BUILDING CONTROL ALGORITHMS FOR STATE SPACE SEARCH

6

If we carefully factor out the influences of the task environments from the influences of the underlying hardware components and organization, we reveal the true simplicity of the adaptive system. For, as we have seen, we need postulate only a very simple information processing system in order to account for human problem solving in such tasks as chess, logic, and cryptarithmetic. The apparently complex behavior of the information processing system in a given environment is produced by the interaction of the demands of the environment with a few basic parameters of the system, particularly characteristics of its memories.

—A. Newell and H. A. Simon, "Human Problem Solving" (1972)

What we call the beginning is often the end
And to make an end is to make a beginning.
The end is where we start from . . .

T. S. Eliot, "Four Quartets"

6.0 Introduction

To this point, Part II has represented problem solving as search through a set of problem situations or states. Chapter 2 presented the predicate calculus as a medium for describing states of a problem and sound inference as a method for producing new states. Chapter 3 introduced graphs to represent and link problem situations. The backtrack algorithm, as well as algorithms for depth-first and breadth-first search can explore these graphs. Chapter 4 presented algorithms for heuristic search. In Chapter 5, given probabilistic states of the world, stochastic inference was used to produce new states. To summarize, Part II has:

1. Represented a problem solution as a path in a graph from a start state to a goal.

2. Used search to test systematically alternative paths to goals.
3. Employed backtracking, or some other mechanism, to allow algorithms to recover from paths that failed to find a goal.
4. Used lists to keep explicit records of states under consideration.
    a. The *open* list allows the algorithm to explore untried states if necessary.
    b. The *closed* list of visited states allows the algorithm to implement loop detection and avoid repeating fruitless paths.
5. Implemented the *open* list as a *stack* for depth-first search, a *queue* for breadth-first search, and a *priority queue* for best-first search.

Chapter 6 introduces further techniques for building search algorithms. In Section 6.1, *recursive search* implements depth-, breadth-, and best-first search in a more concise and natural fashion than was done in Chapter 3. Further, recursion is augmented with *unification* to search the state space generated by predicate calculus assertions. This *pattern-directed* search algorithm is the basis of PROLOG, see Section 14.3, and several of the expert systems discussed in Chapter 8. In Section 6.2 we introduce *production systems*, a general architecture for pattern-directed problem solving that has been used extensively to model human problem solving, Chapter 16, as well as in other AI applications, including expert systems, Chapter 7. Finally, in Section 6.3, we present another AI problem-solving control architecture, the *blackboard*.

6.1 Recursion-Based Search (optional)

6.1.1 Recursive Search

In mathematics, a recursive definition uses the term being defined as part of its own definition. In computer science, recursion is used to define and analyze both data structures and procedures. A recursive procedure consists of:

1. A recursive step: the procedure calls itself to repeat a sequence of actions.
2. A terminating condition that stops the procedure from recurring endlessly (the recursive version of an endless loop).

Both these components are essential and appear in all recursive definitions and algorithms. Recursion is a natural control construct for data structures that have a regular structure and no definite size, such as lists, trees, and graphs, and is particularly appropriate for state space search.

A direct translation of the depth-first search algorithm of Chapter 3 into recursive form illustrates the equivalence of recursion and iteration. This algorithm uses global variables closed and open to maintain lists of states. Breadth-first and best-first search can be designed with virtually the same algorithm, that is, by retaining closed as a global data structure and by implementing open as a queue or a priority queue rather than as a stack (build stack becomes build queue or build priority queue):

```
function depthsearch;                                   % open & closed global

   begin
      if open is empty
         then return FAIL;
      current_state := the first element of open;
      if current_state is a goal state
         then return SUCCESS
         else
            begin
               open := the tail of open;
               closed := closed with current_state added;
               for each child of current_state
                  if not on closed or open                    % build stack
                     then add the child to the front of open
            end;
      depthsearch                                                   % recur
   end.
```

Depth-first search as just presented does not utilize the full power of recursion. It is possible to simplify the procedure further by using recursion itself (rather than an explicit open list) to organize states and paths through the state space. In this version of the algorithm, a global closed list is used to detect duplicate states and prevent loops, and the open list is implict in the activation records of the recursive environment. Since the open list can no longer be explicitly manipulated, breadth-first and best-first search are no longer natural extensions of the following algorithm:

```
function depthsearch (current_state);                    % closed is global

begin
   if current_state is a goal
      then return SUCCESS;
   add current_state to closed;
   while current_state has unexamined children
      begin
         child := next unexamined child;
         if child not member of closed
            then if depthsearch(child) = SUCCESS
               then return SUCCESS
      end;
   return FAIL                                        % search exhausted
end
```

Rather than generating all children of a state and placing them on an open list, this algorithm produces the child states one at a time and recursively searches the descendants of each child before generating its sibling. Note that the algorithm assumes an order to the state generation operators. In recursively searching a child state, if some descendant of that state is a goal, the recursive call returns success and the algorithm ignores the siblings.

If the recursive call on the child state fails to find a goal, the next sibling is generated and all of its descendants are searched. In this fashion, the algorithm searches the entire graph in a depth-first order. The reader should verify that it actually searches the graph in the same order as the depth-first search algorithm of Section 3.2.3.

The omission of an explicit open list is made possible through recursion. The mechanisms by which a programming language implements recursion include a separate *activation record* (Aho and Ullman 1977) for each recursive call. Each activation record captures the local variables and state of execution of each procedure call. When the procedure is called recursively with a new state, a new activation record stores its parameters (the state), any local variables, and the current state of execution. In a recursive search algorithm, the series of states on the current path are recorded in the sequence of activation records of the recursive calls. The record of each call also indicates the last operation used to generate a child state; this allows the next sibling to be generated when needed.

Backtracking is effected when all descendants of a state fail to include a goal, causing the recursive call to fail. This returns fail to the procedure expanding the parent state, which then generates and recurs on the next sibling. In this situation, the internal mechanisms of recursion do the work of the open list used in the iterative version of the algorithm. The recursive implementation allows the programmer to restrict his or her point of view to a single state and its children rather than having to explicitly maintain an open list of states. The ability of recursion to express global concepts in a closed form is a major source of its power.

As these two algorithms demonstrate, state space search is an inherently recursive process. To find a path from a current state to a goal, move to a child state and recur. If that child state does not lead to a goal, try its siblings in order. Recursion breaks a large and difficult problem (searching the whole space) into smaller, simpler pieces (generate the children of a single state) and applies this strategy (recursively) to each of them. This process continues until a goal state is discovered or the space is exhausted.

In the next section, this recursive approach to problem solving is extended into a controller for a logic-based problem solver that uses unification and inference to generate and search a space of logical relations. The algorithm supports the and of multiple goals as well as back chaining from a goal to premises.

6.1.2 A Recursive Search Example: Pattern-Driven Reasoning

In Section 6.1.2 we apply recursive search to a space of logical inferences; the result is a general search procedure for predicate calculus based problem specifications.

Suppose we want to write an algorithm that determines whether a predicate calculus expression is a logical consequence of some set of assertions. This suggests a goal-directed search with the initial query forming the goal, and modus ponens defining the transitions between states. Given a goal (such as p(a)), the algorithm uses unification to select the implications whose conclusions match the goal (e.g., $q(X) \rightarrow p(X)$). Because the algorithm treats implications as potential rules for solving the query, they are often simply called *rules*. After unifying the goal with the conclusion of the implication (or rule) and applying the resulting substitutions throughout the rule, the rule premise becomes a new goal (q(a)). This is called a *subgoal*. The algorithm then recurs on the subgoal. If a subgoal matches a fact in the knowledge base, search terminates. The series of inferences that led from the initial goal to the given facts prove the truth of the original goal.

```
function pattern_search (current_goal);

begin
  if current_goal is a member of closed                              % test for loops
    then return FAIL
    else add current_goal to closed;
  while there remain in data base unifying facts or rules do
    begin
      case
        current_goal unifies with a fact:
          return SUCCESS;
        current_goal is a conjunction (p ∧ ...):
          begin
            for each conjunct do
              call pattern_search on conjunct;
            if pattern_search succeeds for all conjuncts
              then return SUCCESS
              else return FAIL
          end;
        current_goal unifies with rule conclusion (p in q → p):
          begin
            apply goal unifying substitutions to premise (q);
            call pattern_search on premise;
            if pattern_search succeeds
              then return SUCCESS
              else return FAIL
          end;
      end;                                                            % end case
    end;
    return FAIL
  end.
```

In the function pattern_search, search is performed by a modified version of the recursive search algorithm that uses unification, Section 2.3.2, to determine when two expressions match and modus ponens to generate the children of states. The current focus of the search is represented by the variable current_goal. If current_goal matches with a fact, the algorithm returns success. Otherwise the algorithm attempts to match current_goal with the conclusion of some rule, recursively attempting to solve the premise. If current_goal does not match any of the given assertions, the algorithm returns fail. This algorithm also handles conjunctive goals.

For simplicity, the algorithm does not address the problem of maintaining consistency among the variable substitutions produced by unification. This is important when solving conjunctive queries with shared variables (as in $p(X) \wedge q(X)$). Both conjuncts succeed, but they must also succeed with the same unifiable binding for X, see Section 2.3.2.

The major advantage of using general methods such as unification and modus ponens to generate states is that the resulting algorithm may search *any* space of logical inferences where the specifics of a problem are described using predicate calculus assertions. Thus, we have a means of separating problem-solving knowledge from its control and implementation on the computer. pattern_search provides our first example of the separation of problem knowledge from control of search.

Although the initial version of pattern_search defined the behavior of a search algorithm for predicate calculus expressions, several subtleties must still be addressed. These include the order with which the algorithm tries alternative matches and proper handling of the full set of logical operators ($\wedge$, $\vee$, and $\neg$). Logic is declarative, and without a prescribed search strategy: it defines a space of possible inferences but does not tell a problem solver how to make the useful ones.

To reason with predicate calculus, we need a control regime that systematically searches the space, avoiding meaningless paths and loops. A control algorithm such as pattern_search must try alternative matches in some sequential order. Knowing this order allows the program designer to control search by properly ordering rules in the knowledge base. A simple way to define such an order is to require that the algorithm try rules and facts in the order that they appear in the knowledge base.

A second issue is the existence of logical connectives in the rule premises: e.g., implications of the form "$p \leftarrow q \wedge r$" or "$p \leftarrow q \vee (r \wedge s)$." As will be recalled from the discussion of and/or graphs, an $\wedge$ operator indicates that both expressions must be shown to be true for the entire premise to be true. In addition, the conjuncts of the expression must be solved with consistent variable bindings. Thus, to solve $p(X) \wedge q(X)$, it is not sufficient to solve $p(X)$ with the substitution $\{a/X\}$ and $q(X)$ with the substitution $\{b/X\}$. Both must be solved with the same unifiable binding for X. An or operator, on the other hand, indicates that either expression must be found to be true. The search algorithm must take this into account.

The last addition to the algorithm is the ability to solve goals involving logical negation ($\neg$). pattern_search handles negated goals by first solving the operand of the $\neg$. If this subgoal succeeds, then pattern_search returns fail. If the operand fails, then pattern_search returns an empty substitution set, indicating success. Note that even though a subgoal may contain variables, the result of solving its negation may not contain any substitutions. This is because $\neg$ can succeed only if its operand *fails*; hence, it cannot return any bindings for the operand.

Finally, the algorithm should not return success but should return the bindings involved in the solution. The complete version of pattern_search, which returns the set of unifications that satisfies each subgoal, is:

```
function pattern_search(current_goal);

begin
  if current_goal is a member of closed                                % test for loops
    then return FAIL
    else add current_goal to closed;
  while there remain unifying facts or rules do
```

```
begin
   case

      current_goal unifies with a fact:
         return unifying substitutions;
      current_goal is negated (¬ p):
         begin
            call pattern_search on p;
            if pattern_search returns FAIL
               then return {};                                          % negation is true
               else return FAIL;
         end;

      current_goal is a conjunction (p ∧ …):
         begin
            for each conjunct do
               begin
                  call pattern_search on conjunct;
                  if pattern_search returns FAIL
                     then return FAIL;
                     else apply substitutions to other conjuncts;
               end;
            if pattern_search returns SUCCESS for all conjuncts
               then return composition of unifications;
               else return FAIL;
         end;

      current_goal is a disjunction (p ∨ …):
         begin
            repeat for each disjunct
               call pattern_search on disjunct
            until no more disjuncts or SUCCESS;
            if pattern_search returns SUCCESS
               then return substitutions
               else return FAIL;
         end;

      current_goal unifies with rule conclusion (p in p ← q):
         begin
            apply goal unifying substitutions to premise (q);
            call pattern_search on premise;
            if pattern_search returns SUCCESS
               then return composition of p and q substitutions
               else return FAIL;
         end;
      end;                                                              %end case
   end                                                                  %end while
   return FAIL
end.
```

This pattern_search algorithm for searching a space of predicate calculus rules and facts is the basis of Prolog (where the *Horn clause* form of predicates is used, Section 14.3) and in many goal-directed expert system shells (Chapter 8). An alternative control structure for pattern-directed search is provided by the *production system*, discussed in the next section.

6.2 Production Systems

6.2.1 Definition and History

The *production system* is a model of computation that has proved particularly important in AI, both for implementing search algorithms and for modeling human problem solving. A production system provides pattern-directed control of a problem-solving process and consists of a set of *production rules*, a *working memory*, and a *recognize–act* control cycle.

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(s).

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(s) 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 simple: *working memory* is initialized with the beginning problem description. The current state of the problem-solving is maintained as a set of patterns in working memory. These patterns are matched against the conditions of the production rules; this produces a subset of the production rules, 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*. To fire a rule, its *action* is performed, changing the contents of working memory. After the selected production is fired, the control cycle repeats with the modified working memory. The process terminates when the contents of working memory do not match any rule's conditions.

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. Most 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 6.1.

A very simple example of production system execution appears in Figure 6.2. 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. 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 Example 3.3.6. It is also closely related to the approach taken by Markov algorithms (Markov 1954) and, like them, is equivalent in power to a Turing machine.

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. 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

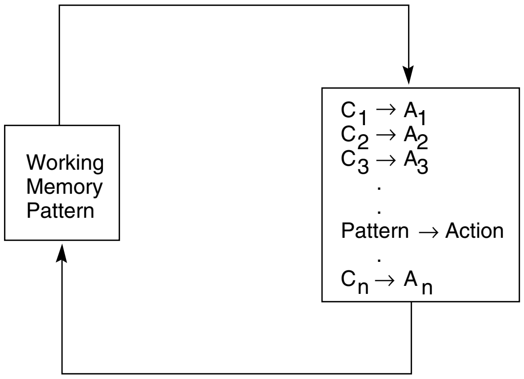

Figure 6.1 A production system. Control loops until working memory pattern no longer matches the conditions of any productions.

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	2, 3	2
3	acbac	1, 3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	Ø	Halt

Figure 6.2 Trace of a simple production system.

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.

These origins of the production system technology are further described in *Human Problem Solving* by Newell and Simon (1972) and in Luger (1978, 1994). 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); ACT* (Anderson 1983b) and SOAR (Newell 1990) build on this tradition.

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 designing and building expert systems as we see in Sections 8.1 and 8.2.

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 for other AI applications. OPS5 was the implementation language for the VAX configurer XCON and other early expert systems developed at Digital Equipment Corporation (McDermott 1981, 1982; Soloway et al. 1987; Barker and O'Connor 1989). OPS interpreters are widely available for PCs and workstations. CLIPS, implemented in the C programming language, is a widely used, object-oriented version of a production system built by NASA. JESS, a production system implemented in Java, was created by Sandia National Laboratories.

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

6.2.2 Examples of Production Systems

EXAMPLE 6.2.1: THE 8-PUZZLE, REVISITED

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-first and breadth-first search, as well as the heuristic strategies of Chapter 4. We now present a production system solution.

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 6.3. Of course, all four of these moves are applicable only when the blank is in the center; when it

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	$\rightarrow$	halt
blank is not on the left edge	$\rightarrow$	move the blank left
blank is not on the top edge	$\rightarrow$	move the blank up
blank is not on the right edge	$\rightarrow$	move the blank right
blank is not on the bottomedge	$\rightarrow$	move the blank down

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 6.3 The 8-puzzle as a production system.

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, taken from Nilsson (1980), of the space searched in finding a solution for the problem given in Figure 6.3 follows in Figure 6.4. 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/depth of the current path and to force backtracking if this bound is exceeded.) A depth bound of 5 is used in the solution of Figure 6.4. Note that the number of possible states of working memory grows exponentially with the depth of the search.

EXAMPLE 6.2.2: THE KNIGHT'S TOUR PROBLEM

In the game of chess, a knight can move two squares either horizontally or vertically followed by one square in an orthogonal direction as long as it does not move off the board. There are thus at most eight possible moves that the knight may make (Figure 6.5).

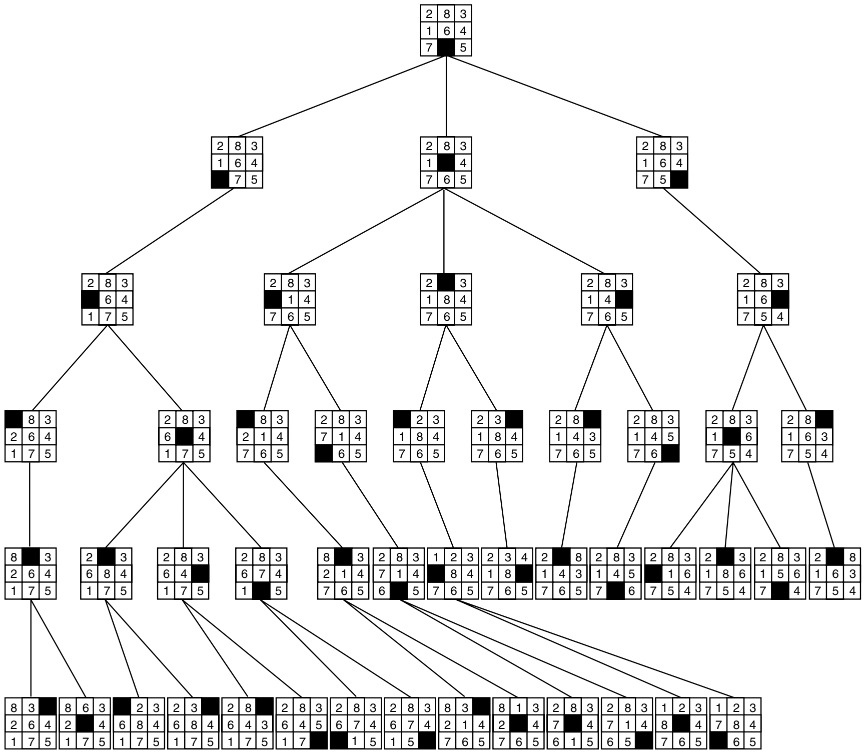

Figure 6.4 The 8-puzzle searched by a production system with loop detection and depth bound 5, from Nilsson (1971).

As traditionally defined, the knight's tour problem attempts to find a series of legal moves in which the knight lands on each square of the chessboard exactly once. This problem has been a mainstay in the development and presentation of search algorithms. The example we use in this chapter is a simplified version of the knight's tour problem. It asks whether there is a series of legal moves that will take the knight from one square to another on a reduced-size (3 × 3) chessboard.

Figure 6.6 shows a 3 × 3 chessboard with each square labeled with integers 1 to 9. This labeling scheme is used instead of the more general approach of giving each space a row and column number in order to further simplify the example. Because of the reduced size of the problem, we simply enumerate the alternative moves rather than develop a general move operator. The legal moves on the board are then described in predicate calculus using a predicate called move, whose parameters are the starting and ending squares of a legal move. For example, move(1,8) takes the knight from the upper left-hand corner to the middle of the bottom row. The predicates of Figure 6.6 enumerate all possible moves for the 3 × 3 chessboard.

The 3 × 3 knight's tour problem may be solved with a production system. Each move can be represented as a rule whose condition is the location of the knight on a particular

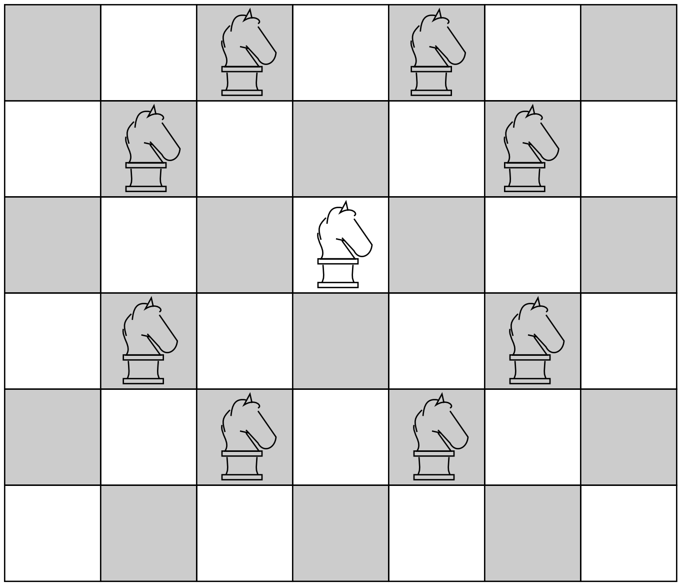

Figure 6.5 Legal moves of a chess knight.

move(1,8)	move(6,1)
move(1,6)	move(6,7)
move(2,9)	move(7,2)
move(2,7)	move(7,6)
move(3,4)	move(8,3)
move(3,8)	move(8,1)
move(4,9)	move(9,2)
move(4,3)	move(9,4)

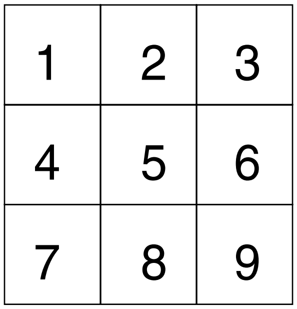

Figure 6.6 A 3 × 3 chessboard with move rules for the simplified knight tour problem.

square and whose action moves the knight to another square. Sixteen productions, presented in Table 6.1, represent all possible moves of the knight.

We next specify a recursive procedure to implement a control algorithm for the production system. Because path(X,X) will unify only with predicates such as path(3,3) or path(5,5), it defines the desired terminating condition. If path(X,X) does not succeed we look at the production rules for a possible next state and then recur. The general recursive path definition is then given by two predicate calculus formulas:

$\forall$ X path(X,X)
$\forall$ X,Y [path(X,Y) $\leftarrow \exists$ Z [move(X,Z) $\wedge$ path(Z,Y)

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

Table 6.1. Production rules for the 3x3 knight problem.

. Working memory, the parameters of the recursive path predicate, 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 thus a loop), the control regime must also allow backtracking; an execution of this production system that determines whether a path exists from square 1 to square 2 is charted in Figure 6.7. This characterization of the path definition as a production system is given in Figure 6.8.

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 because 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 Table 6.1 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*

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 6.7 A production system solution to the 3×3 knight s tour problem.

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 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.

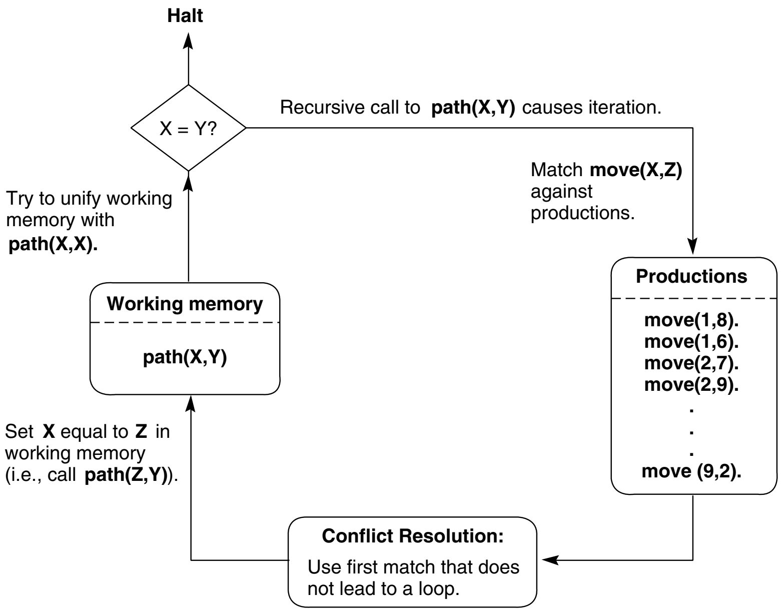
Halt
X = Y?
Recursive call to path(X,Y) causes iteration.
Match move(X,Z) against productions.
Try to unify working memory with path(X,X).
Productions
move(1,8).
move(1,6).
move(2,7).
move(2,9).
move (9,2).
Working memory
path(X,Y)
Set X equal to Z in working memory (i.e., call path(Z,Y)).
Conflict Resolution: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

Figure 6.8 The recursive path algorithm as production system.

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 controller for the production system is:

$\forall$ X path(X,X)
$\forall$ X,Y [path(X,Y) $\leftarrow \exists$ Z [move(X,Z) $\wedge \neg$ (been(Z)) $\wedge$ assert(been(Z)) $\wedge$ 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, $\neg$(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.

EXAMPLE 6.2.3: 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 6.5).

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 $\leq$ 6 $\wedge$ current column $\leq$ 7
ACTION: new row = current row + 2 $\wedge$ 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 addition; less_than_or_equals and equals have the obvious arithmetic interpretations. Seven additional rules can be designed that compute the remaining possible moves. These rules replace the move facts in the 3 × 3 version of the problem.

The path definition from the 3 × 3 knight's example offers a control loop for this problem as well. 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 the interpretation of 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.

EXAMPLE 6.2.4: THE FINANCIAL ADVISOR AS A PRODUCTION SYSTEM

In Chapters 2 and 3, 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", including JESS and CLIPS, are production systems with added features for supporting the user interface, handling uncertainty in the reasoning, editing the knowledge base, and tracing execution.

6.2.3 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-driven or goal-driven search strategies, the structure of the rules themselves, and the choice of different options for conflict resolution.

Control through Choice of Data-Driven 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 state 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 been either assumed to be true or deduced as true with previous use of production rules) is placed in working memory. The recognize–act cycle then matches the current state against 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 $\rightarrow$ ACTION. When the CONDITION matches some elements of working memory, its ACTION is performed. If the production

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:

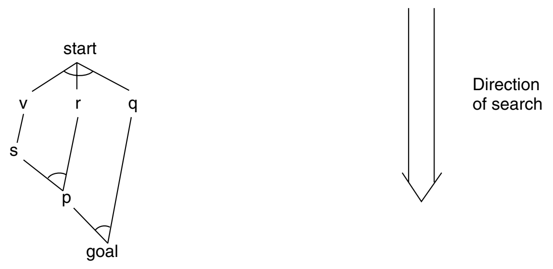

Figure 6.9 Data-driven search in a production system.

rules are formulated as logical implications and the ACTION adds assertions to working memory, then the act of firing a rule can correspond to an application of the inference rule modus ponens. This creates a new state of the graph.

Figure 6.9 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.

To this point we have treated production systems in a data-driven fashion; however they may also be used to produce a goal-driven search. As defined in Chapter 3, goal-driven search begins with a goal and works backward to the facts of the problem to satisfy that goal. 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 facts are 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 CONDITIONs supporting the productions fired in this backward fashion that link to a goal are found to

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:

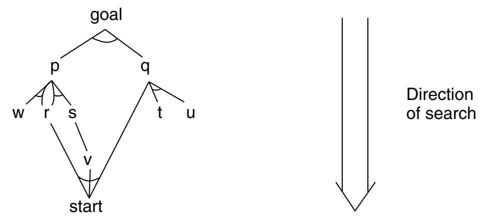

Figure 6.10 Goal-driven search in a production system.

be true. The CONDITIONs and the chain of rule firings leading to the goal form a proof of its truth through successive inferences such as modus ponens. See Figure 6.10 for an instance of goal-driven reasoning on the same set of productions used in Figure 6.9. 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-driven 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.

Because a set of rules may be executed in either a data-driven 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* (Section 4.5). These measures of search complexity can provide a cost estimate for both the data-driven and goal-driven versions of a problem solver and therefore help in selecting the most effective strategy.

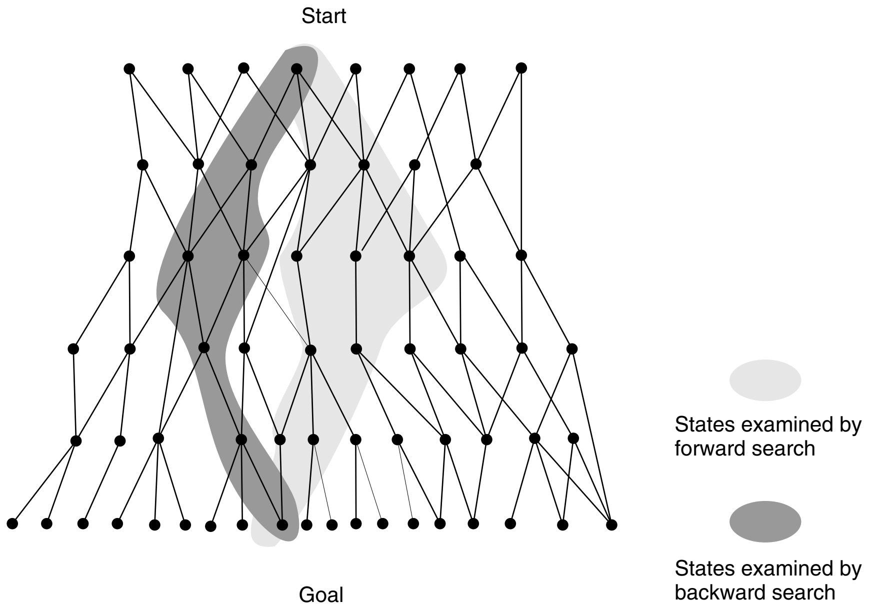
Start
Goal
States examined by forward search
States examined by backward search

Figure 6.11 Bidirectional search missing in both directions, resulting in excessive search.

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 4) is used, the parts of the graphs actually searched may "miss" each other and ultimately require more search than a simpler approach, as in Figure 6.11. 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, as is seen in Figure 6.12.

Control of Search through Rule Structure

The structure of rules in a production system, including the distinction between the condition and the action and the order in which conditions are tried, determines the fashion in which the space is searched. In introducing predicate calculus as a representation language, we emphasized the *declarative* nature of its semantics. That is, predicate calculus expressions simply define true relationships in a problem domain and make no assertion about their order of interpretation. Thus, an individual rule might be $\forall$ X (foo(X) $\wedge$ goo(X) $\rightarrow$ moo(X)). Under the rules of predicate calculus, an alternative form of the same rule is $\forall$ X (foo(X) $\rightarrow$ moo(X) $\vee$ $\neg$ goo(X)). The equivalence of these two clauses is left as an exercise (Chapter 2).

Although these formulations are logically equivalent, they do not lead to the same results when interpreted as productions because the production system imposes an order

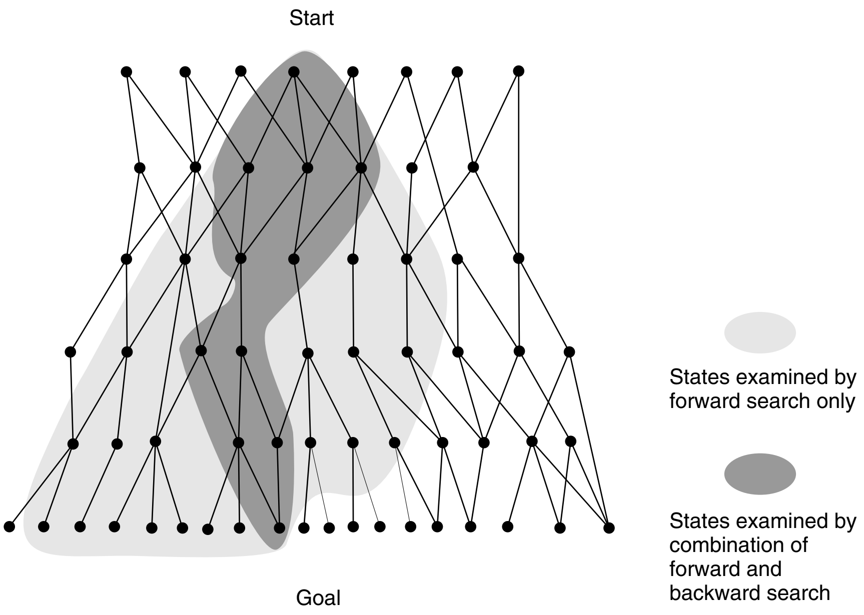
Start
Goal
States examined by forward search only
States examined by combination of forward and backward search

Figure 6.12 Bidirectional search meeting in the middle, eliminating much of the space examined by unidirectional search.

on the matching and firing of rules. Thus, the specific form of the rules determines the ease (or possibility) of matching a rule against a problem instance. This is a result of differences in the way in which the production system *interprets* the rules. The production system imposes a *procedural semantics* on the declarative language used to form the rules.

Because the production system tries each of its rules in a specific order, the programmer may control search through the structure and order of rules in the production set. Although logically equivalent, $\forall$ X (foo(X) $\wedge$ goo(X) $\rightarrow$ moo(X)) and $\forall$ X (foo(X) $\rightarrow$ moo(X) $\vee \neg$ goo(X)) do not have the same behavior in a search implementation.

Human experts encode crucial heuristics within their rules of expertise. The order of premises encodes often critical procedural information for solving the problem. It is important that this form be preserved in building a program that “solves problems like the expert”. When a mechanic says, “If the engine won’t turn over and the lights don’t come on, then check the battery”, she is suggesting a specific sequence of actions. This information is not captured by the logically equivalent statement “the engine turns over or the lights come on or check the battery.” This form of the rules is critical in controlling search, making the system behave logically and making order of rule firings more understandable.

Control of Search through Conflict Resolution

Though 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 it is appropriate to use a more specific problem-solving rule rather than to use a more general one. One rule is more specific than another if it has more conditions, which implies that it will match fewer working memory patterns.

6.2.4 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 often 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. (Several techniques for heuristic control 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 can be displayed. Because each rule corresponds to a single "chunk" of problem-solving knowledge, the rule content can provide a meaningful explanation of the system's current state and action. Furthermore, the chain of rules used within a solution process reflects both a path in the graph as well as a human expert's "line of reasoning", as we see in detail in Chapter 8. In contrast, a single line of code or procedure in a traditional applicative language such as Pascal, FORTRAN, or Java 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 $A \Rightarrow B$, 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, e.g., CLIPS and JESS.

Although predicate calculus offers the advantage of logically sound inference, many problems require reasoning that is not sound in the logical sense. Instead, they involve probabilistic reasoning, use of uncertain evidence, and default assumptions. Later chapters (7, 8, and 9) 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, see Newell and Simon (1972). They continue to be used as a model for human performance in many areas of cognitive science research (Chapter 16).

Pattern-directed search gives us the ability to explore the space of logical inferences in the predicate calculus. Many problems build on this technique by using predicate calculus to model specific aspects of the world such as time and change. In the next section *blackboard systems* are presented as a variation of the production system methodology, where task specific groups of production rules are combined into *knowledge sources* and cooperate in problem solving by communication through a global working memory or *blackboard*.

6.3 The Blackboard Architecture for Problem Solving

The *blackboard* is the final control mechanism presented in this chapter. When we want to examine the states in a space of inferences in a very deterministic fashion, production systems provide great flexibility by allowing us to represent multiple partial solutions simultaneously in working memory and to select the next state through conflict resolution. Blackboards extend production systems by allowing us to organize production memory into separate modules, each of which corresponds to a different subset of the production rules. Blackboards integrate these separate sets of production rules and coordinate the actions of these multiple problem solving agents, sometimes called *knowledge sources*, within a single global structure, the *blackboard*.

Many problems require the coordination of a number of different types of agents. For example, a speech understanding program may have to first manipulate an utterance represented as a digitized waveform. As the decoding process continues, it must find words in this utterance, form these into phrases and sentences, and finally produce a semantic representation of the utterance's meaning.

A related problem occurs when multiple processes must cooperate to solve a single problem. An example of this is the sensor fusion problem (Lesser and Corkill 1983). Assume that we have a network of sensors, each of which is monitored by a separate process. Assume also that the processes can communicate and that proper interpretation of each sensor's data depends on the data received by other sensors in the network. This problem arises in situations as diverse as tracking airplanes across multiple radar sites to combining the readings of multiple sensors in a manufacturing process.

The *blackboard architecture* is a model of control that has been applied to these and other problems requiring the coordination of multiple processes or knowledge sources. A *blackboard* is a central global data base for the communication of independent asynchronous knowledge sources focusing on related aspects of a particular problem. Figure 6.13 gives a schematic of the blackboard design.

In Figure 6.13 each *knowledge source* KS_i gets its data from the blackboard, processes the data, and returns its results to the blackboard to be used by the other knowledge sources. Each KS_i is independent in that it is a separate process operating according to its own specifications and, when a multiprocessing or multiprocessor system is used, it is independent of the other processing in the problem. It is an asynchronous system in that each KS_i begins its operation whenever it finds appropriate input data posted on the blackboard. When it finishes its processing it posts its results to the blackboard and awaits new appropriate input data.

The blackboard approach to organizing a large program was first presented in the HEARSAY-II research (Erman et al. 1980, Reddy 1976). HEARSAY-II was a speech understanding program; it was initially designed as the front end for a library database of

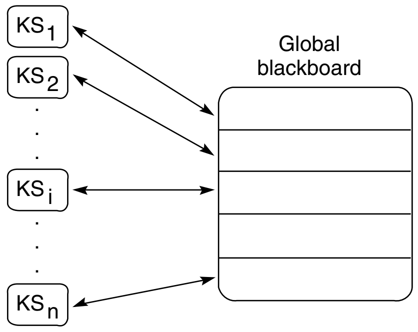

Figure 6.13 Blackboard architecture.

computer science articles. The user of the library would address the computer in spoken English with queries such as, “Are any by Feigenbaum and Feldman?” and the computer would answer the question with information from the library database. Speech understanding requires that we integrate a number of different processes, all of which require very different knowledge and algorithms, all of which can be exponentially complex. Signal processing; recognition of phonemes, syllables, and words; syntactic parsing; and semantic analysis mutually constrain each other in interpreting speech.

The blackboard architecture allowed HEARSAY-II to coordinate the several different knowledge sources required for this complex task. The blackboard is usually organized along two dimensions. With HEARSAY-II these dimensions were *time* as the speech act was produced and the *analysis level* of the utterance. Each level of analysis was processed by a different class of knowledge sources. These analysis levels were:

KS_1	The waveform of the acoustic signal.
KS_2	The phonemes or possible sound segments of the acoustic signal.
KS_3	The syllables that the phonemes could produce.
KS_4	The possible words as analyzed by one KS.
KS_5	The possible words as analyzed by a second KS (usually considering words from different parts of the data).
KS_6	A KS to try to generate possible word sequences.
KS_7	A KS that puts word sequences into possible phrases.

We can visualize these processes as components of Figure 6.13. In processing spoken speech, the waveform of the spoken signal is entered at the lowest level. Knowledge sources for processing this entry are enabled and post their interpretations to the blackboard, to be picked up by the appropriate process. Because of the ambiguities of spoken language, multiple competing hypotheses may be present at each level of the blackboard. The knowledge sources at the higher levels attempt to disambiguate these competing hypotheses.

The analysis of HEARSAY-II should not be seen as simply one lower level producing data that the higher levels can then analyze. It is much more complex than that. If a KS at one level cannot process (make sense of) the data sent to it, that KS can request the KS that sent it the data to go back for another try or to make another hypothesis about the data. Furthermore, different KSs can be working on different parts of the utterance at the same time. All the processes, as mentioned previously, are asynchronous and data-driven; they act when they have input data, continue acting until they have finished their task, and then post their results and wait for their next task.

One of the KSs, called the *scheduler*, handles the "consume-data post-result" communication between the KSs. This scheduler has ratings on the results of each KS's activity and is able to supply, by means of a priority queue, some direction in the problem solving. If no KS is active, the scheduler determines that the task is finished and shuts down.

When the HEARSAY program had a database of about 1,000 words it worked quite well, although a bit slowly. When the database was further extended, the data for the knowledge sources got more complex than they could handle. HEARSAY-III (Balzer et al. 1980, Erman et al. 1981) is a generalization of the approach taken by HEARSAY-II. The time dimension of HEARSAY-II is no longer needed, but the multiple KSs for levels of analysis are retained. The blackboard for HEARSAY-III is intended to interact with a general-purpose relational database system. Indeed, HEARSAY-III is a shell for the design of expert systems; see Section 8.1.1.

An important change in HEARSAY-III has been to split off the scheduler KS (as described above for HEARSAY-II) and to make it a separate blackboard controller for the first (or problem domain) blackboard. This second blackboard allows the scheduling process to be broken down, just as the domain of the problem is broken down, into separate KSs concerned with different aspects of the solution procedure (for example, when and how to apply the domain knowledge). The second blackboard can thus compare and balance different solutions for each problem (Nii and Aiello 1979, Nii 1986*a*, 1986*b*). An alternative model of the blackboard retains important parts of the knowledge base in the blackboard, rather than distributing them across knowledge sources (Skinner and Luger 1991, 1992).

6.4 Epilogue and References

Chapter 6 discussed the implementation of the search strategies of Chapters 3 and 4. Thus, the references listed in the epilogue to those chapters are also appropriate to this chapter. Section 6.1 presented recursion as an important tool for programming graph search, implementing the depth-first and backtrack algorithms of Chapter 3 in recursive form. Pattern-directed search with unification and inference rules, as in Chapter 2, simplifies the implementation of search through a space of logical inferences.

In Section 6.2 the production system was shown as a natural architecture for modeling problem solving and implementing search algorithms. The section concluded with examples of production system implementations of data-driven and goal-driven search. In fact, the production system has always been an important paradigm for AI programming, beginning with work by Newell and Simon and their colleagues at Carnegie Mellon University (Newell and Simon 1976, Klahr et al. 1987, Neches et al. 1987, Newell et al. 1989). The production system has also been an important architecture supporting research in cognitive science (Newell and Simon 1972, Luger 1994; see also Chapter 16).

References on building production systems, especially the OPS languages, include *Programming Expert Systems in OPS5* by Lee Brownston et al. (1985), and *Pattern Directed Inference Systems* by Donald Waterman and Frederick Hayes-Roth (1978). For modern production systems in C and Java, go to the web sites for CLIPS and JESS.

Early work in blackboard models is described in HEARSAY-II research (Reddy 1976, Erman et al. 1980). Later developments in blackboards is described in the HEARSAY-III work (Lesser and Corkill 1983, Nii 1986a, Nii 1986b), and *Blackboard Systems*, edited by Robert Engelmore and Tony Morgan (1988).

Research in production systems, planning, and blackboard architectures remains an active part of artificial intelligence. We recommend that the interested reader consult recent proceedings of the American Association for Artificial Intelligence Conference and the International Joint Conference on Artificial Intelligence. Morgan Kaufmann and AAAI Press publish conference proceedings, as well as collections of readings on AI topics.

6.5 Exercises

1. a. Write a member-check algorithm to recursively determine whether a given element is a member of a list.
   b. Write an algorithm to count the number of elements in a list.
   c. Write an algorithm to count the number of atoms in a list.
   (The distinction between atoms and elements is that an element may itself be a list.)

2. Write a recursive algorithm (using open and closed lists) to implement breadth-first search. Does recursion allow the omission of the open list when implementing breadth-first search? Explain.

3. Trace the execution of the recursive depth-first search algorithm (the version that does not use an open list) on the state space of Figure 3.14.

4. In an ancient Hindu tea ceremony, there are three participants: an elder, a servant, and a child. The four tasks they perform are feeding the fire, serving cakes, pouring tea, and reading poetry; this order reflects the decreasing importance of the tasks. At the beginning of the ceremony, the child performs all four tasks. They are passed one at a time to the servant and the elder until, at the end of the ceremony, the elder is performing all four tasks. No one can take on a less important task than those they already perform. Generate a sequence of moves to transfer all the tasks from the child to the elder. Write a recursive algorithm to perform the move sequence.

5. Using the move and path definitions for the knight's tour of Section 6.2.2, trace the execution of pattern_search on the goals:

   a. path(1,9).
   b. path(1,5).
   c. path(7,6).

   When the move predicates are attempted in order, there is often looping in the search. Discuss loop detection and backtracking in this situation.

6. Write the pseudo-code definition for a breadth-first version of pattern_search (Section 6.1.2). Discuss the time and space efficiency of this algorithm.

7. Using the rule in Example 6.2.3 as a model, write the eight move rules needed for the full 8×8 version of the knight's tour.

8. Using the goal and start states of Figure 6.3, hand run the production system solution to the 8-puzzle:

   a. In goal-driven fashion.
   b. In data-driven fashion.

9. Consider the financial advisor problem discussed in Chapters 2, 3, and 4. Using predicate calculus as a representation language:

   a. Write the problem explicitly as a production system.
   b. Generate the state space and stages of working memory for the data-driven solution to the example in Chapter 3.

10. Repeat problem 9.b to produce a goal-driven solution.

11. Section 6.2.3 presented the general conflict resolution strategies of refraction, recency, and specificity. Propose and justify two more such strategies.

12. Suggest two applications appropriate for solution using the blackboard architecture. Briefly characterize the organization of the blackboard and knowledge sources for each implementation.