ADAPTIVE PRODUCTION SYSTEMS

D. A. Waterman

Department of Psychology
Carnegie-Mellon University

ABSTRACT

Adaptive production systems are defined and used to illustrate adaptive techniques in production system construction. A learning paradigm is described within the framework of adaptive production systems, and is applied to a simple rote learning task, a nonsense syllable association and discrimination task, and a serial pattern acquisition task. It is shown that with the appropriate production building mechanism, all three tasks can be solved using similar adaptive production system learning techniques.

I. INTRODUCTION

This paper presents results in the design and use of adaptive production systems (PS's). A PS (Newell & Simon, 1972; Newell, 1973) is a collection of production rules (PR's), that is, condition-action pairs, C => A, where the left side is a set of conditions relevant to a data base or working memory (WM) and the right side is a list of actions which can modify WM. The PS's to be discussed are written in PAS-II (Waterman & Newell, 1973; Waterman, 1973) and each is a set of ordered PR's. The control cycle consists of selecting one PR from the set and executing its actions. The first rule (in the ordered set) whose conditions match WM is the one selected. After the actions of the selected rule are executed the cycle repeats. This process continues until *no* conditions match.

An adaptive PS is one which can modify its own PR's. There are three ways this can take place: by adding new rules, deleting old rules, and changing existing rules; however, the PS's described here use only addition of new rules.

We now postulate a common machinery for learning: (1) a PS interpreter for ordered PS's, (2) a PS representation for learning programs, (3) PR actions for building rules and adding them to the system, and (4) the learning technique of adding new PR's above error-causing rules to correct the errors. Three learning tasks are investigated: arithmetic, verbal association, and series completion.

The programs for the tasks are written as short PS's which access a single WM composed of an ordered set of memory elements (ME's). When PR's "fire," i.e., their actions are executed, they modify WM by adding, deleting, or rearranging ME's. Such changes may cause different rules to fire on the next cycle and new memory modifications to be made. Thus the system uses WM as a buffer for holding initial data and intermediate results. Most actions modify WM. Some modify the PS by assembling WM elements into a PR and adding it to the PS. These actions give the PS its self-modification capability.

The arithmetic task consists *of* learning to add two integers given only an ordering over the set of integers. From this ordering, PR's which define the successor function are created and then used to calculate the desired sum. The verbal association task is a PS implementation of EPAM (Feigenbaum, 1963). Instead of growing an EPAM discrimination net, the system creates a set of PR's equivalent to such a net. The series completion task consists of predicting the next symbols in a sequence, such as AABBAABB.. Here PR's are created which represent hypotheses about which symbol contexts lead to which new symbols, i.e., "two A's always lead to a B." These rules constitute the concept of the series and are used to predict new symbols.

II. PAS-II PRODUCTION SYSTEM

The PAS-II PS interpreter is modeled after PSC (Newell, 1972, 1973). PR's in PAS consist of condition-action pairs, where the condition side is a set of conditions with implicit MEMBER and AND functions and the action side is an ordered list of independent actions. A rule to deposit (C) and (D) into WM if it already contains (A) and (B) is:
(A) (B) => (DEP (C)) (DEP (!))), where the action DEP deposits its argument into WM. The control cycle of the PS interpreter consists of two mechanisms: RECOGNIZE and ACT. A cycle *is* defined to be a single RECOGNIZE-ACT sequence, and is repeated un~ til either no rules match WM or a *PS* halting action is executed.

RECOGNIZE. RECOGNIZE selects a rule to be executed. When many rules match WM a conflict occurs, as RECOGNIZE must produce one rule for ACT to work on. Conflict resolution consists of applying a scheme to select one rule from those that match WM. The only conflict resolution used here is priority ordering. Thus the rule recognized is the highest priority rule whose conditions match WM.

The match mechanism assumes implicit MEMBER and AND notation and scans condition elements (CE's) in order from left to right to see if each element is in WM. When all the CE's in a rule match corresponding ME's, the ME's are brought to the front of memory (just before actions are executed) in the order specified in the rule. A ME can match only one CE in any rule and the order of the ME's does not have to correspond to the order of the CE's. For example, the conditions (A) (B) (A) will match WM; (B) (A) (A), but not WM: (A) (B). A CE will match a ME if the ME contains all items the CE contains, in the same order, starting from the beginning of the ME. Thus CE (A T) will match ME's (A T), and (A T E) but not (A), (T A), or (T A T). The match routine searches for the absence of a ME if the CE is preceeded by a minus sign (-). Thus (A) - (B) will match WM if it contains (A) but does not contain (B). Free variables can be used in the CE's and are denoted x1, x2,...xn. When a match occurs each item in the ME which corresponds to a variable is bound to that variable. For example, with WM; (A) (B(L)) and PR: (xl) (B x2) -> (DEP x2), xl is bound to A, and x2 to (L). The action taken will be to deposit (L) into WM.

ACT. ACT takes the rule specified by RECOGNIZE and executes all its actions, one at a time, in order from left to right. Actions in a PS are critical since they determine the grain of the system. If the grain is too coarse a single action may embody all interesting activity, obscuring it from view. The criterion in defining actions is to make them primitive enough so the PS trace will exhibit the activity deemed interesting.

BASIC ACTIONS

(DEP a): Deposit a into front of WM.
(REM a): Remove first occurrence of a from WM.
(REP a b n): Replace a with b in nth ME. Default n is 1.
(SAY a): Print a. Handles multiple arguments.
(CLEAR a): All ME's except a are removed from WM.
(ATTEND): Read from terminal, to let user modify WM.
(STOP): Stop production system execution.

MODIFICATION ACTIONS

(COND a): Deposits (COND a) into WM. Is the same as (DEP (COND a)).
(ACTION a): Deposits (ACTION a) into WM. Is the same as (DEP (ACTION a)).
(MARK a): Marks each ME that just matched the CE's of the rule containing (MARK a). Element e is marked by changing it to (a e).
(USED): Exactly equivalent to (MARK USED).
(OLD): Exactly equivalent to (MARK OLD).
(PROD a): Creates a PR from all ME's marked (COND...) and (ACTION...) and inserts it into the PS just in front of the first PR that contains an element identical to any arguments of PROD. If PROD has no arguments the rule is inserted at the front of the PS. If it has argument END, it is inserted at the end. In all cases all ME's marked (COND...) and (ACTION...) are removed from WM.

SPECIAL ACTIONS

(SUCC): Changes every ME beginning with a number by replacing that number with its successor.
(PERCEIVE a b): Breaks syllable a into letters and tags the letters with b and a number specifying their order in a.
(OBSERVE a b): Breaks word a into letters and tags the letters with b.
(PRODS a b c d): Here b is the hypothesized period (initially 1), and c is the series. As in PROD, ME's marked (COND...) and (ACTION...) are combined and removed from WM to create a new rule. This rule is generalized according to the template heuristic (see text) and placed just above the first rule containing a. The d is the number of rules already added to PS by PRODS.

Table 1. PAS Production System Actions

The three types of PAS actions are: basic, modification, and special, as shown in Table 1. They assume WM is an ordered list of ME's going from left to right. Thus DEP places ME's into WM at the left, and REP counts ME's starting from the left. The modification actions will now be illustrated.

```
WM: (B)(A)(C)(ACTION (SAY DONE)(STOP))(COND (C))
PS: 1. - (B) => (DEP (B))
    2. (A) (B) => (MARK COND) (PROD (A))
    3. (C) => (DEP (A))
```

When PS is fired, rule 2 is the first to match WM. The (MARK COND) marks (A) and (B) and memory becomes:

```
WM: (COND (A))(COND (B))(C)(ACTION (SAY DONE)(STOP
    (COND (C))
```

Then (PROD (A)) creates a PR out of the elements marked COND and ACTION, removes them from WM, and puts the new PR just above the first rule containing (A), rule 2.

```
WM:     (C)
PS:     1. - (B) => (DEP (B))
      1.5. (A) (B) (C) => (SAY DONE) (STOP)
        2. (A) (B) => (MARK COND) (PROD (A))
        3. (C) => (DEP (A))
```

After the insertion of rule 1.5, execution continues, terminating with the firing of 1.5.

Implicit actions are implemented in PAS, i.e. predecessors and successors on letters can be accessed implicitly by placing apostrophes before or after PR variables. Thus the value of x1' is the successor of the value of x1, and the value of ''x1 is the double predecessor of the value of x1.

III. PRODUCTION SYSTEM FOR ADDITION

The addition PS is designed to illustrate adaptive techniques and to compare PS programming with conventional methods. It does not attempt to model data on human performance in this task. Figure 1 shows the ADD PS.

```
1. (READY) (ORDER X1) => (REP (READY) (COUNT X1))
   (ATTEND)
2. (N X1) - (NN) - (S NN) => (DEP (NN X1))
3. (COUNT X1) (M X1) (NN X2) (N X3) =>
   (SAY X2 IS THE ANSWER) (COND (M X1) (N X3))
   (ACTION (STOP)) (ACTION (SAY X2 IS THE ANSWER))
   (PROD) (STOP)
4. (COUNT) (NN) => (REP (COUNT) (S COUNT))
   (REP (NN) (S NN) 2)
5. (ORDER X1 X2) => (REP (X1 X2) (X2)) (COND (S X3 X1))
   (ACTION (REP (S X3 X1) (X3 X2))) (PROD)
```

Figure 1. ADD: A Production System for Addition of Integers

When ATTEND is executed the system is given two integers, (M a) (N b). It calculates a + b and prints the answer. The algorithm used is shown below.

```
2.1  add(m,n) = count ← 0; nn ← n;
2.2  L1 if count = m then return(nn);
2.3     count ← successor(count);          (2)
2.4     nn ← successor(nn);
2.5     go(L1).
```

Count and nn are local variables initialized to zero and n respectively. Count and nn are continuously incremented by one, using the successor function, until count equals m. At this point the answer is nn.

ADD performs these steps with some differences. First, it has no successor function, so it creates a PR representation of that function. Second, once a sum is calculated it adds a rule that produces the answer directly the next time. Thus it builds the addition table for integers.

There is a direct mapping, however, between the code in (2) and that in Figure 1. Rules 1 and 2 in Figure 1 correspond to line 2.1. Rule 3 corresponds to 2.2, and rule 4 to 2.3 and 2.4. Rule 5 has no correspondent in (2) since the code assumes the existence of the successor function, while the PS creates it. Note that 2.5, the GOTO statement, has no correspondent in Figure 1. In ADD the function of the GOTO and label is handled by control cycle repetition, which permits looping, and memory modification, which in this case makes rules 1 and 2 inoperative. A trace of ADD solving 4 + 2 is shown in Figure 2.

IV. PRODUCTION SYSTEM IMPLEMENTATIONS OF EPAM

EPAM (Feigenbaum, 1963; Feigenbaum & Simon, 1964) is a program which simulates verbal learning behavior by memorizing three-letter nonsense syllables presented in associate pairs. The program learns to predict the correct response when given a stimulus syllable by growing a discrimination net composed of nodes which are tests on the values of certain attributes of the letters in the syllable. Responses are stored at the terminal nodes, and are retrieved by sorting the stimuli down the net. A paired associate training sequence for this learning task is shown in Figure 3.

Stimulus	Response
PAX	CON
BEK	LUQ
CIT	DER
BUK	MAB
NAL	LEQ
REB	MOL
NOJ	PED

Figure 3. Paired Associate Training Sequence for Verbal Learning Task.

EPAM1. Figure 4 shows EPAM1. This program grows a PS which is analogous to a discrimination net with tests for stimulus letters (i.e., "is the 3rd letter R?") at the intermediate nodes and complete responses at terminal nodes.

```
*memory display ps display
MEMORY MODE
    1. STM * (READY) (ORDER 0 1 2 3 4 5 6 7 8 9)

PS MODE
    1. (READY) (ORDER X1) => (REP (READY)
       (COUNT X1)) (ATTEND)
    2. (N X1) - (NN) - (S NN) => (DEP (NN X1))
    3. (COUNT X1) (M X1) (NN X2) (N X3) =>
       (SAY X2 IS THE ANSWER) (COND (M X1)
       (N X3)) (ACTION (STOP))
       (ACTION (SAY X2 IS THE ANSWER))
       (PROD) (STOP)
    4. (COUNT) (NN) => (REP (COUNT) (S COUNT))
       (REP (NN) (S NN) 2)
    5. (ORDER X1 X2) => (REP (X1 X2) (X2))
       (COND (S X3 X1))
       (ACTION (REP (S X3 X1) (X3 X2))) (PROD)

*fire
    1 TRUE IN PS
OUTPUT FOR (ATTEND) * (dep (m 4)(n 2))
    STM: (N 2) (M 4) (COUNT 0)
       (ORDER 0 1 2 3 4 5 6 7 8 9)

    2 TRUE IN PS
    STM: (NN 2) (N 2) (M 4) (COUNT 0)
       (ORDER 0 1 2 3 4 5 6 7 8 9)

    4 TRUE IN PS
    STM: (S COUNT 0) (S NN 2) (N 2) (M 4)
       (ORDER 0 1 2 3 4 5 6 7 8 9)

    5 TRUE IN PS
NOW INSERTING
(S X3 0) => (REP (S X3 0) (X3 1))
ON LINE 0.5
    STM: (ORDER 1 2 3 4 5 6 7 8 9)
       (S COUNT 0)(S NN 2) (N 2) (M 4)

  0.5 TRUE IN PS
    STM: (COUNT 1) (ORDER 1 2 3 4 5 6 7 8 9)
       (S NN 2) (N 2) (M 4)

    5 TRUE IN PS
NOW INSERTING
(S X3 1) => (REP (S X3 1) (X3 2))
ON LINE 0.25
    STM: (ORDER 2 3 4 5 6 7 8 9) (COUNT 1)
       (S NN 2) (N 2) (M 4)

    5 TRUE IN PS
NOW INSERTING
(S X3 2) => (REP (S X3 2) (X3 3))
ON LINE 0.13
    STM: (ORDER 3 4 5 6 7 8 9) (COUNT 1)
       (S NN 2) (N 2) (M 4)

  0.13 TRUE IN PS
    STM: (NN 3) (ORDER 3 4 5 6 7 8 9)
       (COUNT 1) (N 2) (M 4)

    4 TRUE IN PS
    STM: (S COUNT 1) (S NN 3)
       (ORDER 3 4 5 6 7 8 9) (N 2) (M 4)

  0.25 TRUE IN PS
    STM: (COUNT 2) (S NN 3)
       (ORDER 3 4 5 6 7 8 9) (N 2) (M 4)

    5 TRUE IN PS
NOW INSERTING
(S X3 3) => (REP (S X3 3) (X3 4))
ON LINE 0.05
    STM: (ORDER 4 5 6 7 8 9) (COUNT 2)
       (S NN 3) (N 2) (M 4)

  0.05 TRUE IN PS
    STM: (NN 4) (ORDER 4 5 6 7 8 9)
       (COUNT 2) (N 2) (M 4)

    4 TRUE IN PS
    STM: (S COUNT 2) (S NN 4)
       (ORDER 4 5 6 7 8 9) (N 2) (M 4)

  0.13 TRUE IN PS
    STM: (COUNT 3) (S NN 4)
       (ORDER 4 5 6 7 8 9) (N 2) (M 4)

    5 TRUE IN PS
NOW INSERTING
(S X3 4) => (REP (S X3 4) (X3 5))
ON LINE 0.03
    STM: (ORDER 5 6 7 8 9) (COUNT 3)
       (S NN 4) (N 2) (M 4)

  0.03 TRUE IN PS
    STM: (NN 5) (ORDER 5 6 7 8 9)
       (COUNT 3) (N 2) (M 4)

    4 TRUE IN PS
    STM: (S COUNT 3) (S NN 5)
       (ORDER 5 6 7 8 9) (N 2) (M 4)

  0.05 TRUE IN PS
    STM: (COUNT 4) (S NN 5)
       (ORDER 5 6 7 8 9) (N 2) (M 4)

    5 TRUE IN PS
NOW INSERTING
(S X3 5) => (REP (S X3 5) (X3 6))
ON LINE 0.02
    STM: (ORDER 6 7 8 9) (COUNT 4)
       (S NN 5) (N 2) (M 4)

  0.02 TRUE IN PS
    STM: (NN 6) (ORDER 6 7 8 9)
       (COUNT 4) (N 2) (M 4)

    3 TRUE IN PS

6 IS THE ANSWER

NOW INSERTING
(M 4) (N 2) => (SAY 6 IS THE ANSWER) (STOP)
ON LINE 0.01
    STM: (COUNT 4) (M 4) (NN 6)
       (N 2) (ORDER 6 7 8 9)

*display
  0.01: (M 4) (N 2) => (SAY 6 IS THE ANSWER)
       (STOP)
  0.02: (S X3 5) => (REP (S X3 5) (X3 6))
  0.03: (S X3 4) => (REP (S X3 4) (X3 5))
  0.05: (S X3 3) => (REP (S X3 3) (X3 4))
  0.13: (S X3 2) => (REP (S X3 2) (X3 3))
  0.25: (S X3 1) => (REP (S X3 1) (X3 2))
   0.5: (S X3 0) => (REP (S X3 0) (X3 1))
```

Figure 2. Trace of ADD Production System on 4 + 2.

```
1. (READY) (STIM X1) => (REM (READY)) (PERCEIVE X1 ?)
2. (READY) => (ATTEND STIM)
3. (REPLY) - (RESP) => (ATTEND RESP)
4. (REPLY X1) - (RESP X1) => (REP REPLY WRONG)
5. (USED X1) (WRONG X2) => (REP USED COND)
6. - (RESP) => (DEP (REPLY ?)) (SAY ?) (ATTEND RESP)
7. (X1 X2 ?) (RESP X3) (WRONG X4) => (COND (X1 X2 ?))
   (ACTION (USED) (DEP (REPLY X3)) (SAY X3))
   (PROD (SAY X4)) (STOP)
```

Figure 4. EPAM1: A Production System Implementation of EPAM

EPAM1 learning PAX-CON and PUM-JES is illustrated below.

```
*fire
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim pax))
    STM: (STIM PAX) (READY)

    1 TRUE IN PS
    STM: (1 P ?) (3 X ?) (2 A ?) (STIM PAX)

    6 TRUE IN PS

?

OUTPUT FOR (ATTEND RESP) = (dep (resp con))
    STM: (RESP CON) (REPLY ?) (1 P ?) (3 X ?) (2 A ?)
       (STIM PAX)
```

Initially WM (here called STM) contains (READY). Rule 2 fires and the system asks for the stimulus. Then 1 fires, adding stimulus components to memory. Next 6 fires and prints a question mark as the system's reply to the stimulus, adds this reply to memory, and asks for the correct response.

```
    4 TRUE IN PS
    STM: (WRONG ?) (RESP CON) (1 P ?) (3 X ?) (2 A ?)
       (STIM PAX)

    7 TRUE IN PS
NOW INSERTING
(1 P ?) -> (USED) (DEP (REPLY CON)) (SAY CON)
ON LINE 5.5
    STM: (1 P ?) (RESP CON) (WRONG T) (3 X ?) (2 A T)
       (STIM PAX)
```

Since the reply (?) does not match the response (CON), 4 fires and changes the label REPLY to WRONG. Now 7 fires creating rule 5.5.

```
*initialize fire
INITIALIZED
    2 TRUE IN PS
OUTPUT FOR (ATTEND STIM) = (dep (stim pum))
    STM: (STIM PUM) (READY)

    1 TRUE IN PS
    STM: (1 P ?) (3 M ?) (2 U ?) (STIM PUM)

  5.5 TRUE IN PS

CON

    STM: (REPLY CON) (USED (1 P ?)) (3 M ?) (2 U ?)
       (STIM PUM)
```

Before the second pair of syllables is presented, memory is initialized back to (READY), and the system is restarted. Again 2 and 1 are fired to obtain and perceive the stimulus. But now 5.5 matches WM and causes (1 P ?) to be marked USED, and the system to reply CON and add the reply to memory. This is an example of stimulus generalization: the system confused PUM with PAX since it was only noticing first letters.

```
    3 TRUE IN PS
OUTPUT FOR (ATTEND RESP) = (dep (resp jes))
    STM: (RESP JES) (REPLY CON) (USED (1 P ?)) (3 M ?)
       (2 U ?) (STIM PUM)

    4 TRUE IN PS
    STM: (WRONG CON) (RESP JES) (USED (1 P ?)) (3 M ?)
       (2 U ?) (STIM PUM)

    5 TRUE IN PS
    STM: (COND (1 P ?)) (WRONG CON) (RESP JES) (3 M ?)
       (2 U ?) (STIM PUM)

    7 TRUE IN PS
NOW INSERTING
(3 M ?) (1 P ?) => (USED) (DEP (REPLY JES)) (SAY JES)
ON LINE 5.3
    STM: (3 M ?) (RESP JES) (WRONG CON) (2 U ?)
       (STIM PUM)
```

Now memory contains a reply but no response, so 3 fires and elicits the correct response (JES) from the user. Rule 4 fires, since the reply differs from the response, marking the reply wrong. Next 5 fires, changing the USED label to COND. Finally 7 fires and creates a new rule with two condition elements, one from the COND already in memory and one from the COND inserted by rule 7, The two rules just added are:

```
5.3. (3 M ?) (1 P ?) => (USED) (DEP (REPLY JES))
     (SAY JES)
5.5. (1 P ?) => (USED) (DEP (REPLY CON)) (SAY CON)
```

PAX will now elicit the response CON, and PUM the response JES, as desired.

EPAM2. Figure 5 shows EPAM2. This complete version of EPAM grows a PS in which response cues rather than complete responses are stored in some terminal nodes. These cues (i.e., C_N) are retrieved by dropping the stimulus through the net, and are then themselves dropped through the net to retrieve the responses stored in other terminal nodes.

```
 1. (READY) (STIM X1) => (REM (READY)) (PERCEIVE X1 ?)
 2. (READY) => (ATTEND STIM)
 3. (REPLY) - (RESP) => (ATTEND RESP)
 4. (REPLY X1) - (RESP X1) => (REP REPLY WRONG)
 5. (REPLY X1) (RESP X1) => (STOP)
 6. (USED) (TEST X1) - (TEST X2) => (REP USED USED*)
 7. (TEST X1) (TEST X2) (X3 X4 ?) => (REM (X3 X4 ?))
 8. (TEST X1) (TEST X2) - (R-GEN) =>
    (DEP (REPLY X1) (R-GEN)) (SAY X1)
 9. - (RESP) => (DEP (REPLY ?)) (SAY ?) (ATTEND RESP)
10. (RESP X1) - (X2 X3 RESP) => (PERCEIVE X1 RESP)
11. (WRONG) (TEST X1) (STIM X1) - (R-GEN) =>
    (DEP (R-GEN))
12. (OLD X1) (R-GEN) => (REP OLD COND) (DEP (HOLD X1))
```

```
13. (USED X1) (USED*) (R-GEN) => (REP USED COND)
    (DEP (HOLD X1))
14. (R-GEN) (COND (X1 X2 ?)) (X1 X2 RESP) =>
    (REM (X1 X2 RESP))
15. (X1 X2 RESP) (RESP X3) (WRONG X4) - (DONE) =>
    (COND (X1 X2 ?)) (ACTION (OLD) (DEP (REPLY X3))
    (SAY X3)) (PROD (SAY X4) (TEST X4)) (DEP (DONE))
16. (USED* X1) => (REP USED* COND)
17. (OLD) (DONE) - (TEST) => (REP OLD COND)
18. (R-GEN) (HOLD (X1 X2 ?)) => (REM (HOLD (X1 X2 ?)))
    (ACTION (DEP (X1 X2 ?)))
19. (R-GEN) (X1 X2 RESP) (STIM X3) (WRONG X4) =>
    (ACTION (DEP (TEST X3))) (ACTION (USED)
    (DEP (X1 X2 ?))) (PROD (DEP (TEST X3))) (STOP)
20. (X1 X2 ?) (X3 X4 RESP) (STIM X5) (WRONG X6) =>
    (COND (X1 X2 ?)) (ACTION (USED) (DEP (X3 X4 ?))
    (DEP (TEST X5))) (PROD (SAY X6)) (STOP)
```

Figure 5. EPAM2: A Production System Implementation of EPAM

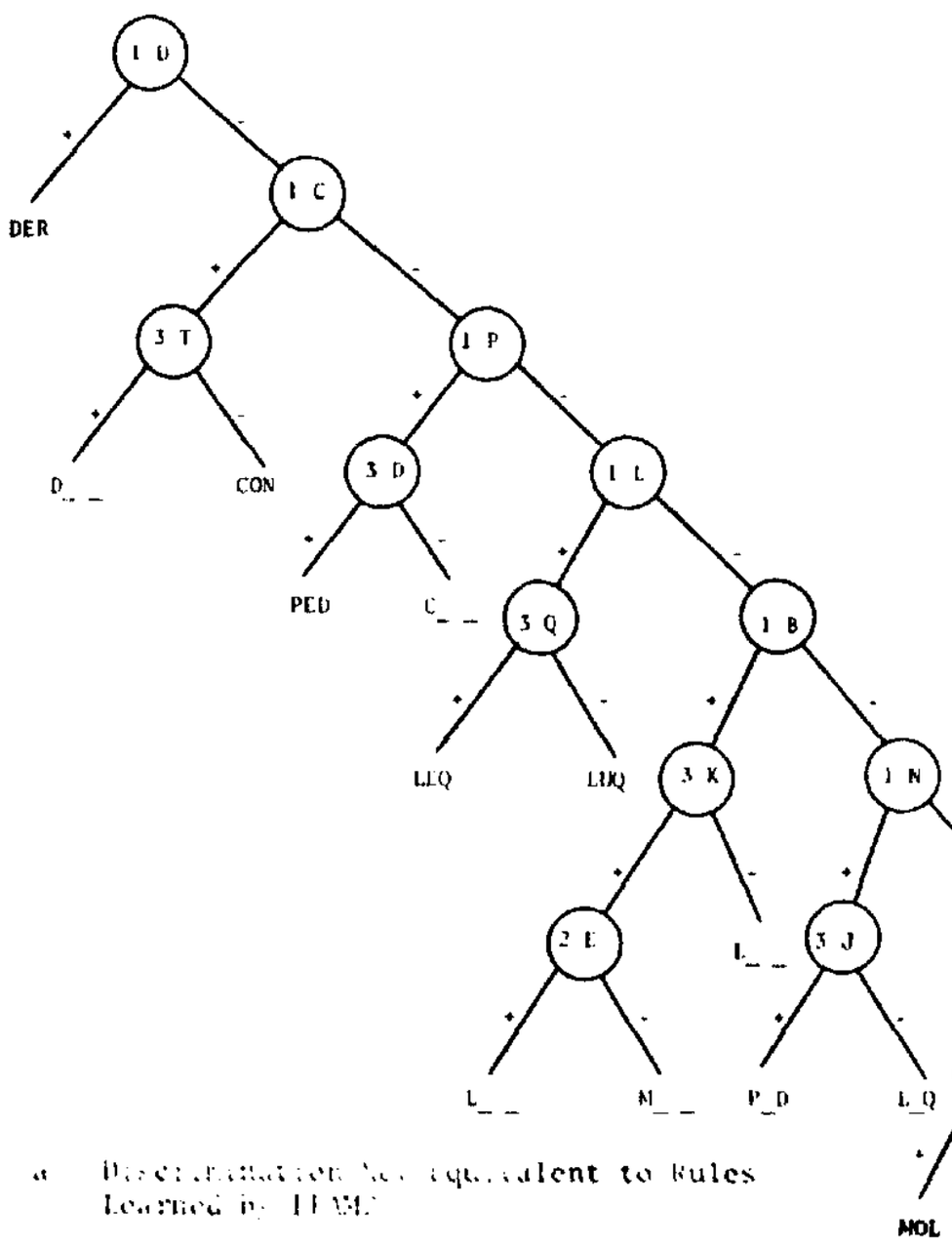
EPAM2 was given the stimulus-response pairs of Figure 3 and produced the output shown in Figure 6. There were two instances of stimulus generalization, two of response generalization, one of both stimulus and response generalization, and two of stimulus-response confusion.

STIMULUS	REPLY 1	REPLY 2	REPLY 3	RESPONSE
PAX	?	CON	CON	CON
BEK	?	MAB (SG)	LUQ	LUQ
CIT	CON (SR)	DER	DER	DER
BUK	LUQ (SG)	MAB	MAB	MAB
NAL	?	LUQ (RG)	LEQ	LEQ
REB	?	MAB (RG)	MOL	MOL
NOJ	LUQ (SG RG)	PAX (SR)	PED	PED

Figure 6. EPAM2 Output for Three Training Trials (SG: stimulus generalization error, RG: response generalization error, SR: stimulus-response confusion).

The PR's learned by EPAM2 and the corresponding discrimination net are shown in shorthand notation* in Figure 7. Note that the condition elements are analogous to intermediate nodes and the response elements to the terminal nodes in the net, and the path through the net from the top to a terminal node corresponds to the sequence of conditions tested in the PS to obtain a response

*Conditions, like (1 P), are elements denoting a letter and its location in the syllable, and are ordered (first, third, second) according to syllable location. Actions are response words like CON, or partial response cues like (1 M).



a. Discrimination Net Equivalent to Rules Learned by EPAM2

```
                (1 D) =>  DER
(1 C)(3 T)      (1 D) =>  (1 D)
                (1 C) =>  CON
(1 P)(3 D)      (1 D) =>  PED
                (1 P) =>  (1 C)
(1 L)(3 Q)      (1 L) =>  LEQ
                (1 L) =>  LUQ
(1 B)(3 K)(2 E) (1 L) =>  (1 L)
(1 M)(3 L)      (1 M) =>  MOL
                (1 M) =>  MAB
(1 B)(3 K)      (1 M) =>  (1 M)
                (1 P) =>  PED
(1 N)(3 J)      (1 P) =>  (1 P)(3 D)
(1 N)(3 J)      (1 P) =>  (1 P)
                (1 L) =>  LUQ
                (1 B) =>  (1 L)
                (1 L) =>  LEQ
                (1 N) =>  (1 L)(3 Q)
                (1 N) =>  (1 L)
                (1 M) =>  MOL
                (1 R) =>  (1 M)(3 L)
                (1 R) =>  (1 M)
```

b. Production Rules Learned by EPAM2

Figure 7. Discrimination Net and Production Rules for EPAM2.

V. PRODUCTION SYSTEM FOR SERIES COMPLETION

Computer models of series completion (Simon 6 Kotovsky, 1963; Klahr & Wallace, 1970; Williams, 1972) have been complex programs with structures quite dissimilar from those of more basic learning models. Here we provide a common structure for these learning tasks. The essence of their commonality is (1) an ordered PS representation of what is learned, and (2), the technique of adding new PR's above the error-causing rules to correct errors A PS will now be described which can solve complex letter series completion tasks which require the use of same, successor, or predecessor operations on the alphabet.

Learning Technique

PR's are created which represent hypotheses about what symbols come next given a current context of symbols. These hypotheses are tested by checking the given series to see if the current set of PR's (the learned PS) correctly predicts each symbol in the series given the partial series up to that symbol. When every symbol is correctly predicted, the system uses the learned PS and the entire problem series to predict the next symbol in the series.

For the series CABCAB the rule CA-^B would be learned. This means "if the last two letters of the partial series are CA, the next is B."

Before being added to the system, rules are generalized to take into account the relevant letter relationships. The problem is that rules can be generalized many ways, each being a hypothesis about which letter relationships are relevant for the series. The variations on C A -> B are shown below.

```
      x1 A → B
      C x1 → B
      x1 A → 'x1
      C x1 → x1'
    x1 x2 → B
x1 ''x1 → B
    x1 x2 → 'x1
x1 ''x1 → 'x1
```

The first rule above means "any letter followed by A leads to B", the second is "C followed by any letter leads to B", and the third "any letter followed by A leads to the predecessor of that letter."

If for every new rule the system arbitrarily picked a generalization, intending to backtrack to try the others when an error occurred, a huge tree of possibilities would be generated, making the problem unsolvable. The solution is to use tree-pruning heuristics to limit the number of generalizations at each step. The PS to be described uses one powerful heuristic, the template heuristic.

EXAMPLE OF LEARNING TECHNIQUE

Line Number	Period Size Hypothesis	Series	Old PS	Prediction	Valid?	Rule Components	New Rule	New PS
1	1	A / B H B C I C D	x1 → x1	A	(-)	A → B	x1 → x1'	x1 → x1' (1) x1 → x1 (default)
2	1	A B / H B C I C D	x1 → x1' x1 → x1	C	(-)	A B → H	x1 x1' → H	x1 x1' → H (2) x1 → x1' (1) x1 → x1 (default)
3	2	A / B H B C I C D	x1 → x1	A	(-)	A → B	x1 → B	x1 → B (init) x1 → x1 (default)
4	2	A B / H B C I C D	x1 → B x1 → x1	B	(-)	A B → H	none	x1 → B (init) x1 → x1 (default)
5	3	A / B H B C I C D	x1 → x1	A	(-)	A → B	x1 → B	x1 → B (init) x1 → x1 (default)
6	3	A B / H B C I C D	x1 → B x1 → x1	B	(-)	A B → H	x1 x2 → H	x1 x2 → H (init) x1 → B (init) x1 → x1 (default)
7	3	A B H / B C I C D	x1 x2 → H x1 → B x1 → x1	H	(-)	A B H → B	x1 x2 x3 → x1'	x1 x2 x3 → x1' (1) x1 x2 → H (init) x1 → B (init) x1 → x1 (default)
8	3	A B H B / C I C D	x1 x2 x3 → x1' x1 x2 → H x1 → B x1 → x1	C	(+)	none	none	same as above
9	3	A B H B C / I C D	same as above	I	(+)	none	none	same as above
10	3	A B H B C I / C D	same as above	C	(+)	none	none	same as above
11	3	A B H B C I C / D	same as above	D	(+)	none	none	same as above
12	(period known to be 3)	A B H B C I C D	same as above	J				

Table 2. Learning Technique Illustrated for ABHBCICD

The template heuristic consists of hypothesizing period size, and recognizing only relations between letters which occupy the same relative position within the period, while generalizing on all letters. For example, if given the series ACABA with period 2, then the relations looked for are shown by the arrows below.

Learning proceeds as follows: period size is hypothesized and the series goes through a partition-prediction cycle. Generalized rules are added, and the cycle is performed once for each period hypothesis. A period hypothesis is false if:

(1) no relation is found between letters occupying the same relative position within the period
or (2) the number of inter-period rules added exceeds the period size hypothesis.

When the period hypothesis is false, it is increased by 1, and the cycle starts over. Table 2 shows this procedure for the series ABHBCICD. In line 1 we see the default rule x1 -> x1 (always considered to generate an error) and the partitioned series. Everything to the left of the slash (/) is the current context. Context A is dropped through the rules and A is predicted. This is not valid (-), as the actual next letter is B. Now the system takes context A and next letter B to form A -. B, generalizes it to get x1 -> x1', and places it above the error-causing (default) rule as shown. In line 2 the number of rules added (2) exceeds the period size hypothesis (1) so a new size hypothesis (2) is made in line 3 In line 4 the rule cannot be generalized since no relation can be found between A and H*, thus size 3 is hypothesized in line 5. Line 11 completes the learning cycle and line 12 illustrates the PS making its first actual extension to the series. The concept of the series is now embodied in the numbered rules (the inter-period rules). Thus we say that xl x2 x3 -> x1' is the concept learned by the system, and the series predicted by this concept is ABHBCICDJDEK... .

Production System

Figure 8 shows the PS for letter series completion. Rules 1 and 2 provide initialization, rule 16 acts as the default rule, and rule 13 adds productions to the system. Figure 9 shows concepts learned using the 15 series from Simon and Kotovsky (1963). The correct predictions are made in all cases. For more on serial pattern acquisition see Waterman (1975).

•The system does not search for relations higher than triple predecessor or successor.

```
1. (READY) (SERIES X1) => (REP READY CONT)
   (DEP (PNUM 2) (COUNTS 0)) (OBSERVES X1 ?)
2. (READY) => (ATTEND SERIES) (DEP (PERIOD 1))
3. (COUNT) (COUNTS X1) => (REM (COUNT))
   (REP X1 X1')
4. (0 X1 ?) => (SUCC)
5. (FAIL) (PERIOD X1) (SERIES X2) => (ERASE)
   (CLEAR) (DEP (READY) (PERIOD X1') (SERIES X2))
6. (PERIOD X1) (COUNTS X1') (SERIES X2) =>
   (ERASE) (CLEAR)
   (DEP (READY) (PERIOD X1') (SERIES X2))
7. (NEXT X1) - (X2 ?) - (ACTION) => (SAY X1)
   (DEP (MATCH) (X1 ?)) (STOP)
8. (NEXT X1) (USED) (ACTION (USED) (DEP (NEXT X1)))
   - (MATCH) - (ERROR) => (DEP (MATCH))
9. (X1 X2 ?) (NEXT) - (DONE) =>
   (DEP (OLD (X1 X2 ?))) (DEP (DONE))
10. (USED (X1 X2 ?)) => (REP USED OLD)
   (DEP (X1 X2 ?))
11. (OLD (X1 X2 ?)) => (REP OLD COND)
12. (MATCH) (NEXT X1) (SERIES X2) (LOC X3) =>
   (REP (NEXT X1) CONT 2)
   (REM (MATCH) (DONE) (LOC X3)) (PROD (SERIES X2))
13. (LOC X1) (NEXT X2) (PERIOD X3) (SERIES X4)
   (COUNTS X5) => (REM (LOC X1) (DONE) (ERROR))
   (REP (NEXT X2) CONT) (PRODS (LOC X1) X3 X4 X5)
14. (CONT) (X1 ?) (PNUM X2) (X3 ?) =>
   (REP X1 (0 X1) 2) (REP X2 X2' 3) (REM (CONT))
   (ACTION (USED) (DEP (NEXT X3)) (DEP (LOC X2)))
15. (CONT) (X1 ?) => (REP X1 (0 X1) 2)
   (REM (CONT))
16. (I X1 ?) => (DEP (NEXT X1) (LOC 1))
```

Figure 8. Production System for Series Completion Task

	Series	Rules	Prediction
1.	CDCDCD	x1 x2 → x1	CDC
2.	AAABBB	x1 x2 x3 → x1'	CCC
3.	ATBATAATB	x1 x2 x3 x4 x5 x6 → x1	ATA
4.	ABMCDMEF	x1 M x3 x1'' → M x1 x2 x3 → x1''	MGH
5.	DEFGEFGH	x1 x2 x3 x4 → x1'	FGH
6.	QXAPXBQXA	x1 x2 x3 x4 x5 x6 → x1	PXB
7.	ADUACUAEUABUAF	x1 U A x4 x5 x6 'x1 U A → x4' U A x3 x4 x5 x6 U A → 'x3 A x2 x3 x4 x5 x6 A → x2' x1 U x3 x4 x5 x6 'x1 → U x1 x2 x3 x4 x5 x6 → x1	UAA
8.	MABMBCMCDM	M x2 x3 x4 x5 x6 M → x2'' x1 x2 M x4 x5 x6 x1'' x2'' → M x1 x2 x3 x4 x5 x6 x1'' → x2'' x1 x2 x3 x4 x5 x6 → x1	DEM
9.	URTUSTU	U x2 x3 U x2' → TTU x1 x2 x3 → x1	TTU
10.	ABYABXAB	B x2 x3 B → 'x2 x1 x2 x3 → x1	XAB
11.	RSCDSTDE	x1 x2 x3 x4 → x1'	TUE
12.	APAOQAPR	x1 A x3 x1' → A x1 x2 x3 → x1'	AQS
13.	WXAXYBY	x1 x2 x3 → x1'	ZCC
14.	JKQRKLRS	x1 x2 x3 x4 → x1'	LMS
15.	PONONMNN	x1 x2 x3 → 'x1	LMN

Figure 9. Rules Learned by Series Completion Production System

VI. CONCLUSION

The PAS-II system has been described and used to illustrate adaptive techniques in production system construction. The focus has been on the machinery needed to implement self-modification within a PS framework. It has been demonstrated that using a simple production building action in an ordered PS leads to relatively short, straightforward programs.

Moreover, it has been shown that one can create a learning paradigm which applies to (1) simple rote learning tasks such as learning the addition table, (2) more involved learning tasks like nonsense syllable association and discrimination, and (3) complex induction tasks such as inducing the concept of a serial pattern. In all three cases the paradigm consisted of creating an ordered PS representation of the concept learned by adding new PR's (or hypotheses) above the error-causing rules.

Adaptive PS's are quite parsimonious; that is, the system which learns the concept is represented in the same way as the concept being learned. Both are represented as PR's in a single PS. This eliminates the need for two types of control in the system; one for activating the learning mechanism and another for accessing the concept learned. The concepts learned are not passive, static structures which must be given a special interpretation, but rather are self-contained programs which are executed automatically in the course of executing the learning mechanism.

The ADD PS is somewhat different from the PS's for verbal learning or sequence prediction. This is because ADD is self-modifying but not really adaptive in the strict sense of the word. It creates new rules, not on the basis of external feedback, but rather on the basis of internal information, i.e., the ordering on the set of integers. Furthermore, rules are added only when needed to solve the problem at hand. This is a good example of an explicit view of predetermined developmental potential. The system has the capacity to develop the addition table or the successor function on integers but does so only when the environment demands it.

The EPAM and series completion PS's are extremely compact pieces of code which perform sizable amounts of information processing. Their power comes from the strong pattern matching capabilities inherent in the PS interpreter and from the primitive but highly useful memory modification and system building actions employed. The compactness is due, in part, to the use of ordered PR's, since much information concerning rule applicability is implicit in the location of the rules. With ordered rules the system can use the simple heuristic "add a new rule immediately above the one that made the error" to great advantage.

Finally, the analogy between an ordered PS and a discrimination net has been made clear, i.e., that the condition elements are non-terminal nodes in the net, the action elements are terminal nodes, and the searches through the conditions in the PS are analogous to the paths from the top element to the terminal elements in the net.

ACKNOWLEDGMENTS

The author thanks David Klahr, Dick Hayes, Herbert Simon, and Allen Newell for their suggestions concerning this paper. This work was supported by NIH MH-07722, and by ARPA (1-58200-8130).

REFERENCES

Feigenbaum, E.A. The simulation of verbal learning behavior. In Feigenbaum, E., and Feldman, J. (Eds.), Computers and Thought. McGraw-Hill, New York, 1963, pp. 297-309.

Feigenbaum, E.A., & Simon, H.A. An information processing theory of some effects of similarity, familiarization, and meaningfulness in verbal learning. J. Verbal Learning and Verbal Behavior, Vol. 3, 1964, pp. 385-396

Klahr, D. & Wallace, J.G. The development of serial completion strategies: An information processing analysis. British Journal of Psychology, Vol. 61, 1970, pp. 243-257.

Newell, A. A theoretical exploration of mechanisms for coding the stimulus. In Melton, A.W., & Marton, E. (Eds.), Coding Processes in Human Memory, Washington, D.C., Winston & Sons,

Newell, A. Production systems; Models of control struetures. Visual Information Processing, Chase, W. (Ed.), Academic Press, 1973.

Newell, A., & Simon, H.A. Human Problem Solving. Englewood Cliffs, N.J., Prentice Hall, 1972.

Simon, H.A., & Kotovsky, K. Human acquisition of concepts for sequential patterns. Psychological Review, Vol. 70, no. 6, 1963, pp. 534-546.

Waterman, D.A. Generalization learning techniques for automating the learning of heuristics. Artificial Intelligence, Vol. 1, nos. 15 & 2, pp. 121-170.

Waterman, D.A. PAS-II Reference Manual. Computer Science Department Report, CMU, June, 1973.

Waterman, D.A. Serial pattern acquisition: A production system approach. CIP Working Paper #286, CMU, February, 1975.

Waterman, D.A., & Newell, A. PAS-I1: An interactive task-free version of an automatic protocol analysis system. Proceedings of the Third IJCAI, 1973, pp. 431-445.

Williams, D.S. Computer program organization induced from problem examples. In Simon, H.A., & Siklossy, L. (Eds.), Representation and Meaning, Prentice Hall, Englewood Cliffs, N.J., 1972, pp. 143-205.