4.3 Production Systems

4.3.1 Definition and History

The *production system* is a model of computation that has proved particularly importan
in AI, both for implementing search algorithms and for modeling human problem solv-
ing. A production system provides pattern-directed control of a problem-solving proces

and consists of a set of *production rules*, a *working memory*, and a *recognize-act* control cycle (Brownston et al. 1985).

D E F I N I T I O N

PRODUCTION SYSTEM

(A *production system* is defined by:

1. *The set of production rules*. These are often simply called *productions*. A production is a *condition-action* pair and defines a single chunk of problem-solving knowledge. The *condition part* of the rule is a pattern that determines when that rule may be applied to a problem instance. The *action part* defines the associated problem-solving step.
2. *Working memory* contains a description of the *current state of the world* in a reasoning process. This description is a pattern that is matched against the condition part of a production to select appropriate problem-solving actions. When the condition element of a rule is matched by the contents of working memory, the action associated with that condition may then be performed. The actions of production rules are specifically designed to alter the contents of working memory.
3. *The recognize-act cycle*. The control structure for a production system is straightforward: The current state of the problem-solving process is maintained as a set of patterns in *working memory*. Working memory is initialized with the beginning problem description. The patterns in working memory are matched against the conditions of the production rules; this produces a subset of the productions, called the *conflict set*, whose conditions match the patterns in working memory. The productions in the conflict set are said to be *enabled*. One of the productions in the conflict set is then selected (*conflict resolution*) and the production is *fired*. That is, the action of the rule is performed, changing the contents of working memory. After the selected production rule is fired, the control cycle repeats with the modified working memory. The process terminates when no rule conditions are matched by the contents of working memory.

Conflict resolution chooses a rule from the conflict set for firing. Conflict resolution strategies may be simple, such as selecting the first rule whose condition matches the state of the world, or may involve complex rule selection heuristics. This is an important way in which a production system allows the addition of heuristic control to a search algorithm.

The *pure* production system model has no mechanism for recovering from dead ends in the search; it simply continues until no more productions are enabled and halts. Many practical implementations of production systems allow backtracking to a previous state of working memory in such situations.

A schematic drawing of a production system is presented in Figure 4.3.

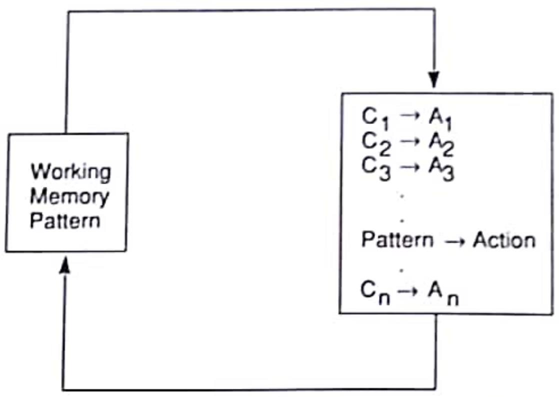

Figure 4.3 A production system. Control loops until working memory pattern no longer matches the conditions of any productions.

A very simple example of production system execution appears in Figure 4.4. This is a production system program for sorting a string composed of the letters a, b, and c. In this example, a production is enabled if its condition matches a portion of the string in working memory. When a rule is fired, the substring that matched the rule condition is replaced by the string on the right-hand side of the rule. As this example suggests, production systems are a general model of computation that can be programmed to do anything that can be done on a computer. Their real strength, however, is as an architecture for knowledge-based systems.)

The idea for the "production"-based design for computing came originally from writings of Post (1943), who proposed a production rule model as a formal theory of computation. The main construct of this theory was a set of rewrite rules for strings (in many ways similar to the parsing rules in the example in Chapter 3). It is also closely related to the approach taken by markov algorithms (Markov 1954).

An interesting application of production rules to modeling human cognition is found in the work of Newell and Simon at the Carnegie Institute of Technology (now Carnegie Mellon University) in the 1960s and 1970s. The programs they developed, including the *General Problem Solver*, are largely responsible for the importance of production systems in AI. In this research, human subjects were monitored in various problem-solving activities such as solving problems in predicate logic and playing games like chess. The *protocol* (behavior patterns, including verbal descriptions of the problem-solving process, eye movements, etc.) of problem-solving subjects was recorded and broken down to its elementary components. These components were regarded as the basic bits of problem-solving knowledge in the human subjects and were composed as a search through a graph (called the *problem behavior graph*). A production system was then used to implement search of this graph.

(The production rules represented the set of problem-solving skills of the human subject. The present focus of attention was represented as the current state of the world.

Production set:

1. ba → ab
2. ca → ac
3. cb → bc

Iteration #	Working memory	Conflict set	Rule fired
0	cbaca	1, 2, 3	1
1	cabca	2	2
2	acbca	3, 2	2
3	acbac	1, 3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	∅	Halt

Figure 4.4 Trace of a simple production system.

In executing the production system, the ''attention'' or ''current focus'' of the problem solver would match a production rule, which would change the state of ''attention'' to match another production-encoded skill, and so on.

It is important to note that in this work Newell and Simon used the production system not only as a vehicle for implementing graph search but also as an actual model of human problem-solving behavior. The productions corresponded to the problem-solving skills in the human's *long-term memory*. Like the skills in long-term memory, these productions are not changed by the execution of the system; they are invoked by the ''pattern'' of a particular problem instance, and new skills may be added without requiring ''recoding'' of the previously existing knowledge. The production system's working memory corresponds to *short-term memory* (or current focus of attention) in the human and describes the current stage of solving a problem instance. The contents of working memory are generally not retained after a problem has been solved.)

This research is described in *Human Problem Solving* by Newell and Simon (1972) and in Luger (1978). Newell, Simon, and others have continued to use production rules to model the difference between novices and experts (Larkin et al. 1980; Simon and Simon 1978) in areas such as solving algebra word problems and physics problems. Production systems also form a basis for studying learning in both humans and computers (Klahr et al. 1987).

(Production systems provide a model for encoding human expertise in the form of rules and designing pattern-driven search algorithms, tasks that are central to the design of the rule-based expert system. In expert systems, the production system is not necessarily assumed to actually model human problem-solving behavior; however, the aspects of production systems that make them useful as a potential model of human problem solving (modularity of rules, separation of knowledge and control, separation of working memory and problem-solving knowledge) make them an ideal tool for building expert systems.)

An important family of AI languages comes directly out of the production system language research at Carnegie Mellon. These are the OPS languages; OPS stands for Official Production System. Although their origins are in modeling human problem solving, these languages have proved highly effective for programming expert systems and other AI applications. OPS5, the newest dialect, is the major implementation language for the VAX configurer (XCON) and other expert systems developed at Digital Equipment Corporation (McDermott 1981, 1982).

In the next section we give examples of how the production system may be used to solve a variety of search problems.

4.3.2 Examples of Production Systems

EXAMPLE 4.3.1: THE 8-PUZZLE

(The search space generated by the 8-puzzle, introduced in Chapter 3, is both complex enough to be interesting and small enough to be tractable, so it is frequently used to explore different search strategies, such as depth- and breadth-first search, as well as the heuristic strategies discussed in Chapter 5. It also lends itself to solution using a production system.

Recall that we gain generality by thinking of "moving the blank space" rather than moving a numbered tile. Legal moves are defined by the productions in Figure 4.5. Of course, all four of these moves are applicable only when the blank is in the center; when it is in one of the corners only two moves are possible. If a beginning state and a goal state for the 8-puzzle are now specified, it is possible to make a production system accounting of the problem's search space.)

An actual implementation of this problem might represent each board configuration with a "state" predicate with nine parameters (for nine possible locations of the eight tiles and the blank); rules could be written as implications whose premise performs the required condition check. Alternatively, arrays or list structures could be used for board states.

An example of the space searched in finding a solution for the problem given in Figure 4.5 follows in Figure 4.6. Because this solution path can go very deep if unconstrained, a depth bound has been added to the search. (A simple means for adding a depth bound is to keep track of the length of the current path and to force backtracking if this bound is exceeded.) A depth bound of five is used in the solution of Figure 4.6. Note that the number of possible states of working memory grows exponentially with the depth of the search.)

EXAMPLE 4.3.2: THE KNIGHT'S TOUR PROBLEM

The 3 × 3 knight's tour problem presented in Section 4.2 may be solved using a production system approach. Here each move would be represented as a production whose con-

Start state:

2	8	3
1	6	4
7	▨	5

Goal state:

1	2	3
8	▨	4
7	6	5

Production set:

Condition		Action
goal state in working memory	→	halt
blank is not on the top edge	→	move the blank up
blank is not on the right edge	→	move the blank right
blank is not on the bottom edge	→	move the blank down
blank is not on the left edge	→	move the blank left

Working memory is the present board state and goal state.

Control regime:

1. Try each production in order.
2. Do not allow loops.
3. Stop when goal is found.

Figure 4.5 The 8-puzzle as a production system.

dition is the location of the knight on a particular square and whose action moves the knight to another square. Sixteen productions represent all possible moves of the knight:

RULE #	CONDITION		ACTION
1	knight on square 1	→	move knight to square 8
2	knight on square 1	→	move knight to square 6
3	knight on square 2	→	move knight to square 9
4	knight on square 2	→	move knight to square 7
5	knight on square 3	→	move knight to square 4
6	knight on square 3	→	move knight to square 8
7	knight on square 4	→	move knight to square 9
8	knight on square 4	→	move knight to square 3
9	knight on square 6	→	move knight to square 1
10	knight on square 6	→	move knight to square 7
11	knight on square 7	→	move knight to square 2
12	knight on square 7	→	move knight to square 6
13	knight on square 8	→	move knight to square 3
14	knight on square 8	→	move knight to square 1
15	knight on square 9	→	move knight to square 2
16	knight on square 9	→	move knight to square 4

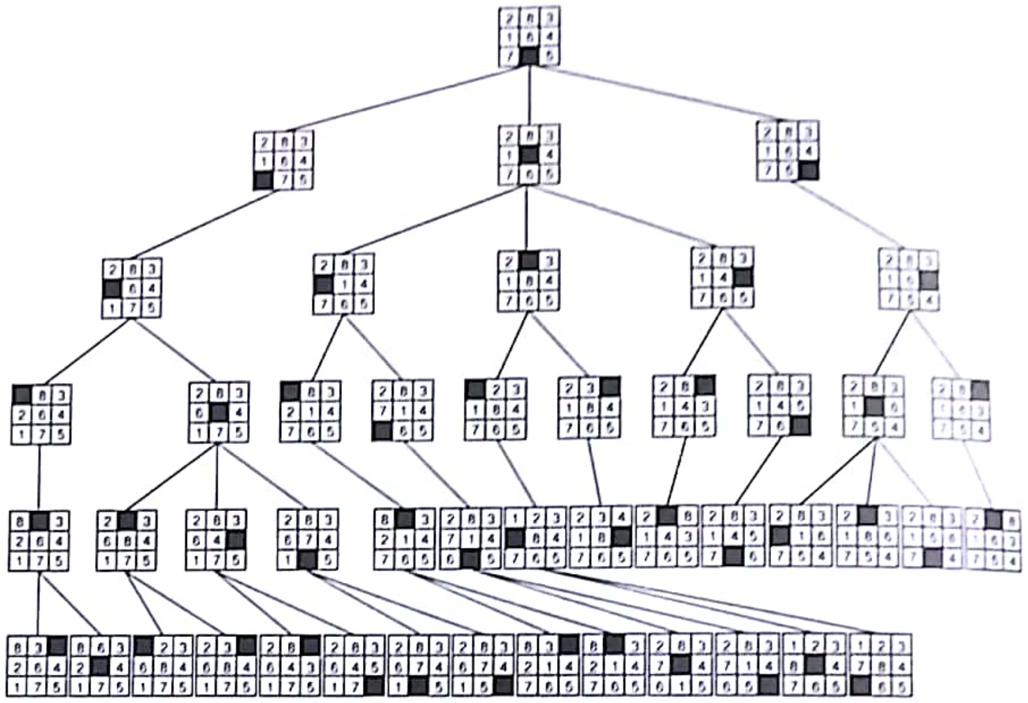

Figure 4.6 State space of the 8-puzzle searched by a production system with loop detection and a depth bound of 5.

Working memory contains both the current board state and the goal state. The control regime applies rules until the current state equals the goal state and then halts. A simple conflict resolution scheme would fire the first rule that did not cause the search to loop. Because the search may lead to dead ends (from which every possible move leads to a previously visited state and, consequently, a loop), the control regime should also allow backtracking. An execution of this production system that determines if there is a path from square 1 to square 2 is charted in Figure 4.7.

It is interesting to note that in implementing the path predicate in the knight's tour example of Section 4.2, we have actually implemented this production system solution! From this point of view, pattern_search is simply an interpreter, with the actual search implemented by the path definition. The productions are the move facts, with the first parameter specifying the condition (the square the piece must be on to make the move) and the second parameter the action (the square to which it can move). The recognize-act cycle is implemented by the recursive path predicate. Working memory contains the current state and the desired goal state and is represented as the parameters of the path predicate. On a given iteration, the conflict set is all of the move expressions that will unify with the goal move(X,Z). This program uses the simple conflict resolution strategy of selecting and firing the first move predicate encountered in the knowledge base that

Iteration #	Working memory		Conflict set (rule #'s)	Fire rule
	Current square	Goal square		
0	1	2	1, 2	1
1	8	2	13, 14	13
2	3	2	5, 6	5
3	4	2	7, 8	7
4	9	2	15, 16	15
5	2	2		Halt

Figure 4.7 Production system solution to the 3 × 3 knight's tour problem.

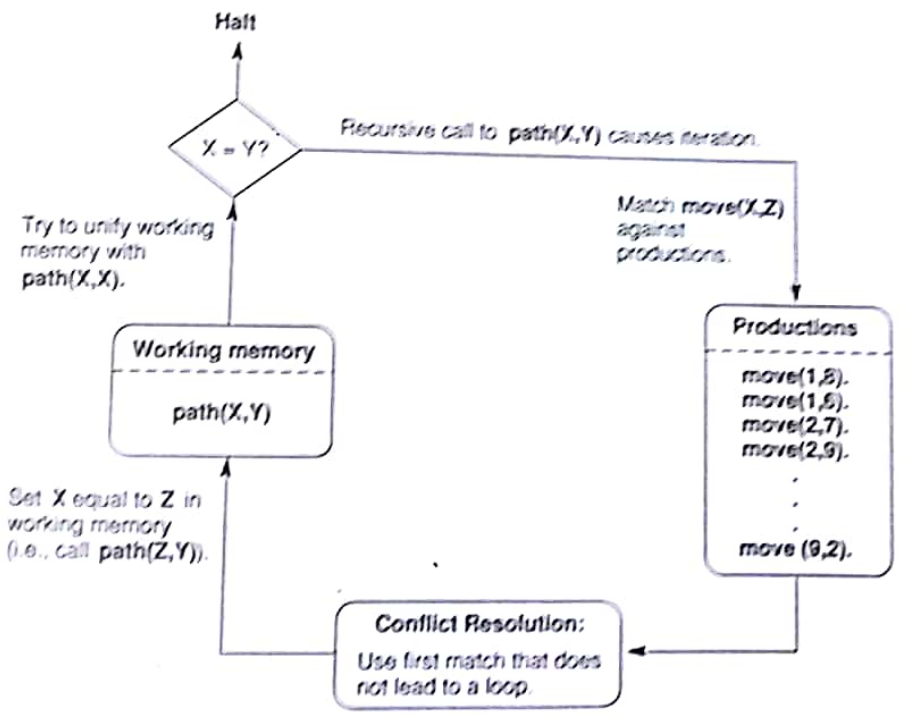
Halt
X = Y?
Recursive call to path(X,Y) causes iteration.
Match move(X,Z) against productions.
Try to unify working memory with path(X,X).
Working memory
path(X,Y)
Productions
move(1,8).
move(1,6).
move(2,7).
move(2,9).
move (9,2).
Set X equal to Z in working memory (i.e., call path(Z,Y)).
Conflict Resolution:
Use first match that does not lead to a loop.

Figure 4.8 Recursive path algorithm as a production system.

does not lead to a repeated state. The controller also backtracks from dead-end states. This characterization of the path definition as a production system is given in Figure 4.8.

Production systems are capable of generating infinite loops when searching a state space graph. These loops are particularly difficult to spot in a production system since the rules can fire in any order. That is, looping may appear in the execution of the system but it cannot easily be found from a syntactic inspection of the rule set. For example, with the "move" rules of the knight's tour problem ordered as in Section 4.2 and a conflict resolution strategy of selecting the first match, the pattern move(2,X) would match with move(2,9), indicating a move to square 9. On the next iteration, the pattern move(9,X) would match with move(9,2), taking the search back to square 2, causing a loop.

To prevent looping, pattern_search checked a global list (closed) of visited states. The actual conflict resolution strategy was therefore: select the first matching move *that leads to an unvisited state*.

In a production system, the proper place for recording such case-specific data as a list of previously visited states is not a global closed list but the working memory itself. We can alter the path predicate to use working memory for loop detection.

Assume that pattern_search does not maintain a global closed list or otherwise perform loop detection. Assume that our predicate calculus language is augmented by the addition of a special construct, assert(X), which causes its argument X to be entered into the working memory. assert is not an ordinary predicate but an action that is performed; hence, it always succeeds.

assert is used to place a "marker" in working memory to indicate when a state has been visited. This marker is represented as a unary predicate, been(X), which takes as its argument a square on the board. been(X) is added to working memory when a new state X is visited. Conflict resolution may then require that been(Z) must not be in working memory before move(X,Z) can fire. For a specific value of Z this can be tested by matching a pattern against working memory.

The modified recursive path definition is written:

$$\forall X\ \text{path}(X,X).$$
$$\forall X,Y\ \text{path}(X,Y) \Leftarrow \exists Z\ \text{move}(X,Z) \wedge \neg(\text{been}(Z)) \wedge \text{assert}(\text{been}(Z)) \wedge \text{path}(Z,Y)$$

In this definition, move(X,Z) succeeds on the first match with a move predicate. This binds a value to Z. If been(Z) matches with an entry in working memory, ¬(been(Z)) will cause a failure (i.e., it will be false). pattern_search will then backtrack and try another match for move(X,Z). If square Z is a new state, the search will continue, with been(Z) asserted to the working memory to prevent future loops. The actual firing of the production takes place when the path algorithm recurs. Thus, the presence of been predicates in working memory implements loop detection in this production system.

Note that although predicate calculus is used as the language for both productions and working memory entries, the procedural nature of production systems requires that the goals be tested in left-to-right order in the path definition. This order of interpretation is provided by pattern_search.

EXAMPLE 4.3.3: THE FULL KNIGHT'S TOUR

We may generalize the knight's tour solution to the full 8 × 8 chessboard. Because it makes little sense to enumerate moves for such a complex problem, we replace the 16 move facts with a set of 8 rules to generate legal knight moves. These moves (productions) correspond to the 8 possible ways a knight can move (Figure 4.1).

If we index the chessboard by row and column numbers, we can define a production rule for moving the knight down two squares and right one square:

```
CONDITION: current row <= 6 ∧ current column number <= 7

ACTION: new row = current row + 2 ∧ new column = current column + 1
```

If we use predicate calculus to represent productions, then a board square could be defined by the predicate square(R,C), representing the Rth row and Cth column of the board. The above rule could be rewritten in predicate calculus as:

```
move(square(Row,Column),square(Newrow,Newcolumn) )⇐
            less_than_or_equals(Row,6) ∧
            equals(Newrow,plus(Row,2)) ∧
            less_than_or_equals(Column,7) ∧
            equals(Newcolumn,plus(Column,1))
```

plus is a function for arithmetic addition; less_than_or_equals and equals have the obvious arithmetic interpretations. Seven addition rules can be designed that similarly compute the remaining possible moves. These eight rules replace the move facts in the 3 × 3 version of the problem.

The path definition from the 3 × 3 example defines the control loop for this problem. As we have seen, when predicate calculus descriptions are interpreted procedurally (such as through the pattern_search algorithm), subtle changes are made to the semantics of predicate calculus. One such change is the sequential fashion in which goals are solved. This imposes an ordering, or *procedural semantics*, on predicate calculus expressions. Another change is the introduction of *meta-logical* predicates such as assert which indicate actions beyond the truth value interpretation of predicate calculus expressions. These issues are discussed in more detail in the PROLOG chapters (6 and 12) and in the LISP implementation of a logic programming engine (Chapter 13).

EXAMPLE 4.3.4: THE FINANCIAL ADVISOR AS A PRODUCTION SYSTEM

In the previous two chapters, we developed a small financial advisor, using predicate calculus to represent the financial knowledge and graph search to make the appropriate inferences in a consultation. The production system provides a natural vehicle for its implementation. The implications of the logical description form the productions. The case-specific information (an individual's salary, dependents, etc.) is loaded into working memory. Rules are enabled when their premises are satisfied. A rule is chosen from this conflict set and fired, adding its conclusion to working memory. This continues until all possible top-level conclusions have been added to the working memory. Indeed, many expert system "shells" are production systems with added features for supporting the user interface, handling uncertainty in the reasoning, editing the knowledge base, tracing execution, etc. (Sections 12.2.1 and 13.4).

4.3.3 Control of Search in Production Systems

The production system model offers a range of opportunities for adding heuristic control to a search algorithm. These include the choice of data- or goal-driven strategies, the structure of the rules themselves, and the choice of strategies for conflict resolution.

Control Through Choice of Data- or Goal-Driven Search Strategy. Data-driven search begins with a problem description (such as a set of logical axioms, symptoms of an illness, or a body of data that needs interpretation) and infers new knowledge from the data. This is done by applying rules of inference, legal moves in a game, or other state-generating operations to the current description of the world and adding the results to that problem description. This process continues until a goal is reached.

This description of data-driven reasoning emphasizes its close fit with the production system model of computation. The "current state of the world" (data that have either been assumed to be true or deduced as true with previous use of production rules) is placed in working memory. The recognize-act cycle then passes it in front of the (ordered) set of productions. When these data match (are unified with) the condition(s) of one of the production rules, the action of the production adds (by modifying working memory) a new piece of information to the current state of the world.

All productions have the form CONDITION ⇒ ACTION. When the CONDITION matches some elements of working memory, its ACTION is performed. If the production rules are formulated as logical implications and the ACTION adds assertions to working memory, then the act of firing a rule corresponds to an application of modus ponens. This creates a new state of the graph.

Figure 4.9 presents a simple data-driven search on a set of productions expressed as propositional calculus implications. The conflict resolution strategy is a simple one of choosing the enabled rule that has fired least recently (or not at all); in the event of ties, the first rule is chosen. Execution halts when a goal is reached. The figure also presents the sequence of rule firings and the stages of working memory in the execution, along with a graph of the space searched.

Although we have treated production systems in a data-driven fashion, they may also be used to characterize goal-driven search. As defined in Chapter 3, goal-driven search begins with a goal and works backward to establish its truth. To implement this in a production system, the goal is placed in working memory and matched against the ACTIONS of the production rules. These ACTIONs are matched (by unification, for example) just as the CONDITIONs of the productions were matched in the data-driven reasoning. All production rules whose conclusions (ACTIONs) match the goal form the conflict set.

When the ACTION of a rule is matched, the CONDITIONs are added to working memory and become the new subgoals (states) of the search. The new states are then matched to the ACTIONs of other production rules. The process continues until a fact is found, usually in the problem's initial description or, as is often the case in expert systems, by directly asking the user for specific information. The search stops when the CONDITIONs of all the productions fired in this backward fashion are found to be true. These CONDITIONs and the chain of rule firings leading to the original goal form a proof of its truth through successive inferences such as modus ponens. See Figure 4.10

Production set:

1. $p \wedge q \rightarrow goal$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow q$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $start \rightarrow v \wedge r \wedge q$

Trace of execution:

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

Space searched by execution:

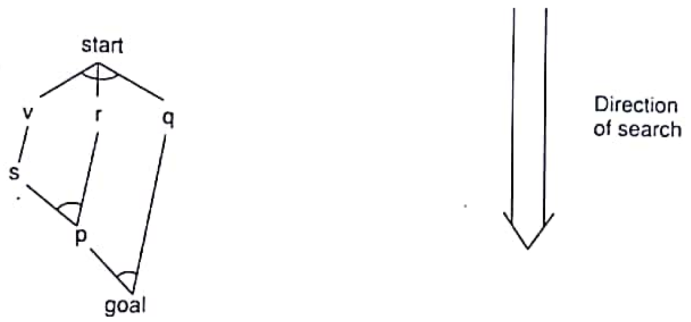

Figure 4.9 Data-driven search in a production system.

for an instance of goal-driven reasoning on the same set of productions used in Figure 4.9. Note that the goal-driven search fires a different series of productions and searches a different space than the data-driven version.

As this discussion illustrates, the production system offers a natural characterization of both goal- and data-driven search. The production rules are the encoded set of inferences (the "knowledge" in a rule-based expert system) for changing state within the graph. When the current state of the world (the set of true statements describing the world) matches the CONDITIONs of the production rules and this match causes the ACTION part of the rule to create another (true) descriptor for the world, it is referred to as data-driven search.

Alternatively, when the goal is matched against the ACTION part of the rules in the production rule set and their CONDITIONs are then set up as subgoals to be shown to be "true" (by matching the ACTIONs of the rules on the next cycle of the production system), the result is goal-driven problem solving.

Since a set of rules may be executed in either a data- or goal-driven fashion, we can compare and contrast the efficiency of each approach in controlling search. The complexity of search for either strategy is measured by such notions as *branching factor* or *penetrance* (Chapter 5). These measures of search complexity can provide a cost estimate for both the data- and goal-driven versions of a problem solver and therefore help in selecting the most effective strategy.

Production set:

1. $p \wedge q \rightarrow goal$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow p$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $start \rightarrow v \wedge r \wedge q$

Trace of execution:

Iteration #	Working memory	Conflict set	Rule fired
0	goal	1	1
1	goal, p, q	1, 2, 3, 4	2
2	goal, p, q, r, s	1, 2, 3, 4, 5	3
3	goal, p, q, r, s, w	1, 2, 3, 4, 5	4
4	goal, p, q, r, s, w, t, u	1, 2, 3, 4, 5	5
5	goal, p, q, r, s, w, t, u, v	1, 2, 3, 4, 5, 6	6
6	goal p, q, r, s, w, t, u, v, start	1, 2, 3, 4, 5, 6	halt

Space searched by execution:

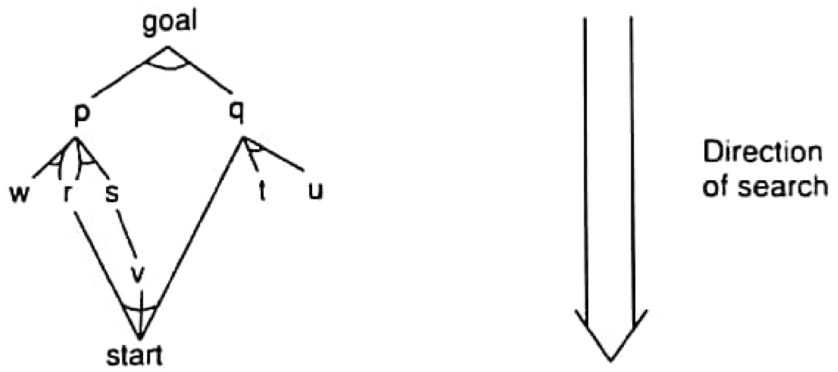

Figure 4.10 Goal-driven search in a production system.

We can also employ combinations of strategies. For example, we can search in a forward direction until the number of states becomes large and then switch to a goal-directed search to use possible subgoals to select among alternative states. The danger in this situation is that, when heuristic or best-first search (Chapter 5) is used, the parts of the graphs actually searched may ''miss'' each other and ultimately require more search than a simpler approach (Fig. 4.11). However, when the branching of a space is constant and exhaustive search is used, a combined search strategy can cut back drastically the amount of space searched. (See Figure 4.12.)

more understandable, etc.

Control of Search Through Conflict Resolution. While production systems (like all architectures for knowledge-based systems) allow heuristics to be encoded in the knowledge content of rules themselves, they offer other opportunities for heuristic control through conflict resolution. Although the simplest such strategy is to choose the first rule that matches the contents of working memory, any strategy may potentially be applied to conflict resolution. For example, conflict resolution strategies supported by OPS5 (Brownston et al. 1985) include:

1. *Refraction.* Refraction specifies that once a rule has fired, it may not fire again until the working memory elements that match its conditions have been modified. This discourages looping.
2. *Recency.* The recency strategy prefers rules whose conditions match with the patterns most recently added to working memory. This focuses the search on a single line of reasoning.
3. *Specificity.* This strategy assumes that a more specific problem-solving rule is preferable to a general rule. A rule is more specific than another if it has more conditions. This implies that it will match fewer potential working memory patterns.

These are representative of the general strategies that may be used in conflict resolution.

4.3.4 Advantages of Production Systems for AI

As illustrated by the preceding examples, the production system offers a general framework for implementing search. Because of its simplicity, modifiability, and flexibility in applying problem solving knowledge, the production system has proved to be an important tool for the construction of expert systems and other AI applications. The major advantages of production systems for artificial intelligence include:

Separation of Knowledge and Control. The production system is an elegant model of separation of knowledge and control in a computer program. Control is provided by the recognize act cycle of the production system loop, and the problem solving knowledge is encoded in the rules themselves. The advantages of this separation include ease of modifying the knowledge base without requiring a change in the code for program control and, conversely, the ability to alter the code for program control without changing the set of production rules.

A Natural Mapping onto State Space Search. The components of a production system map naturally into the constructs of state space search. The successive states of working memory form the nodes of a state space graph. The production rules are the set of possible transitions between states, with conflict resolution implementing the selection of a branch in the state space. These rules simplify the implementation, debugging, and documentation of search algorithms.

Modularity of Production Rules. An important aspect of the production system model is the lack of any syntactic interactions between production rules. Rules may only effect the firing of other rules by changing the pattern in working memory; they may not "call" another rule directly as if it were a subroutine, nor may they set the value of variables in other production rules. The scope of the variables of these rules is confined to the individual rule. This syntactic independence supports the incremental development of expert systems by successively adding, deleting, or changing the knowledge (rules) of the system.

Pattern-Directed Control. The problems addressed by AI programs require particular flexibility in program execution. This goal is served by the fact that the rules in a production system may fire in any sequence. The descriptions of a problem that make up the current state of the world determine the conflict set and, consequently, the particular search path and solution.)

Opportunities for Heuristic Control of Search. These were described in the preceding section.

Tracing and Explanation. The modularity of rules and the iterative nature of their execution make it easier to trace execution of a production system. At each stage of the recognize act cycle, the selected rule may be displayed. Since each rule corresponds to a single "chunk" of problem solving knowledge, the rule content should provide a meaningful explanation of the system's current state and action (Chapter 8). In contrast, a single line of code in a traditional language such as Pascal or FORTRAN is virtually meaningless.

(**Language Independence.** The production system control model is independent of the representation chosen for rules and working memory, as long as that representation supports pattern matching. We described production rules as predicate calculus implications of the form A ⇒ B, where the truth of A and the inference rule modus ponens allow us to conclude B. Although there are many advantages to using logic as both the basis for representation of knowledge and the source of sound inference rules, the production system model may be used with other representations.

Although predicate calculus offers the advantage of logically sound inference, much "real-world" reasoning is not sound in the logical sense. Instead, it involves probabilistic reasoning, use of uncertain evidence, and default assumptions. Later chapters (8, 9, and 11) discuss alternative inference rules that provide these capabilities. Regardless of the type of inference rules employed, however, the production system provides a vehicle for searching the state space.

A Plausible Model of Human Problem Solving. Modeling human problem solving was among the first uses of production systems; they continue to be used as a model for human performance in much cognitive science research (Chapter 15).)