CHAPTER 5

PRODDUCTION SYSTEMS

This part of the course discusses production rules and how these rules can be used to infer answers to queries.

By: Dr Muhanad Tahrir Younis

5-1 Production Systems

A production system is a model of computation which provides pattern directed control of a problem-solving process and consists of the following components:

- **A Set of Production Rules:** A production rule is a ***condition-action*** pair and defines a single chunk of problem-solving knowledge. The ***condition*** part determines when that rule may be applied to a problem instance. The ***action*** part defines the associated problem solving step. For instance, in an expert system for medical diagnosis, if a patient has fever (condition), then the nurse should give him paracetamol (action).

- **Working Memory:** It contains a description of the current state of the world. Specifically, the state is described in terms of predicates.

- **The Recognize-Act Cycle**: This is the control structure for a production system. It loops until the working memory pattern no longer matches the conditions of any of the production rules. This termination means two things: the system halts abnormally or there is an answer found to the query of the user.

A production rule is *enabled* when it matches the contents of working

memory. An enabled rule is part of the *conflict set. A conflict resolution technique is* needed to determine which rule to select based on the elements of the conflict set. It is *fired* when it is selected as the rule to use to know the next action.
A schematic drawing of a production system is presented in figure (5-1).

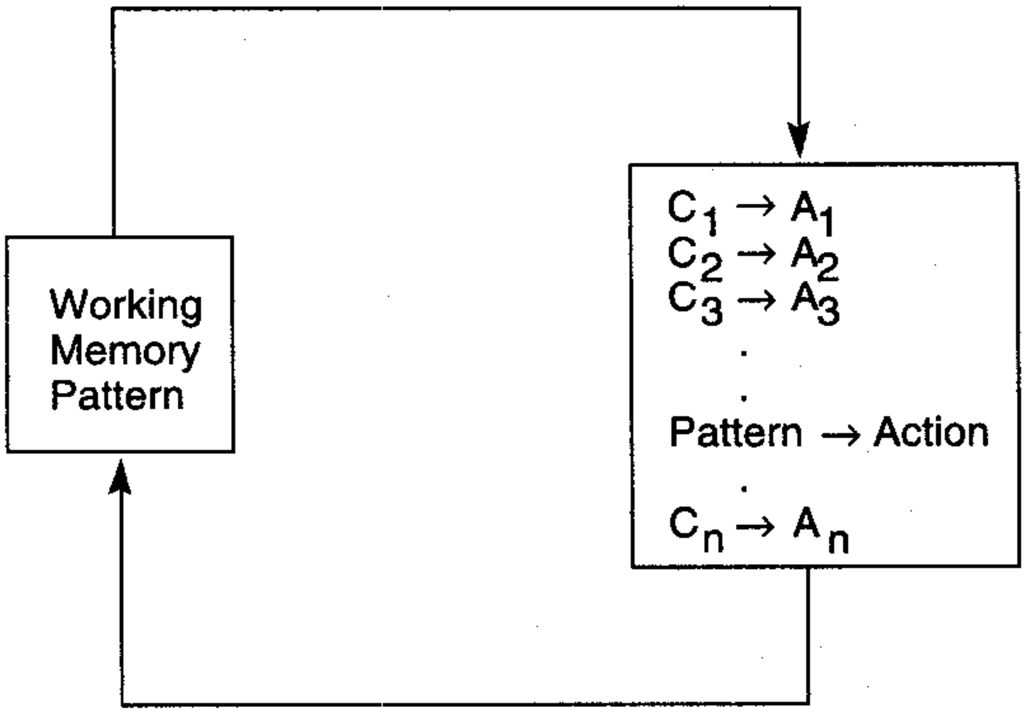


Figure (5-1): A production System. Control loops until working memory pattern no longer matches the conditions of any productions.

As an example, consider the production set program for sorting a string composed of the letters a, b, c. The production set has three rules. With the rule ba → ab, the ba is the condition and ab is the action. The same is true for the next 2 rules.

Production set:

(1) ba → ab
(2) ca → ac
(3) cb → bc

Suppose the current state of the working memory is: cbaca. The conflict set will contain rules 1,2 and 3 since ba, ca and cb are substrings of the current contents of working memory. The first rule is fired because the conflict resolution strategy used is lower-valued rule first. When rule 1 is fired it replaces the substring ba with ab, yielding cabca. Now only rule 2 matches the contents of working memory. Since there is only 1 rule matching the working memory, it is fired: This action changes the contents of working memory from cabca to acbca. This process goes op until there are no more rules which matches the contents of working memory. Figure (5-2) shows a trace of how this production system works.

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	ø	Halt

Figure (5-2): A Trace of How a Production System Works.

5-2 Control of Search in Production Systems

Given the recognize-act cycle in production systems, there are several ways for us to look at solutions to problems. We may either start with the data given to us, or start from the goal we wish to verify. These two approaches are called ***data-driven search*** (forward chaining) and ***goal-driven search*** (backward chaining).

- **Data-driven search (Forward Chaining):** This approach takes the facts of the problem and applies the rules/legal moves to produce new facts that lead to the goal.

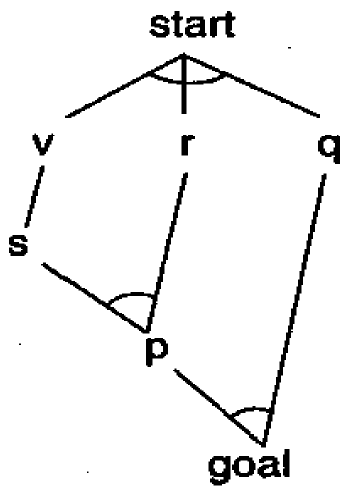
- **Goal driven search (Backward chaining):** It focuses on the goal, finds the rules that could produce the goals, and chains backward through successive rules and subgoals to the given facts of the problem.

How do we know which approach to use given our applications? If there are a large number of potential goals, or if all or most of the data is given it is best to use data-driven search. On the other hand, if a goal or a certain hypothesis is given, or if there is a large number of rules that match the facts of the problem or if the data is minimal then it is best to use goal-driven search.

Let us have another example and apply these two strategies.

(1) p Λ q → goal
(2) r Λ s → p
(3) w Λ r → q
(4) t Λ u→q
(5) v → s
(6) start → v Λ r Λ q

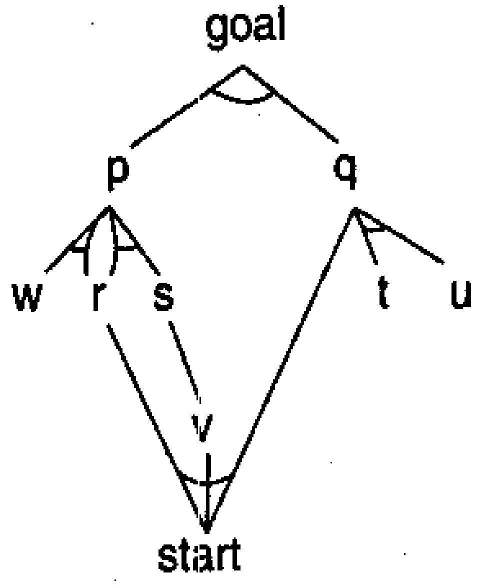
In this example we are given 6 rules, and our working memory initially contains **start**. We use forward chaining first. The conflict set will contain rule 6, and since it is the only rule in the conflict set it will be fired Notice that we match the contents of working memory with the condition part of the production rule. Firing rule 6 changes the contents of working memory to start, v, r, q. Looking at the condition part of our rules again, we match the contents of working memory with it. The conflict set this time contains rules 5 and 6. We fire rule 5 because we are applying the recency rule. We examine the rules again and see that this changes the contents of working memory to start, v, r, q, s. This process continues until we have reached the goal. Figure (5-3) shows the applying of forward chaining to a Production Set.



Iteration #	Working Memory	Conflict Set	Rule Fired
0	start	6	6
1	start,v,r,q	5,6	5
2	start,v,r,q,s	5,6,2	2
3	start,v,r,q,s,p	5,6,2,1	1
4	start,v,r,q,s,p,goal	ø	Halt

Figure (5-3): Applying Forward Chaining to a Production Set

Using backward chaining, we start the process from the goal. In this case hypothesis is identified to be goal. This time we do not match the contents of memory with the condition, instead we match it with the action part of the production rule. In this case, the only rule which matches the working memory is rule 1. This rule is fired and the working memory contains goal, p and q. This process continues, the contents of working memory being matched with the action part of the rule until no more rule matches the working memory. Figure (5-4) shows the applying of backward chaining to a Production System.



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	ø	Halt

Figure (5-4): Applying Backward Chaining to a Production System.

5-3 Examples of Production Systems

EXAMPLE (1): THE 8-PUZZLE

The 8-puzzle is a 3 x 3 array in which eight tiles can be moved around in nine spaces. Although in the physical puzzle moves are made by moving tiles, it is much simpler to think in terms of "moving the blank space" instead. The legal moves are:

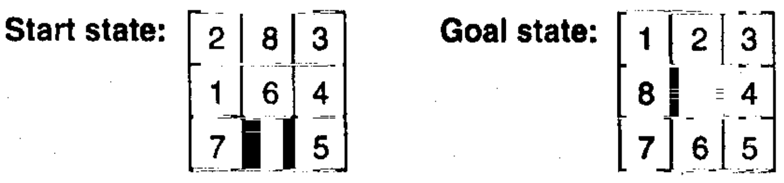
1- Move the blank up ↑
2- Move the blank right →
3- Move the blank down ↓
4- Move the blank left ←

In order to apply a move, we must sure that it does not move the blank off the board. Therefore, all four moves are not applicable at all times; for example, when the blank is in one of the corners only two moves are possible.

Legal moves are defined by the productions in Figure (5-5). 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 (5-5) follows in Figure (5-6). 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 5 is used in the solution of Figure (5-6). Note that the number of possible states of working memory grows exponentially with the depth of the search.



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 5.5 The 8-puzzle as a production system.

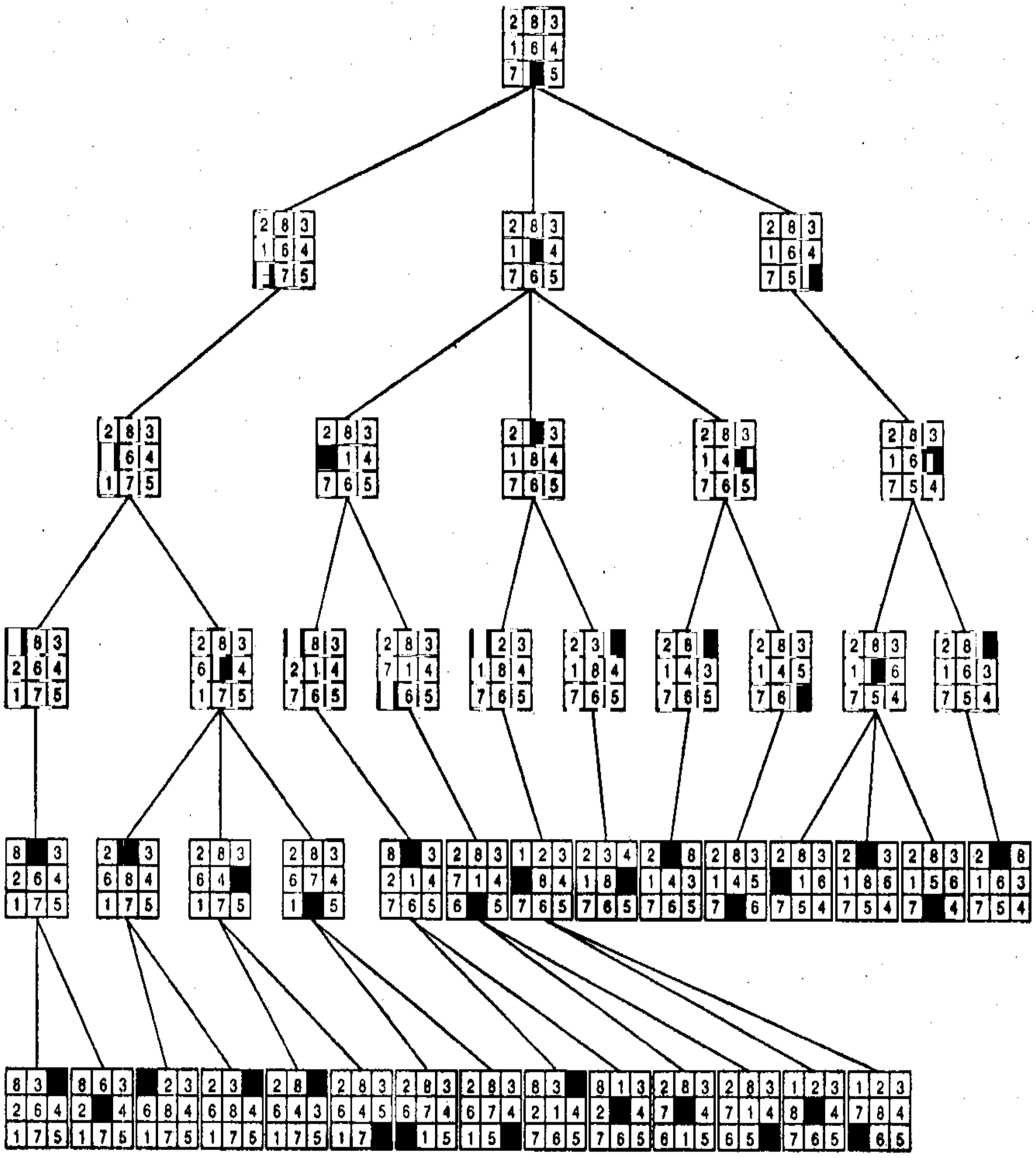


Figure 5.6 State space of the 8-puzzle searched by a production system with loop detection and a depth bound of 5.

EXAMPLE (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. The example given here is a simplified version of it: is there is a series of legal moves that will take the knight from one square to another on a reduced-size (3 x 3) chessboard?
Figure (5-7) shows a 3 x 3 chessboard with each square labeled with integers 1 to 9. the legal moves on the board are then described in predicate logic using a predicate called move, whose parameters are the starting and ending squares of a legal move.

```
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).
```

1	2	3
4	5	6
7	8	9

Figure (5-7): a 3 x 3 chessboard with move rules.

This problem may be solved using a production system approach. Here each move would be represented as a rule whose condition 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.
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 whether a path exists from square 1 to square 2 is charted in Figure (5-8).

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

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 (5-8): Production system solution to the 3 x 3 knight's tour problem.

There are several conflict resolution strategies we can apply to production systems. Among these are:

- **Rule Ordering**: Arrange all rules in one long priority list. The triggering rule appearing earliest in the list has the highest priority.
- **Refraction**: This strategy specifies that once a rule has fired, it may not fire again until the working memory elements that match its conditions have been modified.
- **Recency**: This approach prefers to select rules whose conditions match with the patterns most recently added to working memory.
- **Specificity**: This strategy supposes the conditions of one triggering rule are a superset of the conditions of another triggering rule. Use the rule with the superset that is more specialized to the current situation.