X).
```

It states "There is something X to eat in room Y if X is located in Y, and X is edible."

We can now use the new rule directly in a query to find things to eat in a room. As before, the semicolon

(;) after an answer is used to find all the answers.

```
?- where_food(X, kitchen).
X = apple ;
X = crackers ;
no
?- where_food(Thing, 'dining room').
no
```

Or it can check on specific things

```
?- where_food(apple, kitchen).
yes
```

Or it can tell us everything.

```
?- where_food(Thing, Room).
Thing = apple
Room = kitchen ;
Thing = crackers
Room = kitchen ;
```

no

Just as we had multiple facts defining a predicate, we can have multiple rules for a predicate. For example, we might want to have the broccoli included in where_food/2. (Prolog doesn't have an opinion on whether or not broccoli is legitimate food. It just matches patterns.) To do this we add another

where_food/2 clause for things that 'taste_yucky.'

```
where_food(X,Y) :-
 location(X,Y),
 edible(X).
where_food(X,Y) :-
 location(X,Y),
 tastes_yucky(X).
```

Now the broccoli shows up when we use the semicolon (;) to ask for everything.

```
?- where_food(X, kitchen).
X = apple ;
X = crackers ;
X = broccoli ;
no
```

Until this point, when we have seen Prolog try to satisfy goals by searching the clauses of a predicate, all of the clauses have been facts.

How Rules Work

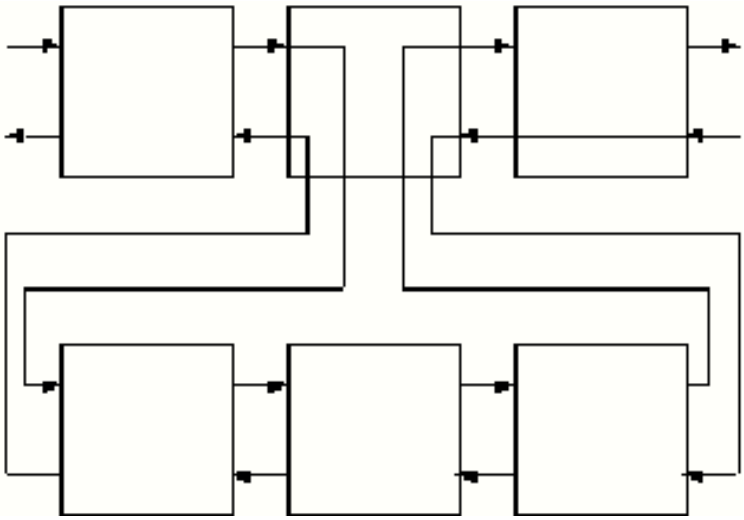
With rules, Prolog unifies the goal pattern with the head of the clause. If unification succeeds, then Prolog initiates a new query using the goals in the body of the clause.

Rules, in effect, give us multiple levels of queries. The first level is composed of the original goals. The next level is a new query

composed of goals found in the body of a clause from the first level.

Each level can create even deeper levels. Theoretically, this could continue forever. In practice it can continue until the listener runs out of space.

Figure 6.2 shows the control flow after the head of a rule has been matched. Notice how backtracking from the third goal of the first level now goes into the second level.



In this example, the middle goal on the first level succeeds or fails if its body succeeds or fails. When entered from the right (redo) the goal reenters its body query from the right (redo). When the query fails, the next clause of the first-level goal is tried, and if the next clause is also a rule, the process is repeated with the second clause's body.

As always with Prolog, these relationships become clearer by studying a trace. Figure 6.3 contains the annotated trace of the where_food/2 query. Notice the appearance of a two-part number. The first part of the number indicates the query level. The second part indicates the number of the goal within the query, as before. The parenthetical number is the clause number. For example

2-1 EXIT (7) location(crackers, kitchen)

means the exit occurred at the second level, first goal using clause number seven.

The query is

?- where_food(X, kitchen).

First the clauses of where_food/2 are searched.

1-1 CALL where_food(X, kitchen)

The pattern matches the head of the first clause, and while it is not at a port, the trace could inform us of the clause it is working on.

1-1 try (1) where_food(X, kitchen)

The body of the first clause is then set up as a query, and the trace continues.

2-1 CALL location(X, kitchen)

From this point the trace proceeds exactly as it did for the compound query in the previous chapter.

2-1 EXIT (2) location(apple, kitchen)

2-2 CALL edible(apple)

2-2 EXIT (1) edible(apple)

Since the body has succeeded, the goal from the previous (first) level succeeds.

1-1 EXIT (1) where_food(apple, kitchen)

X = apple ;

Backtracking goes from the first-level goal, into the second level, proceeding as before.

1-1 REDO where_food(X, kitchen)

2-2 REDO edible(apple)

2-2 FAIL edible(apple)

2-1 REDO location(X, kitchen)

2-1 EXIT (6) location(broccoli, kitchen)

2-2 CALL edible(broccoli)

2-2 FAIL edible(broccoli)

2-1 REDO location(X, kitchen)

2-1 EXIT (7) location(crackers, kitchen)

2-2 CALL edible(crackers)

2-2 EXIT (2) edible(crackers)

1-1 EXIT (1) where_food(crackers, kitchen)

X = crackers ;

Now any attempt to backtrack into the query will result in no more answers, and the query will fail

2-2 REDO edible(crackers)

2-2 FAIL edible(crackers)

2-1 REDO location(X, kitchen)

2-1 FAIL location(X, kitchen)

This causes the listener to look for other clauses whose heads match the query pattern. In our

example, the second clause of where_food/2 also matches the query pattern.

1-1 REDO where_food(X, kitchen)

Again, although traces usually don't tell us so, it is building a query from the body of the second clause.

1-1 try (2) where_food(X, kitchen)

Now the second query proceeds as normal, finding the broccoli, which tastes_yucky.

2-1 CALL location(X, kitchen)

2-1 EXIT (2) location(apple, kitchen)

2-2 CALL tastes_yucky(apple)

2-2 FAIL tastes_yucky(apple)

2-1 REDO location(X, kitchen)

2-1 EXIT (6) location(broccoli, kitchen)

2-2 CALL tastes_yucky(broccoli)

2-2 EXIT (1) tastes_yucky(broccoli)

1-1 EXIT (2) where_food(broccoli, kitchen)

X = broccoli ;

Backtracking brings us to the ultimate no, as there are no more where_food/2 clauses to try.

2-2 REDO tastes_yucky(broccoli)

2-2 FAIL tastes_yucky(broccoli)

2-1 REDO location(X,kitchen)

2-1 EXIT (7) location(crackers, kitchen)

2-2 CALL tastes_yucky(crackers)

2-2 FAIL tastes_yucky(crackers)

2-2 REDO location(X, kitchen)

2-2 FAIL location(X, kitchen)

1-1 REDO where_food(X, kitchen)

1-1 FAIL where_food(X, kitchen)

no

It is important to understand the relationship between the first-level and second-level variables in this

query. These are independent variables, that is, the X in the query is not the same as the X that shows up in the body of the

where_food/2 clauses, values for both happen to be equal due to unification.

To better understand the relationship, we will slowly step through the process of transferring control.

Subscripts identify the variable levels.

The goal in the query is

?- where_food(X1, kitchen)

The head of the first clause is

where_food(X2, Y2)

Remember the 'sleeps' example in chapter 3 where a query with a variable was unified with a fact with a variable? Both variables were set to be equal to each other. This is exactly what happens here. This might be implemented by setting both variables to a common internal variable. If either one takes on a new value, both take on a new value.

So, after unification between the goal and the head, the variable bindings are

X1 = _01

X2 = _01

Y2 = kitchen

The second-level query is built from the body of the clause, using these bindings.

location(_01, kitchen), edible(_01).

When internal variable _01 takes on a value, such as 'apple,' both X's then take on the same value. This is fundamentally different from the assignment statements that set variable values in most computer languages.

Using Rules

Using rules, we can solve the problem of the one-way doors. We can define a new two-way predicate with two clauses, called connect/2.

connect(X,Y) :- door(X,Y).

connect(X,Y) :- door(Y,X).

It says "Room X is connected to a room Y if there is a door from X to Y, or if there is a door from Y to

X." Note the implied 'or' between clauses. Now connect/2 behaves the way we would like.

?- connect(kitchen, office).

yes

?- connect(office, kitchen).

yes

We can list all the connections (which is twice the number of doors) with a general query.

?- connect(X,Y).

X = office

Y = hall ;

X = kitchen

Y = office ;

...

X = hall

Y = office ;

X = office

Y = kitchen ;

...

With our current understanding of rules and built-in predicates we can now add more rules to Nani Search. We will start with look/0, which will tell the game player where he or she is, what things are in the room, and which rooms are adjacent.

To begin with, we will write list_things/1, which lists the things in a room. It uses the technique developed at the end of chapter 4 to loop through all the pertinent facts.

```
list_things(Place) :-

location(X, Place),

 tab(2),

 write(X),

 nl,

 fail.
```

We use it like this.

```
?- list_things(kitchen).

 apple

 broccoli

 crackers

no
```

There is one small problem with list_things/1. It gives us the list, but it always fails. This is all right if we call it by itself, but we won't be able to use it in conjunction with other rules that follow it (to the right as illustrated in our diagrams). We can fix this problem by adding a second list_things/1 clause which always succeeds.

```
list_things(Place) :-

 location(X, Place),

 tab(2),

 write(X),
```

```
 nl,

 fail.

list_things(AnyPlace).
```

Now when the first clause fails (because there are no more location/2s to try) the second list_things/1 clause will be tried. Since its argument is a variable it will successfully match with anything, causing list_things/1 to always succeed and leave through the 'exit' port.

As with the second clause of list_things/1, it is often the case that we do not care what the value of a variable is, it is simply a place marker. For these situations there is a special variable called the

anonymous variable, represented as an underscore (_). For example

```
list_things(_).
```

as well as to facts, we can write list_connections/1 just like list_things/1 by using the connection/2 rule.

```
list_connections(Place) :-

 connect(Place, X),

tab(2),

 write(X),

 nl,

 fail.

list_connections(_).
```

Trying it gives us

```
?- list_connections(hall).

 dining room

 office
```

yes

Now we are ready to write look/0. The single fact here(kitchen) tells us where we are in the game.

```
look :-
 here(Place),
 write('You are in the '), write(Place), nl,
 write('You can see:'), nl,
 list_things(Place),
 write('You can go to:'), nl,
 list_connections(Place).
```

Given we are in the kitchen, this is how it works.

```
?- look.
You are in the kitchen
You can see:
 apple
 broccoli
 crackers
You can go to:
 office
 cellar
 dining room
yes
```

We now have an understanding of the fundamentals of Prolog, and it is worth summarizing what we have learned so far. We have seen the following about rules in Prolog.

q A Prolog program is a logicbase of interrelated facts and rules.

q The rules communicate with each other through unification, Prolog's built-in pattern matcher.

q The rules communicate with the user through built-in predicates such as write/1.

q The rules can be queried (called) individually from the listener.

We have seen the following about Prolog's control flow.

q The execution behavior of the rules is controlled by Prolog's built-in backtracking search

q We can force backtracking with the built-in predicate fail.

q We can force success of a predicate by adding a final clause with dummy variables as arguments

We now understand the following aspects of Prolog programming.

q Facts in the logicbase (locations, doors, etc.) replace conventional data definition.

q The backtracking search (list_things/1) replaces the coding of many looping constructs.

q Passing of control through pattern matching (connect/2) replaces conditional test and branch

q The rules can be tested individually, encouraging modular program development.

q Rules that call rules encourage the programming practices of procedure abstraction and data

With this level of understanding, we can make a lot of progress on the exercise applications. Take some time to work with the programs to consolidate your understanding before moving on to the following chapters.

Exercises

Nonsense Prolog

1- Consider the following Prolog logicbase.

a(a1,1).

a(A,2).

a(a3,N).

b(1,b1).

b(2,B).

b(N,b3).

c(X,Y) :- a(X,N), b(N,Y).

d(X,Y) :- a(X,N), b(Y,N).

d(X,Y) :- a(N,X), b(N,Y).

Predict the answers to the following queries, then check them with Prolog, tracing.

?- a(X,2).

?- b(X,kalamazoo).

?- c(X,b3).

?- c(X,Y).

?- d(X,Y).

Adventure Game

2- Experiment with the various rules that were developed during this chapter, tracing them all.

3- Write look_in/1 for Nani Search. It should list the things located in its argument. For example, look_in

(desk) should list the contents of the desk.

4- Build rules for the various family relationships that were developed as queries in the last chapter. For example

mother(M,C):-

parent(M,C),

female(M).

5- Build a rule for siblings. You will probably find your rule lists an individual as his/her own sibling.

Use trace to figure out why.

6- We can fix the problem of individuals being their own siblings by using the built-in predicate that succeeds if two values are unequal, and fails if they are the same. The predicate is \=(X,Y). Jumping ahead a bit (to operator definitions in chapter 12), we can also write it in the form X \= Y.

7- Use the sibling predicate to define additional rules for brothers, sisters, uncles, aunts, and cousins.

8- If we want to represent marriages in the family logicbase, we run into the two-way door problem we encountered in Nani Search. Unlike parent/2, which has two arguments with distinct meanings, married/2 can have the arguments reversed without changing the meaning.

Using the Nani Search door/2 predicate as an example, add some basic family data with a spouse/2 predicate. Then write the predicate married/2 using connect/2 as a model.

9- Use the new married predicate to add rules for uncles and aunts that get uncles and aunts by marriage as well as by blood. You should have two rules for each of these relationships, one for the blood case and one for the marriage case. Use trace to follow their behavior.

10- Explore other relationships, such as those between in-laws.

11- Write a predicate for grandparent/2. Use it to find both a grandparent and a grandchild.

grandparent(someone, X).

grandparent(X, someone).

Trace its behavior for both uses. Depending on how you wrote it, one use will require many more steps than the other. Write two predicates, one called grandparent/2 and one called grandchild/2. Order the goals in each so that they are efficient for their intended uses.

Customer Order Entry

12- Write a rule item_quantity/2 that is used to find the inventory level of a named item. This shields the user of this predicate from having to deal with the item numbers.

13- Write a rule that produces an inventory report using the item_quantity/2 predicate. It should display the name of the item and the quantity on hand. It should also always succeed. It will be similar to list_things/2.

14- Write a rule which defines a good customer. You might want to identify different cases of a good

customer.

Expert Systems

Expert systems are often called rule-based systems. The rules are "rules of thumb" used by experts to solve certain problems. The expert system includes an inference engine, which knows how to use the rules.

expert systems. Prolog is an excellent language for building any kind of expert system. However, certaintypes of expert systems can be built directly using Prolog's native rules. These systems are called

Structured selection systems.

The code listing for 'birds' in the appendix contains a sample system that can be used to identify birds.

You will be asked to build a similar system in the exercises. It can identify anything, from animals to cars to diseases.

15- Decide what kind of expert system you would like to build, and add a few initial identification rules.

For example, a system to identify house pets might have these rules.

pet(dog):- size(medium), noise(woof).

pet(cat):- size(medium), noise(meow).

pet(mouse):- size(small), noise(squeak).

16- For now, we can use these rules by putting the known facts in the logicbase. For example, if we add

size(medium) and noise(meow) and then pose the query pet(X) we will find X=cat.

?- size(medium) :- true.

recorded

?- noise(meow) :- true.

recorded

Jumping ahead, you can also use assert/1 like this

?- assert(size(medium)).

yes

?- assert(noise(meow)).

yes

These examples use the predicates in the general form attribute(value). In this simple example, the pet attribute is deduced. The size and noise attributes must be given.

17- Improve the expert system by having it ask for the attribute/values it can't deduce. We do this by first adding the rules

size(X):- ask(size, X).

noise(X):- ask(noise, X).

For now, ask/2 will simply check with the user to see if an attribute/value pair is true or false. It will use the built-in

predicate read/1 which reads a Prolog term (ending in a period of course).

```
ask(Attr, Val):-

 write(Attr),tab(1),write(Val),

 tab(1),write('(yes/no)'),write(?),

 read(X),

 X = yes.
```

The last goal, X = yes, attempts to unify X and yes. If yes was read, then it succeeds, otherwise, it fails.

Arithmetic

Prolog must be able to handle arithmetic in order to be a useful general purpose programming language.

However, arithmetic does not fit nicely into the logical scheme of things.

That is, the concept of evaluating an arithmetic expression is in contrast to the straight pattern matching we have seen so far. For this reason, Prolog provides the built-in predicate 'is' that evaluates arithmetic expressions. Its syntax calls for the use of operators.

X is <arithmetic expression>

The variable X is set to the value of the arithmetic expression. On backtracking it is unassigned.

The arithmetic expression looks like an arithmetic expression in any other programming language.

Here is how to use Prolog as a calculator.

```
?- X is 2 + 2.

X = 4

?- X is 3 * 4 + 2.

X = 14
```

Parentheses clarify precedence.

?- X is 3 * (4 + 2).

X = 18

?- X is (8 / 4) / 2.

X = 1

In addition to 'is,' Prolog provides a number of operators that compare two numbers. These include 'greater than', 'less than', 'greater or equal than', and 'less or equal than.' They behave more logically, and succeed or fail according to whether the comparison is true or false. Notice the order of the symbols in the greater or equal than and less than or equal operators. They are specifically constructed not to look like an arrow, so that you can use arrow symbols in your programs without confusion.

X > Y

X < Y

X >= Y

X =< Y

Here are a few examples of their use.

?- 4 > 3.

yes

?- 4 < 3.

no

?- X is 2 + 2, X > 3.

X = 4

?- X is 2 + 2, 3 >= X.

no

?- 3+4 > 3*2.

yes

They can be used in rules as well. Here are two example predicates. One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

c_to_f(C,F) :-

F is C * 9 / 5 + 32.

freezing(F) :- Here are some examples of their use.

?- c_to_f(100,X).

X = 212

yes

?- freezing(15).

yes

?- freezing(45).

no

Exercises

Customer Order Entry

1- Write a predicate valid_order that checks whether a customer order is valid. The arguments should be customer, item, and quantity. The predicate should succeed only if the customer is a valid customer with a good credit rating, the item is in stock, and the quantity ordered is less than the quantity in stock.

2- Write a reorder/1 predicate which checks inventory levels in the inventory record against the reorder quantity in the item record. It should write a message indicating whether or not it's time to reorder.

Recursion

Recursion in any language is the ability for a unit of code to call itself, repeatedly, if necessary.

Recursion is often a very powerful and convenient way of representing certain programming constructs.

In Prolog, recursion occurs when a predicate contains a goal that refers to itself.

As we have seen in earlier chapters, every time a rule is called, Prolog uses the body of the rule to create a new query with new variables. Since the query is a new copy each time, it makes no difference whether a rule calls another rule or itself.

A recursive definition (in any language, not just Prolog) always has at least two parts, a boundary condition and a recursive case.

The boundary condition defines a simple case that we know to be true. The recursive case simplifies the problem by first removing a layer of complexity, and then calling itself. At each level, the boundary condition is checked. If it is reached the recursion ends. If not, the recursion continues.

We will illustrate recursion by writing a predicate that can detect things which are nested within other things.

Currently our location/2 predicate tells us the flashlight is in the desk and the desk is in the office, but it does not indicate that the flashlight is in the office.

?- location(flashlight, office).

no

Using recursion, we will write a new predicate, is_contained_in/2, which will dig through layers of nested things, so that it will answer 'yes' if asked if the flashlight is in the office.

To make the problem more interesting, we will first add some more nested items to the game. We will continue to use the location predicate to put things in the desk, which in turn can have other things inside them.

location(envelope, desk).

location(stamp, envelope).

location(key, envelope).

To list all of things in the office, we would first have to list those things that are directly in the office, like the desk. We would then list the things in the desk, and the things inside the things in the desk.

If we generalize a room into being just another thing, we can state a two-part rule which can be used to deduce whether something is contained in (nested in) something else.

q A thing, T1, is contained in another thing, T2, if T1 is directly located in T2. (This is the

q A thing, T1, is contained in another thing, T2, if some intermediate thing, X, is located in

We will now express this in Prolog. The first rule translates into Prolog in a straightforward manner.

is_contained_in(T1,T2) :-

location(T1,T2).

The recursive rule is also straightforward. Notice that it refers to itself.

is_contained_in(T1,T2) :-

location(X,T2),

is_contained_in(T1,X).

Now we are ready to try it.

?- is_contained_in(X, office).

X = desk ;

X = computer ;

X = flashlight ;

X = envelope ;

X = stamp ;

X = key ;

?- is_contained_in(envelope, office).

yes

?- is_contained_in(apple, office).

How Recursion Works

As in all calls to rules, the variables in a rule are unique, or scoped, to the rule. In the recursive case, this means each call to the rule, at each level, has its own unique set of variables. So the values of X, T1, and T2 at the first level of recursion are different from those at the second level.

However, unification between a goal and the head of a clause forces a relationship between the variables of different levels. Using subscripts to distinguish the variables, and internal Prolog variables, we can trace the relationships for a couple of levels of recursion.

First, the query goal is

?- is_contained_in(XQ, office).

The clause with variables for the first level of recursion is

is_contained_in(T11, T21) :-

location(X1, T21),

is_contained_in(T11, X1).

When the query is unified with the head of the clause, the variables become bound. The bindings are

XQ = _01

T11 = _01

T21 = office

X1 = _02

Note particularly that XQ in the query becomes bound to T11 in the clause, so when a value of _01 is found, both variables are found.

With these bindings, the clause can be rewritten as

is_contained_in(_01, office) :-

location(_02, office),

is_contained_in(_01, _02).

When the location/2 goal is satisfied, with _02 = desk, the recursive call becomes

is_contained_in(_01, desk)

That goal unifies with the head of a new copy of the clause, at the next level of the recursion. After that

unification the variables are

XQ = _01 T11 = _01 T12 = _01

T21 = office T22 = desk

X1 = desk X2 = _03

When the recursion finds a solution, such as 'envelope,' all of the T1s and X0 immediately take on that

value. Figure 6.4 contains a full annotated trace of the query.

The query is

?- is_contained_in(X, office).

Each level of the recursion will have its own unique variables, but as in all calls to rules, the variables at a called level will be bound in some relationship to the variables at the calling level. In the following trace, we will use Prolog internal variables, so we can see which variables are bound together and which are not. The items directly in the office are found easily, as the variable _0 is

bound to X in the query and T1 in the rule.

1-1 CALL is_contained_in(_0, office)

1-1 try (1) is_contained_in(_0, office)

2-1 CALL location(_0, office)

2-1 EXIT location(desk, office)

1-1 EXIT is_contained_in(desk, office)

X = desk ;

2-1 REDO location(_0, office)

2-1 EXIT location(computer, office)

1-1 EXIT is_contained_in(computer, office)

X = computer ;

2-1 REDO location(_0,office)

2-1 FAIL location(_0,office)

When there are no more location(X, office) clauses, the first clause of is_contained_in/2 fails, and the second clause is tried. Notice that the call to location does not have its first argument bound to the same variable. It was X in the rule, and it gets a new internal value, _4. T1 stays bound to _0.

1-1 REDO is_contained_in(_0, office)

1-1 try (2) is_contained_in(_0, office)

2-1 CALL location(_4, office)

2-1 EXIT location(desk, office)

3-2 REDO is_contained_in(_0, flashlight)

3-2 try (2) is_contained_in(_0, flashlight)

4-1 CALL location(_11, flashlight)

4-1 FAIL location(_11, flashlight)

3-2 FAIL is_contained_in(_0, flashlight)

Next, it tries to find things in the envelope and comes up with the stamp.

3-1 REDO location(_7, desk)

3-1 EXIT location(envelope, desk)

3-2 CALL is_contained_in(_0, envelope)

4-1 CALL location(_0, envelope)

4-1 EXIT location(stamp, envelope)

3-2 EXIT is_contained_in(stamp, envelope)

2-2 EXIT is_contained_in(stamp, desk)

1-1 EXIT is_contained_in(stamp, office)

X = stamp ;

And then the key.

4-1 REDO location(_0,envelope)

4-1 EXIT location(key, envelope)

3-2 EXIT is_contained_in(key, envelope)

2-2 EXIT is_contained_in(key, desk)

1-1 EXIT is_contained_in(key, office)

X = key ;

And then it fails its way back to the beginning.

3-2 REDO is_contained_in(_0, envelope)

3-2 try (2) is_contained_in(_0, envelope)

4-1 CALL location(_11, envelope)

4-1 EXIT location(stamp, envelope)

4-2 CALL is_contained_in(_0, stamp)

5-1 CALL location(_0, stamp)

5-1 FAIL location(_0, stamp)

4-2 REDO is_contained_in(_0, stamp)

4-2 try(2) is_contained_in(_0, stamp)

5-1 CALL location(_14, stamp)

5-1 FAIL location(_14, stamp)

4-1 REDO location(_11, envelope)

4-1 EXIT location(key, envelope)

4-2 CALL is_contained_in(_0, key)

4-2 try (1) is_contained_in(_0, key)

5-1 CALL location(_0, key)

5-1 FAIL location(_0, key)

4-2 REDO is_contained_in(_0, key)

4-2 try (2) is_contained_in(_0, key)

5-1 CALL location(_14, key)

5-1 FAIL location(_14, key)

4-1 REDO location(_7, desk)

4-1 FAIL location(_7, desk)

3-1 REDO location(_4, office)

3-1 EXIT location(computer, office)

3-2 CALL is_contained_in(_0, computer)

4-1 CALL location(_0, computer)

4-1 FAIL location(_0, computer)

3-2 REDO is_contained_in(_0, computer)

4-1 CALL location(_7, computer)

4-1 FAIL location(_7, computer)

3-1 REDO location(_4, office)

3-1 FAIL location(_4, office)

no

When writing a recursive predicate, it is essential to ensure that the boundary condition is checked at each level . Otherwise, the program might recurse forever.

The simplest way to do this is by always defining the boundary condition first, ensuring that it is always tried first and that the recursive case is only tried if the boundary condition fails.

Pragmatics

We now come to some of the pragmatics of Prolog programming. First consider that the goal location(X,Y) will be satisfied by every clause of location/2. On the other hand, the goals location(X, office) or location(envelope, X) will be satisfied by fewer clauses.

Let's look again at the second rule for is_contained_in/2, and an equally valid alternate coding.

```
is_contained_in(T1,T2):-
 location(X,T2),
 is_contained_in(T1,X).
is_contained_in(T1,T2):-
 location(T1,X),
 is_contained_in(X,T2).
```

Both will give correct answers, but the performance of each will depend on the query. The query is_contained_in(X, office) will execute faster with the first version. That is because T2 is bound, making the search for location(X, T2) easier than if both variables were unbound. Similarly, the second version is faster for queries such as is_contained_in(key, X).

Exercises

Adventure Game

1- Trace the two versions of is_contained_in/2 presented at the end of the chapter to understand the performance differences between them.

2- Currently, the can_take/1 predicate only allows the player to take things which are directly located in a room. Modify it so it uses the recursive is_contained_in/2 so that a player can take anything in a room.

Genealogical Logicbase

3- Use recursion to write an ancestor/2 predicate. Then trace it to understand its behavior. It is possible to write endless loops with recursive predicates. The trace facility will help you debug ancestor/2 if it is not working correctly.

- Use ancestor/2 for finding all of a person's ancestors and all of a person's descendants. Based on your experience with grandparent/2 and grandchild/2, write a descendant/2 predicate optimized for descendants, as opposed to ancestor/2, which is optimized for ancestors.

Lists

Lists are powerful data structures for holding and manipulating groups of things.

In Prolog, a list is simply a collection of terms. The terms can be any Prolog data types, including structures and other lists. Syntactically, a list is denoted by square brackets with the terms separated by commas. For example, a list of things in the kitchen is represented as [apple, broccoli, refrigerator] This gives us an alternative way of representing the locations of things. Rather than having separate location predicates for each thing, we can have one location predicate per container, with a list of things in the container.

loc_list([apple, broccoli, crackers], kitchen).

loc_list([desk, computer], office).

loc_list([flashlight, envelope], desk).

loc_list([stamp, key], envelope).

For lists to be useful, there must be easy ways to access, add, and delete list elements. Moreover, we

loc_list(['washing machine'], cellar).

loc_list([nani], 'washing machine').

There is a special list, called the empty list, which is represented by a set of empty brackets ([]). It is also referred to as nil. It can describe the lack of contents of a place or thing.

loc_list([], hall)

Unification works on lists just as it works on other data structures. With what we now know about lists

we can ask

?- loc_list(X, kitchen).

X = [apple, broccoli, crackers]

?- [_,X,_] = [apples, broccoli, crackers].

X = broccoli

This last example is an impractical method of getting at list elements, since the patterns won't unify unless both lists have the same number of elements.

Should not have to concern ourselves about the number of list items, or their order

These two features allow us to write list utility predicates, such as member/2, which finds members of a list, and append/3, which joins two lists together. List predicates all follow a similar strategy--try something with the first element of a list, then recursively repeat the process on the rest of the list.

First, the special notation for list structures.

[X | Y]

When this structure is unified with a list, X is bound to the first element of the list, called the head. Y is bound to the list of remaining elements, called the tail.

We will now look at some examples of unification using lists. The following example successfully unifies because the two structures are syntactically equivalent. Note that the tail is a list.

?- [a|[b,c,d]] = [a,b,c,d].

yes

This next example fails because of misuse of the bar (|) symbol. What follows the bar must be a single term, which for all practical purposes must be a list. The example incorrectly has three terms after the bar.

?- [a|b,c,d] = [a,b,c,d].

no

Here are some more examples.

?- [H|T] = [apple, broccoli, refrigerator].

H = apple

T = [broccoli, refrigerator]

?- [H|T] = [a, b, c, d, e].

H = a

T = [b, c, d, e]

?- [H|T] = [apples, bananas].

H = apples

T = [bananas]

In the previous and following examples, the tail is a list with one element.

?- [H|T] = [a, [b,c,d]].

H = a

T = [[b, c, d]]

In the next case, the tail is the empty list.

?- [H|T] = [apples].

H = apples

T = []

The empty list does not unify with the standard list syntax because it has no head.

?- [H|T] = [].

no

NOTE: This last failure is important, because it is often used to test for the boundary condition in a recursive routine. That is, as long as there are elements in the list, a unification with the [X|Y] pattern will succeed. When there are no elements in the list, that unification fails, indicating that the boundary condition applies.

We can specify more than just the first element before the bar (|). In fact, the only rule is that what follows it should be a list.

?- [One, Two | T] = [apple, sprouts, fridge, milk].

One = apple

Two = sprouts

T = [fridge, milk]

tail of the left-hand list is unified with [Z]. In both cases, Prolog looks for the most general way to relate or bind the variables.

?- [X,Y|T] = [a|Z].

X = a

Y = _01

T = _03

Z = [_01 | _03]

?- [H|T] = [apple, Z].

H = apple

T = [_01]

Z = _01

Study these last two examples carefully, because list unification is critical in building list utility predicates.

A list can be thought of as a head and a tail list, whose head is the second element and whose tail is a list whose head is the third element, and so on.

?- [a|[b|[c|[d|[]]]]] = [a,b,c,d].

yes

We have said a list is a special kind of structure. In a sense it is, but in another sense it is just like any other Prolog term. The last example gives us some insight into the true nature of the list. It is really an ordinary two-argument predicate. The first argument is the head and the second is the tail. If we called it

dot/2, then the list [a,b,c,d] would be

dot(a,dot(b,dot(c,dot(d,[]))))

In fact, the predicate does exist, at least conceptually, and it is called dot, but it is represented by a period (.) instead of dot.

To see the dot notation, we use the built-in predicate display/1, which is similar to write/1, except it always uses the dot syntax for lists when it writes to the console.

?- X = [a,b,c,d], write(X), nl, display(X), nl.

[a,b,c,d]

.(a,.(b,.(c,.d(,[]))))

?- X = [Head|Tail], write(X), nl, display(X), nl.

[_01, _02]

.(_01,_02)

?- X = [a,b,[c,d],e], write(X), nl, display(X), nl.

[a,b,[c,d],e]

.(a,.(b,.(.(c,.(d,[])),.(e,[]))))

From these examples it should be clear why there is a different syntax for lists. The easier syntax makes for easier reading, but sometimes obscures the behavior of the predicate. It helps to keep this "real" structure of lists in mind when working with predicates that manipulate lists.

This structure of lists is well-suited for the writing of recursive routines. The first one we will look at is member/2, which determines whether or not a term is a member of a list.

As with most recursive predicates, we will start with the boundary condition, or the simple case. An element is a member of a list if it is the head of the list.

member(H,[H|T]).

This clause also illustrates how a fact with variable arguments acts as a rule.

The second clause of member/2 is the recursive rule. It says an element is a member of a list if it is a member of the tail of the list.

member(X,[H|T]) :- member(X,T).

The full predicate is

member(H,[H|T]).

member(X,[H|T]) :- member(X,T).

Note that both clauses of member/2 expect a list as the second argument. Since T in [H|T] in the second clause is itself a list, the recursive call to member/2 works.

?- member(apple, [apple, broccoli, crackers]).

yes

?- member(broccoli, [apple, broccoli, crackers]).

yes

?- member(banana, [apple, broccoli, crackers]).

no

The query is

?- member(b, [a,b,c]).

1-1 CALL member(b,[a,b,c])

The goal pattern fails to unify with the head of the first clause of member/2, because the pattern in the head of the first clause calls for the head of the list and first argument to be identical. The goal pattern can unify with the head of the second clause.

1-1 try (2) member(b,[a,b,c])

The second clause recursively calls another copy of member/2.

2-1 CALL member(b,[b,c])

It succeeds because the call pattern unifies with the head of the first clause.

2-1 EXIT (1) member(b,[b,c])

The success ripples back to the outer level. 1-1 EXIT (2) member(b,[a,b,c])

yes

As with many Prolog predicates, member/2 can be used in multiple ways. If the first argument is a variable, member/2 will, on backtracking, generate all of the terms in a given list.

?- member(X, [apple, broccoli, crackers]).

X = apple ;

X = broccoli ;

X = crackers ;

no

We will now trace this use of member/2 using the internal variables. Remember that each level has its own unique variables, but that they are tied together based on the unification patterns between the goal at one level and the head of the clause on the next level.

In this case the pattern is simple in the recursive clause of member. The head of the clause unifies X with the first argument of the original goal, represented by _0 in the following trace. The body has a call to member/2 in which the first argument is also X, therefore causing the next level to unify with the

same _0.

The query is?- member(X,[a,b,c]).

The goal succeeds by unification with the head of the first clause, if X = a.

1-1 CALL member(_0,[a,b,c])

1-1 EXIT (1) member(a,[a,b,c])

X = a ;

Backtracking unbinds the variable and the second clause is tried.

1-1 REDO member(_0,[a,b,c])

1-1 try (2) member(_0,[a,b,c])

It succeeds on the second level, just as on the first level.

2-1 CALL member(_0,[b,c])

Further backtracking causes an attempt to find a member of the empty list. The empty list does not

2-1 EXIT (1) member(b,[b,c])

1-1 EXIT member(b,[a,b,c])

X = b ;

Backtracking continues onto the third level, with similar results.

2-1 REDO member(_0,[b,c])

2-1 try (2) member(_0,[b,c])

3-1 CALL member(_0,[c])

3-1 EXIT (1) member(c,[c])

2-1 EXIT (2) member(c,[b,c])

1-1 EXIT (2) member(c,[a,b,c])

X = c ;

unify with either of the list patterns in the member/2 clauses, so the query fails back to the beginning

3-1 REDO member(_0,[c])

3-1 try (2) member(_0,[c])

4-1 CALL member(_0,[])

4-1 FAIL member(_0,[])

3-1 FAIL member(_0,[c])

2-1 FAIL member(_0,[b,c])

1-1 FAIL member(_0,[a,b,c])

no

Another very useful list predicate builds lists from other lists or alternatively splits lists into separate pieces. This predicate is usually called append/3. In this predicate the second argument is appended to the first argument to yield the third argument. For example

?- append([a,b,c],[d,e,f],X).

X = [a,b,c,d,e,f]

reducing the first list recursively The boundary condition states that if a list X is appended to the empty list, the resulting list is also X.

append([],X,X).

The recursive condition states that if list X is appended to list [H|T1], then the head of the new list is also

H, and the tail of the new list is the result of appending X to the tail of the first list.

append([H|T1],X,[H|T2]) :-

append(T1,X,T2).

The full predicate is

append([],X,X).

append([H|T1],X,[H|T2]) :-

append(T1,X,T2).

Real Prolog magic is at work here, which the trace alone does not reveal. At each level, new variable bindings are built, that are unified with the variables of the previous level. Specifically, the third argument in the recursive call to append/3 is the tail of the third argument in the head of the clause.

These variable relationships are included at each step in the annotated trace shown in Figure 6.7.

The query is

?- append([a,b,c],[d,e,f],X).

1-1 CALL append([a,b,c],[d,e,f],_0)

X = _0

2-1 CALL append([b,c],[d,e,f],_5)

_0 = [a|_5]

3-1 CALL append([c],[d,e,f],_9)

_5 = [b|_9]

4-1 CALL append([],[d,e,f],_14)

_9 = [c|_14]

By making all the substitutions of the variable relationships, we can see that at this point X is bound as follows (thinking in terms of the dot notation for lists might make append/3 easier to understand).

X = [a|[b|[c|_14]]]

We are about to hit the boundary condition, as the first argument has been reduced to the empty list.

Unifying with the first clause of append/3 will bind _14 to a value, namely [d,e,f], thus giving us the desired result for X, as well as all the other intermediate variables. Notice the bound third arguments at each level, and compare them to the variables in the call ports above.

4-1 EXIT (1) append([],[d,e,f],[d,e,f])

3-1 EXIT (2) append([c],[d,e,f],[c,d,e,f])

2-1 EXIT (2) append([b,c],[d,e,f],[b,c,d,e,f])

1-1 EXIT (2)append([a,b,c],[d,e,f],[a,b,c,d,e,f])

X = [a,b,c,d,e,f]

Like member/2, append/3 can also be used in other ways, for example, to break lists apart as follows.

?- append(X,Y,[a,b,c]).

X = []

Y = [a,b,c] ;

X = [a]

Y = [b,c] ;

X = [a,b]

Y = [c] ;

X = [a,b,c]

Y = [] ;

no

Using the List Utilities

Now that we have tools for manipulating lists, we can use them. For example, if we choose to use loc_list/2 instead of location/2 for storing things, we can write a new location/2 that behaves exactly like the old one, except that it computes the answer rather than looking it up. This illustrates the sometimes fuzzy line between data and procedure. The rest of the program cannot tell how location/2 gets its results, whether as data or by computation. In either case it behaves the same, even on backtracking.

```
location(X,Y):-

 loc_list(List, Y),

 member(X, List).
```

In the game, it will be necessary to add things to the loc_lists whenever something is put down in a room. We can write add_thing/3 which uses append/3. If we call it with NewThing and Container, it will provide us with the NewList.

```
add_thing(NewThing, Container, NewList):-

 loc_list(OldList, Container),

 append([NewThing],OldList, NewList).
```

Testing it gives

?- add_thing(plum, kitchen, X).

X = [plum, apple, broccoli, crackers]

However, this is a case where the same effect can be achieved through unification and the [Head|Tail] list notation.

```
add_thing2(NewThing, Container, NewList):-
```

```
  loc_list(OldList, Container),

 NewList = [NewThing | OldList].
```

It works the same as the other one.

```
?- add_thing2(plum, kitchen, X).

X = [plum, apple, broccoli, crackers]
```

We can simplify it one step further by removing the explicit unification, and using the implicit unification that occurs at the head of a clause, which is the preferred form for this type of predicate.

```
add_thing3(NewTh, Container,[NewTh|OldList]) :-

 loc_list(OldList, Container).
```

It also works the same.

```
?- add_thing3(plum, kitchen, X).

X = [plum, apple, broccoli, crackers]
```

In practice, we might write put_thing/2 directly without using the separate add_thing/3 predicate to build a new list for us.

```
put_thing(Thing,Place) :-

 retract(loc_list(List, Place)),

 asserta(loc_list([Thing|List],Place)).
```

situations. Sometimes backtracking over multiple predicates You might find that some parts of a particular application fit better with multiple facts in the logicbase and other parts fit better with lists. In these cases it is useful to know how to go from one format to the other.

Going from a list to multiple facts is simple. You write a recursive routine that continually asserts the head of the list. In this example we create individual facts in the predicate stuff/1.

```
break_out([]).

break_out([Head | Tail]):-
```

```
 assertz(stuff(Head)),
 break_out(Tail).
```

Here's how it works.

?- break_out([pencil, cookie, snow]).

yes

?- stuff(X).

X = pencil ;

X = cookie ;

X = snow ;

no

Transforming multiple facts into a list is more difficult. For this reason most Prologs provide built-in predicates that do the job. The most common one is findall/3. The arguments are

arg1

A pattern for the terms in the resulting list

arg2

A goal pattern

arg3

The resulting list

findall/3 automatically does a full backtracking search of the goal pattern and stores each result in the list. It can recover our stuff/1 back into a list.

?- findall(X, stuff(X), L).

L = [pencil, cookie, snow]

Fancier patterns are available. This is how to get a list of all the rooms connecting to the kitchen.

?- findall(X, connect(kitchen, X), L).

L = [office, cellar, 'dining room']

potential ambiguity. Here findall/3 builds a list of structures that locates the edible things?- findall(foodat(X,Y), (location(X,Y), edible(X)), L).

L = [foodat(apple, kitchen), foodat(crackers, kitchen)]

Exercises

List Utilities

1- Write list utilities that perform the following functions.

q Remove a given element from a list

q Find the element after a given element

q Split a list into two lists at a given element (Hint - append/3 is close.)

q Get the last element of a list

Count the elements in a list (Hint - the length of the empty list is 0, the length a non-empty list is

1 + the length of its tail.)

2- Because write/1 only takes a single argument, multiple 'writes' are necessary for writing a mixed string of text and variables. Write a list utility respond/1 which takes as its single argument a list of terms to be written. This can be used in the game to communicate with the player. For example respond(['You can''t get to the', Room, 'from here'])

3- Lists with a variable tail are called open lists. They have some interesting properties. For example, member/2 can be used to add items to an open list. Experiment with and trace the following queries.

?- member(a,X).

?- member(b, [a,b,c|X]).

?- member(d, [a,b,c|X]).

?- OpenL = [a,b,c|X], member(d, OpenL), write(OpenL).

Nonsense Prolog

4- Predict the results of the following queries.

?- [a,b,c,d] = [H|T].

?- [a,[b,c,d]] = [H|T].

?- [] = [H|T].

?- [a] = [H|T].

?- [apple,3,X,'What?'] = [A,B|Z].

?- [[a,b,c],[d,e,f],[g,h,i]] = [H|T].

?- [a(X,c(d,Y)), b(2,3), c(d,Y)] = [H|T].

Genealogical Logicbase

5- Consider the following Prolog program

parent(p1,p2).

parent(p2,p3).

parent(p3,p4).

parent(p4,p5).

ancestor(A,D,[A]) :- parent(A,D).

ancestor(A,D,[X|Z]) :-

parent(X,D),

ancestor(A,X,Z).

- What is the purpose of the third argument to ancestor?

7- Predict the response to the following queries. Check by tracing in Prolog.

?- ancestor(a2,a3,X).

?- ancestor(a1,a5,X).

?- ancestor(a5,a1,X).

?- ancestor(X,a5,Z).

Expert System

8- Lists provide a convenient way to provide a simple menu capability to our expert system. We can replace the 'ask' predicate with menuask/3 where appropriate. menuask/3 will ask the player to select an item from a menu. The format is menuask(Attribute, Value, List_of_Choices).

For example

size(X):- menuask(size, X, [large, medium, small]).

This requires two intermediate predicates, menu_display/2 and menu_select/2. The first writes each choice on a separate line preceded by a unique number. The second uses a number entered by the user to return the "nth" element of the list.

Operators

We have seen that the form of a Prolog data structure is

functor(arg1,arg2,...,argN).

This is the ONLY data structure in Prolog. However, Prolog allows for other ways to syntactically represent the same data structure. These other representations are sometimes called syntactic sugaring.

The equivalence between list syntax and the dot (.) functor is one example. Operator syntax is another.

Chapter 6 introduced arithmetic operators. In this chapter we will equate them to the standard Prolog data structures, and learn how to define any functor to be an operator.

Each arithmetic operator is an ordinary Prolog functor, such as -/2, +/2, and -/1. The display/1 predicate can be used to see the standard syntax.

?- display(2 + 2).

+(2,2)

?- display(3 * 4 + 6).

+(*(3,4),6)

?- display(3 * (4 + 6)).

*(3,+(4,6))

You can define any functor to be an operator, in which case the Prolog listener will be able to read the structure in a different format. For example, if location/2 was an operator we could write apple location kitchen.

instead of

location(apple, kitchen).

NOTE: The fact that location is an operator is of NO significance to Prolog's pattern matching. It simply means there is an alternative way of writing the same term.

Operators are of three types.

infix

Example: 3 + 4

prefix

Example: -7 postfix

Example: 8 factorial

They have a number representing precedence which runs from 1 to 1200. When a term with multiple operators is converted to pure syntax, the operators with higher precedences are converted first. A high precedence is indicated by a low number.

Operators are defined with the built-in predicate op/3, whose three arguments are precedence, associativity, and the operator name.

Associativity in the second argument is represented by a pattern that defines the type of operator. The first example we will see is the definition of an infix operator which uses the associativity pattern 'xfx.'

The 'f' indicates the position of the operator in respect to its arguments. We will see other patterns as we proceed.

For our current purposes, we will again rework the location/2 predicate and rename it is_in/2 to go with its new look, and we will represent rooms in the structure room/1.

is_in(apple, room(kitchen)).

We will now make is_in/2 an infix operator of arbitrary precedence 35.

?- op(35,xfx,is_in).

Now we can ask

?- apple is_in X.

X = room(kitchen)

or

?- X is_in room(kitchen).

X = apple

We can add facts to the program in operator syntax.

banana is_in room(kitchen).

To verify that Prolog treats both syntaxes the same we can attempt to unify them.

?- is_in(banana, room(kitchen)) = banana is_in room(kitchen).

yes

And we can use display/1 to look at the new syntax.

?- display(banana is_in room(kitchen)).

is_in(banana, room(kitchen))

Let's now make room/1 a prefix operator. Note that in this case the associativity pattern fx is used to indicate the functor comes before the argument. Also we chose a precedence (33) higher (higher precedence has lower number) than that used for is_in

(35) in order to nest the room structure inside the is_in structure.

?- op(33,fx,room).

Now room/1 is displayed in operator syntax.

?- room kitchen = room(kitchen).

yes

?- apple is_in X.

X = room kitchen\

The operator syntax can be used to add facts to the program.

pear is_in room kitchen.

?- is_in(pear, room(kitchen)) = pear is_in room kitchen.

yes

?- display(pear is_in room kitchen).

is_in(pear, room(kitchen))

CAUTION: If you mix up the precedence (easy to do) you will get strange bugs. If room/1 had a lower precedence (higher number) than is_in/2, then the structure would be room(is_in(apple, kitchen))

Not only doesn't this capture the information as intended, it also will not unify the way we want.

For completeness, an example of a candidate for a postfix operator would be turned_on. Again note that the 'xf' pattern says that the functor comes after the argument.

?- op(33,xf,turned_on).

We can now say

flashlight turned_on.

and

?- turned_on(flashlight) = flashlight turned_on.

yes

Operators are useful for making more readable data structures in a program and for making quick and easy user interfaces.

In our command-driven Nani Search, we use a simple natural language front end, which will be described in the last chapter. We could have alternatively made the commands operators so that

goto(kitchen)

becomes goto kitchen.

turn_on(flashlight)

becomes turn_on flashlight. take(apple)

becomes take apple.

It's not natural language, but it's a lot better than parentheses and commas.

We have seen how the precedence of operators affects their translation into structures. When operators are of equal precedence, the Prolog reader must decide whether to work from left to right, or right to left.

This is the difference between right and left associativity.

An operator can also be non-associative, which means an error is generated if you try to string two together.

The same pattern used for precedence is used for associativity with the additional character y. The options are

Infix:

xfx non-associative

xfy right to left

yfx left to right

Prefix fx non-associative

fy left to right

Postfix:

xf non-associative

yf right to left

The is_in/2 predicate is currently non-associative so this gets an error.

key is_in desk is_in office.

To represent nesting, we would want this to be evaluated from right to left.

?- op(35,xfy,is_in).

yes

?- display(key is_in desk is_in office).

is_in(key, is_in(desk, office))

If we set it left to right the arguments would be different.

yes?- display(key is_in desk is_in office).

is_in(is_in(key, desk), office)

We can override operator associativity and precedence with parentheses. Thus we can get our left to

right is_in to behave right to left like so.

?- display(key is_in (desk is_in office)).

is_in(key, is_in(desk, office))

Many built-in predicates are actually defined as infix operators. That means that rather than following the standard predicate(arg1,arg2) format, the predicate can appear between the arguments as arg1 predicate arg2.

The arithmetic operators we have seen already illustrate this. For example +, -, *, and / are used as you would expect. However, it is important to understand that these arithmetic

structures are just structures like any others, and do not imply arithmetic evaluation. 3 + 4 is not the same as 7 any more than plus

(3,4) is or likes(3,4). It is just +(3,4).

Only special built-in predicates, like is/2, actually perform an arithmetic evaluation of an arithmetic expression. As we have seen, is/2 causes the right side to be evaluated and the left side is unified with the evaluated result.

This is in contrast to the unification (=) predicate, which just unifies terms without evaluating them.

?- X is 3 + 4.

X = 7

?- X = 3 + 4.

X = 3 + 4

?- 10 is 5 * 2.

yes

?- 10 = 5 * 2.

no

Arithmetic expressions can be as arbitrarily complex as other structures.

?- X is 3 * 4 + (6 / 2).

X = 15

Even if they are not evaluated.

?- X = 3 * 4 + (6 / 2).

X = 3 * 4 + (6 / 2)

The operator predicates can also be written in standard notation.

?- X is +(*(3,4) , /(6,2)).

X = 15

?- 3 * 4 + (6 / 2) = +(*(3,4),/(6,2)).

yes

To underscore that these arithmetic operators are really ordinary predicates with no special meaning unless being evaluated by is/2, consider

?- X = 3 * 4 + likes(john, 6/2).

X = 3 * 4 + likes(john, 6/2).

?- X is 3 * 4 + likes(john, 6/2).

error

We have seen that Prolog programs are composed of clauses. These clauses are simply Prolog data structures written with operator syntax. The functor is the neck (:-) which is defined as an infix operator.

There are two arguments.

:-(Head, Body).

The body is a data structure with the functor 'and' represented by a comma (,). The body looks like

,(goal1, ,(goal2,,goal3))

&(goal1, &(goal2, & goal3))

and the following would be equivalent.

head :- goal1 & goal2 & goal3.

:-(head, &(goal1, &(goal2, & goal3))).

But that is not how it was done, so the two forms are

head :- goal1 , goal2 , goal3.

:-(head, ,(goal1, ,(goal2, , goal3))).

Every other comma has a different meaning.

The arithmetic operators are often used by Prolog programmers to syntactically join related terms. For example, the write/1 predicate takes only one argument, but operators give an easy way around this restriction.

?- X = one, Y = two, write(X-Y).

one - two

The slash (/) can be used the same way. In addition, some Prologs define the colon (:) as an operator just for this purpose. It can improve readability by removing some parentheses. For example, the complex structures for defining things in the game can be syntactically represented with the colon as well.

object(apple, size:small, color:red, weight:1).

A query looking for small things would be expressed

?- object(X, size:small, color:C, weight:W).

X = apple

C = red

W = 1

The pattern matching is the same as always, but instead of size(small) we use the pattern size:small,

which is really :(size,small).

Exercises

Adventure Game

1- Define all of the Nani Search commands as operators so the current version of the game can be played without parentheses or commas.

Genealogical Logicbase

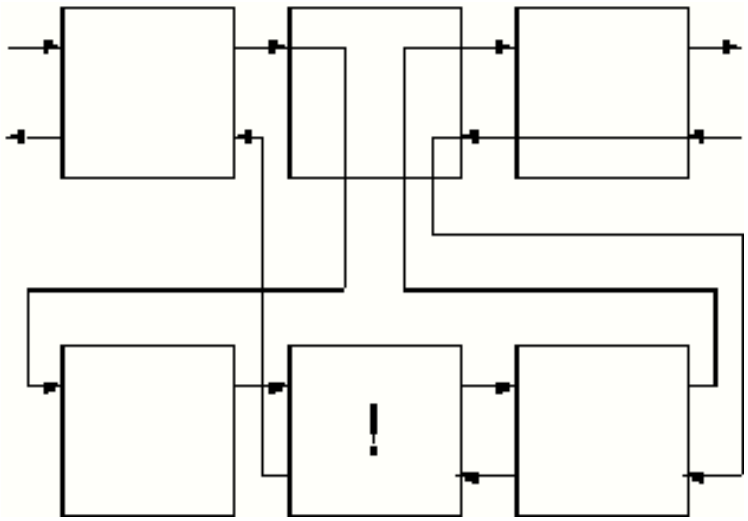
2- Define the various relationships in the genealogical logicbase as operators.

Cut

Up to this point, we have worked with Prolog's backtracking execution behavior. We have seen how to use that behavior to write compact predicates.

Sometimes it is desirable to selectively turn off backtracking. Prolog provides a predicate that performs this function. It is called the cut, represented by an exclamation point (!).

The cut effectively tells Prolog to freeze all the decisions made so far in this predicate. That is, if required to backtrack, it will automatically fail without trying other alternatives.



We will first examine the effects of the cut and then look at some practical reasons to use it.

When the cut is encountered, it re-routes backtracking, as shown in figure 6.8. It short-circuits backtracking in the goals to its left on its level, and in the level above, which contained the cut. That is, both the parent goal (middle goal of top level) and the goals of the particular rule being executed (second level) are affected by the cut. The effect is undone if a new route is taken into the parent goal.

Contrast figure 6.8 with figure 6.2.

We will write some simple predicates that illustrate the behavior of the cut, first adding some data to

backtrack over.

```
data(one).

data(two).

data(three).
```

Here is the first test case. It has no cut and will be used for comparison purposes.

```
cut_test_a(X) :-
 data(X).

cut_test_a('last clause').
```

This is the control case, which exhibits the normal behavior.

```
?- cut_test_a(X), write(X), nl, fail.

one

two

three

last clause

no
```

Next, we put a cut at the end of the first clause.

```
cut_test_b(X) :-
 data(X),
 !.

cut_test_b('last clause').
```

Note that it stops backtracking through both the data/1 subgoal (left), and the cut_test_b parent (above).

?- cut_test_b(X), write(X), nl, fail.

one

no

Next we put a cut in the middle of two subgoals.

```
cut_test_c(X,Y) :-
 data(X),
 !,
 data(Y).
cut_test_c('last clause').
```

Note that the cut inhibits backtracking in the parent cut_test_c and in the goals to the left of (before) the cut (first data/1). The second data/1 to the right of (after) the cut is still free to backtrack.

?- cut_test_c(X,Y), write(X-Y), nl, fail.

one - one

one - two

one - three

no

Performance is the main reason to use the cut. This separates the logical purists from the pragmatists.

Various arguments can also be made as to its effect on code readability and maintainability. It is often called the 'goto' of logic programming.

You will most often use the cut when you know that at a certain point in a given predicate, Prolog has either found the only answer, or if it hasn't, there is no answer. In this case you insert a cut in the predicate at that point.

Similarly, you will use it when you want to force a predicate to fail in a certain situation, and you don't want it to look any further.

Using the Cut

We will now introduce to the game the little puzzles that make adventure games fun to play. We will put them in a predicate called puzzle/1. The argument to puzzle/1 will be one of the game commands, and puzzle/1 will determine whether or not there are special constraints on that command, reacting accordingly.

We will see examples of both uses of the cut in the puzzle/1 predicate. The behavior we want is

q If there is a puzzle, and the constraints are met, quietly succeed.

q If there is a puzzle, and the constraints are not met, noisily fail.

q If there is no puzzle, quietly succeed.

The puzzle in Nani Search is that in order to get to the cellar, the game player needs to both have the flashlight and turn it on. If these criteria are met we know there is no need to ever backtrack through puzzle/1 looking for other clauses to try. For this reason we include the cut.

```
puzzle(goto(cellar)):-

 have(flashlight),

 turned_on(flashlight),

 !.
```

If the puzzle constraints are not met, then let the player know there is a special problem. In this case we also want to force the calling predicate to fail, and we don't want it to succeed by moving to other clauses of puzzle/1. Therefore we use the cut to stop backtracking, and we follow it with fail.

```
puzzle(goto(cellar)):-
```

write('It''s dark and you are afraid of the dark.'),

!, fail.

The final clause is a catchall for those commands that have no special puzzles associated with them.

They will always succeed in a call to puzzle/1.

puzzle(_).

For logical purity, it is always possible to rewrite the predicates without the cut. This is done with the built-in predicate not/1. Some claim this provides for clearer code, but often the explicit and liberal use of 'not' clutters up the code, rather than clarifying it.

When using the cut, the order of the rules becomes important. Our second clause for puzzle/1 safely prints an error message, because we know the only way to get there is by the first clause failing before it reached the cut.

The third clause is completely general, because we know the earlier clauses have caught the special cases.

If the cuts were removed from the clauses, the second two clauses would have to be rewritten.

puzzle(goto(cellar)):-

not(have(flashlight)),

not(turned_on(flashlight)),

write('Scared of dark message'),

fail.

puzzle(X):-

not(X = goto(cellar)).

In this case the order of the clauses would not matter.

It is interesting to note that not/1 is defined using the cut. It also uses call/1, another built-in predicate

that calls a predicate.

not(X) :- call(X), !, fail.

not(X).

In the next chapter we will see how to add a command loop to the game. Until then we can test the puzzle predicate by including a call to it in each individual command. For example

goto(Place) :-

puzzle(goto(Place)),

can_go(Place),

move(Place),

look.

Assuming the player is in the kitchen, an attempt to go to the cellar will fail.

?- goto(cellar).

It's dark and you are afraid of the dark.

no

?- goto(office).

You are in the office...

Then if the player takes the flashlight, turns it on, and return to the kitchen, all goes well.

?- goto(cellar).

You are in the cellar...

Exercises

Adventure Game

1- Test the puzzle/1 predicate by setting up various game situations and seeing how it responds. When testing predicates with cuts you should always use the semicolon (;) after each answer to make sure it behaves correctly on backtracking. In our case puzzle/1 should always give one response and fail on backtracking.

2- Add your own puzzles for different situations and commands.

Expert System

3- Modify the ask and menu ask predicates to use cut to replace the use of not.

Customer Order Entry

4- Modify the good_customer rules to use cut to prevent the search of other cases once we know one has been found.

Model questions:

1. for the following facts and recursive predicate, state what order solutions to the given query are returned:

```
on_top_of(prolog_book, desk).

on_top_of(ai_notes, prolog_book).

on_top_of(time_table, ai_notes).

on_top_of(ai_book, desk).

above(X,Y) :- on_top_of(X,Y).

above(X,Y) :- on_top_of(X,Z),
              above(Z,Y).

?- above(Object, desk).
```

2. What will happen if you try the following program/query:

```
above(prolog_book, desk).
```

```
above(ai_notes, prolog_book).

above(time_table, ai_notes).


above(X,Y) :- above(X,Z),
              above(Z,Y).


?- above(desk, ai_notes).
```

To check you understand terms, unification and backtracking try the following. I may ask questions like these in the exam! (though they wouldn't be worth many marks).

1. Which of the following are valid Prolog terms:

- 23
- foo(X, bar(+(3,4)))
- Foo(x)
- +(fred, jim)
- 1+2.
- Alison Cawsey

2. Which of the following match, and for the ones that match, what are the resultant bindings.

- a(1, 2) = a(X, X).
- a(X, 3) = a(4, Y).

- a(a(3, X)) = a(Y).
- 1+2 = 3.
- X = 1+2.
- a(X, Y) = a(1, X).
- a(X, 2) = a(1, X).

3. For the following (silly) program, state what order the solutions will be returned given the query flies(X). (Your answer should be of the form X=soln1; X=soln2; etc).

```
aeroplane(concorde).
aeroplane(jumbo).
on(fred, concorde).
on(jim, no_18_bus).
bird(percy).
animal(leo).
animal(tweety).
animal(peter).
has_feathers(tweety).
has_feathers(peter).
flies(X) :- bird(X).
flies(X) :- aeroplane(X).
flies(X) :- on(X, Y), aeroplane(Y).
bird(X) :- animal(X), has_feathers(X).
```

Once you have worked out the answers by hand, check them using Prolog.

1. What are the bindings which result from the following queries. If there is more than one solution, given the order in which they are returned:

?- [First, Second|Rest] = [cabbage, onion, tomato, orange]

?- [small(X), small(Y)] = [small(cabbage), small(tomato)]

?- [First, Second|Rest] = [a,b]

?- member(small(X), [large(cabbage), small(tomato)]).

?- member(small(carrot), [large(Y), small(Y)]).

?- member(large(X), [large(apple), large(banana)]).

?- member([X,Y], [[1,2],[3,4],[5,6]]).