Learning procedures from examples and by doing

David M. Neves
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract

This paper describes a program that learns procedures b\ examining worked-out examples in a textbook and bv working problems I wo kinds of production *(If-then)* rules are created: *working forward* rules that produce an action when a proceduie is executed and difference rules that suggest operators from observed transformations. Dining example learning, the program examines two states in an example, fig tires out the operator that produced the second state and creates a production with some part ot the first line in the condition with the operatortor in the action. During learning b\ working problems, the progiam generates its own example trace b\ problem solving and uses the same example learning techniques.

Introduction

ALEX (Algebra example learner) is a program that learned to solve simple linear algebraic equations b\ examining worked-out examples and bv working test problems. The work (Neves, 1981) also included learning simple list algorithms, such as sorting and reversing a list. That work will not be reported here. An earliee version of the program was presented in Ne\es.(1978).

Two important characteristics of these textbook examples (/r/n conditions for learning, c.l. VanLehn, 1983) are that a. one procedure is taught per example and b. there are no hidden variables (however creating programs from examples or learning the commands of an operating system often requires the learner to induce variables that are not observable - Dieiterich, 1984).

The sub|ect mallei for learning comes fiom a high school textbook (Stem and Crabill, 1972) on elementary Algebia. The particular section of interest occurs after students are taught (in preceding chapters) exponents, the Distributive Rule, factoring, and tractions I hev then come to the first chapter in which the notion of solving an equation by manipulating symbols is introduced. I will describe the contents of this section to give an idea of what information is presented in this textbook, and which information ALEX uses to learn.

The chapter on equation (solving tries to give an understanding of what an equation is and what operations can be applied to it by using a seesaw analogy with the equal sign at the fulcrum. Only operations which preserve the balance of the seesaw can be applied to the equation. ALEX does not use this information. A problem with analogy here is that it is difficult to draw the correspondence between a weight on a seesaw, and an operation on an equation for the novice user. Which math operaiions preserve equation "weight"?

The next part of the chapter section teaches the legal operators of algebra by showing examples of how they work. These examples show how to add a number to both sides of an equation, how to subtract a number from both sides, how to multiply both sides by a number, and how to divide both sides by a number. The example for adding to both sides looks something like:

$x = 2$

$x + 5 = 2 + 5$

ALEX uses these examples to learn the legal operations in algebra.

Next, the book demonstrates the futility ot solving for the unknown bv randomly plugging in numbers. ALEX does not use this information

finally, the textbook presents a five step example that solves for the unknown (in the equation "3x - 4 = 0") using legal operations, such as adding a number to both sides of an equation. After this single example six test problems are given and this section ends.

The goal ot this research is to demonstrate how learning can take place by looking at the examples and b\ working the problems. Examples especially are well suited for procedute instruction as thev contain steps that will lead to the goal text (01 teacher) explanation, on the other hand, can be misleading, incomplete, 0r false. Example learning will not generally result in a complete procedure however. The student must be prepared to use problem solving along with incomplete procedures in working the problems at the end of a section. Working problems can produce learning behavior b\ having the student (0f program) generate an example trace through problem solving behaviou.

Basic ideas

Here I will describe some ot the basic ideas behind the ALEX program. A more detailed explanation follows this section.

While learning from an example AEEX looks at two consecutive equation lines at a time. The fust line is viewed as having some operation applied to it to yield the next line The task ot the learner is to figure out the operation that took place and why it was applied in that circumstance. Once the system has this knowledge a *working forward* production rule is built with some part of the first line as the condition and the operator as the action. When done with those two lines the pro gram goes to the next pair of example lines, continuing until the end of the example. At the end of the example the procedure is indexed with a difference production rule. The condition of this rule is made up of the difference between the first and last lines of the example. The action is the procedure name.

Procedure knowledge

Procedural knowledge in ALEX is encoded as production rules (Newell & Simon, 1972; Anderson, 1983). Rules in ALEX are coded in an older version (OPS3) of the OPS family of production languages (Forgv, 1979). I will use a less formal, more English-like presentation of productions in this paper. Productions have been a popular medium for learning since Waterman (1975) because ot their modularity. The unit of knowledge is the production. A newly created production is added to the set of productions already in production memory. There is no need to look at the other productions in the set. Contrast this with a programming language where the placement ot code has important implications for what action is produced.

Pioduction system languages, such as OPS. are flat. That is, anv production can apply at an\ point. To get some top-down control ol the system some method must be developed that *parimons* the production set so that only a few productions can apply The usual way this is done (as is done in ALEX) is to group the productions in subroutines (goals) by placing a goal name in the condition side ol the production When a subroutine of productions is to be called that name is placed in working memory.

In addition to partitioning there must be some method ol passing information to the subroutines and of passing back results. ALEX does this b\ emulating a traditional progiamming language control structure in its working memory. A subioutine call generates a new node in memory. Thai node has attached to n several pieces ol information: the name ol the subroutine, a list ot arguments, and the node ol the subroutine that called it. When a subioutine finishes it attaches the lesult to that subioutine node. e.g.

Call-145
name divide-both-sides
args (3x=2. 3)
calledby Call-144
result ?

Declarative knowledge

} acts are stored in working memory. During example learning the example is stored in working memory. An example is made up ol equation lines connected b\ a next relation. Each equation is represented as a tree structure (propositions) with the top node being the equation, the next level being the lelt side, the equal sign, and the right side; the level after that contains terms Each side can contain 1 or more terms connected by a next relation. For example 3\ 4 - 2 would have 3X before -4 on the lelt and a 2 on the hight

Simple example learning

This section presents some ol the basic ideas behind example learning without the complications described in the next section.

The production system has two kinds ol productions toi example learning. It has difference pioductions which are used dining the learning and it has working lorward productions which are the results of the learning process. A dilleience production has some information about the dilleience between two lines on the condition side ol the production and has an operator (which will cause that difference) on the action side. A working forward pioduction is part ol the subroutine ol productions. It is used when the subroutine is called, lor example, working forward pioductions to solve lor x are shown in Figure 1. These productions will move numbers from the left hand side to the right, will move terms with the unknown from the right side to the left, will combine like terms, and will halt when the value of the unknown is found.

When it is given an example the program starts at the first two lines and computes the difference between them. From this difference it determines which operation was applied to turn the first line into the second line. It then calculates why that operator was applied and creates a new working forward pioduction. After this it repeats the process on the next pair of lines in the example, until the end of the example is reached. At the end it creates a stopping production with the last line as the condition of the production rule (as in P5 above). These steps of example learning are called compute-difference, retrieve-operator, create-condition, and create-production below.

PI It there is ,a number on the left hand side of an equation, then subtract it from both sides.

P2. It there is a term with "x" in it on the right hand side, then sub tract it from hoth sides

P3. It there are two like temis on the left (or right) hand side, then combine them.

P4. II the equation is "num1 x = num2" then divide both sides by num I

PS. II the equation is "x = num1" then STOP

Figure I. A working forward production system to solve for v.

Computing the difference.

The fust thing ALEX does is lo compute the difference between two consecutive lines in the example. This dilleience is a list ol terms that are in the first line but not in the second (i.e. terms that have been removed) and a list ol terms that are in the second line but not in the list (i.e terms that have added). l'or example, the two lines below:

+ 4 : = 5
\ + 2 - 2 = 7

produce a dilleience ol (Add (2)) (Add (+ 7)) (Remove (+ 5)). The actual terms ol the dilleience (e.g. (2)) are nodes in a network representation ol the equation and contain othet information, such as whether they are belore or after the equal sign.

Retrieving the operator

When the dilleience is calculated it is plated in working memory Then the difference productions are called. These productions contain dilleience information in the condition side and supply an operator that produces that dilleience on the action side. One ol the pioductions matches the difference and returns an operator A dilleience production to recognize adding to both sides looks like.

131. 11 (Add (+ Num 1)) and Numl is on the left and (Add (+ Num2)) and Num2 is on the right and Numl equals Num2, Then the operator is add-to-both-sides and the argument to the operator is Numl.

If the two lines in the example are:

x - 5 = 10
x - 5 + 5 = 10 + 5

then production D1 (above) will fire and assert that the operator used was add-to-both-sides with an argument ol *5.*

Making the condition side

I will talk more about this stage later. The condition side ol the working forward pioduction to be created will contain some (or all) ol the first ol the two equation lines being looked at. In the least we must include the argument of the operator, plus some context information.

Making the working forward rule

Once we have an operator and condition side a working forward production can be built and stored in production memory. A sample rule:

RI. 11 (- Numl) is on the left side, then add-to-both-sides(Num1)

The shipping and difference rules

When ALEX reaches the last line of the example it creates a stopping production with the line as the condition side and the action ol returning from the current production system subroutine. It also computes the difference between the first and last lines and create a difference production with the difference as the condition and the operator name as the action.

Details of ALEX

The above description is a basic introduction to learning from examples. In actual practice there are complications, such as skipped steps, and inability to retrieve an operator after calculating the difference between two lines.

Figure 2 shows the top level of ALEX. Two lines are worked on at a time. In the description above, this step was described as if the lines were compared and an operator was retrieved. However, ALEX actually uses a means-ends subroutine.

T1. Start with the first 2 lines of the example.

T2. Represent both lines.

T3. Call means-ends(linel,line2).

T4a If not at the end of the example then look at the next two lines and goto T2.

T4b If at the end of the example then create a difference production with the difference of the first and last lines of the example as the condition, and the name of the routine being learned as the action. Stop.

Figure 2. The top-level of the example learner.

Means-Ends routine

The means-ends routine places problem solving capabilities within the learning program. This routine (shown in Figure 3) takes two lines as input. Its goal is to apply operators to the first (and successively generated lines) until the second line is reached.

The routine is modeled after the Newell, Shaw, and Simon (Newell and Simon, 1972) General Problem Solver (GPS) program. GPS is given operators, a table of connections (a table connecting differences to operators that produce those differences), a start and a goal state. ALEX has no table of connections, but instead uses difference productions which connect operators to a group of changes (the difference).

The means-ends routine below computes the difference between the two lines of the example and calls get-apply. Get-apply retrieves an operator (by using a difference production) that will reduce the difference between the two lines. After retrieving an operator get-apply applies the operator to the first line and returns the result (a new line for the first line). Now means-ends is called recursively with the new line to see if the goal line is reached. If this recursive call is successful (i.e. the goal line was reached) ALEX creates a working forward production with some part of the first line as the condition, and a call to the operator as the action. If the recursive call to means-ends fails then ALEX calls get-apply again to get and apply a different operator.

Inputtwo lines

M1 Compute the difference between the two lines.

M2a If no difference, return success.

M2b If there is a difference, call get-apply.

M3a If get-apply is successful call means-ends(new line, goal line).

M3b If get-apply failed and this is an example, then trv a simple transformation.

M3c If the simple transformation failed then return failure.

M3d If the simple transformation was successful then build a working forward production, build a difference production, return the new line.

M4a If means-ends failed, go to M2b (i.e. call get-applv again).

M4b If means-ends was successful then build a working forward production and return the new line.

Figure 3. The means-ends subroutine.

Skipped Steps

One major reason for using means-ends to work through an example is that the book might have skipped steps in its presentation. For example, suppose two steps in an example are 2x - 4 = 10 and 2x = 14. Several steps have been skipped. The full example is:

2x - 4 = 10
2x - 4 + 4 = 10 4- 4
2x + 0 = 10 + 4
2x = 1 0 + 4
2x = 14

The means-ends routine fills in the skipped steps by applying operators until the second line (2x = 14) is reached. These generated steps are used in the learning process just as though they were there in the original example.

The novice user must fill in the skipped steps. The expert algebra problem solver probably has an operatoi (add-to-both-sides-&-simplify) that goes directly from the first to the second line above. This is an important operator to have because it gives purpose to the add-to-both sides operator. Adding to both sides seems counter-productive because it takes the problem solver away from the goal of solving for the unknown by introducing two new terms. However, once simplification is done, it is seen that the operator is the first step in moving a term from the left to the right hand side of the equation. Skipped steps are a signal for the novice user (as they were to an earlier version of ALEX) to treat the skipped steps as a subroutine to be learned. This subroutine is indexed by a difference production created from the first and second lines of the example,

Creating new transformation

So far I have described get-apply as retrieving an operator and applying it to the line. Sometimes a new transformational operator is being learned, such as in the example teaching to add to both sides below.

x = 2, argument is 5
x + 5 = 2 + 5

In this example there is no operator to be recognized. Instead one is being taught. What ALEX does here (when it does not retrieve an operator) is to construct a new operator with calls to primitive symbol manipulating functions such as "insert-after". The operator created above is a single production that inserts its argument in the left side of the equation and in the right side of the equation. Also a difference

production is created so the operator will be recognized in the future.

ALEX only creates a new transformation operator when learning from an example. It is too dangerous to allow it to try to construct operators while problem solving because non-legal operators could be generated, Even when learning from examples it is possible to learn such operators. For example, ALEX could learn an operator (from 2x - 4 = 0 , 2x = 0 + 4) that moves a number from the left to the right, inverting its sign. It could also learn to delete equal terms on opposite sides of the equal sign from (3-4 = 5\ - 4, 3 = 5x) These are operators that can be proven valid mathematically, however their use may mask ignorance.

Creating he Condition side

Cieatmg the condition side of the production is difficult, and ALEX has some problems here. Learning from examples is often associated with concept learning. Anderson, Kline, & Beasley (1980) call this of a production by applying it to different situations. 11 an operator is seen to apply in several different situations a general condition side can be constructed. If a production fites when it should not have fired, then it is punished and clauses have to be added to the condition to make it more specific. These tuning capabilities can take some time. ALEX does not have them Instead, ALEX has several heuristics which do a good, but not perfect, |ob in cieatmg condition sides for productions. These are described below

Working forward productions call an operator with some arguments. The condition side of the working forward production must contain those arguments. The condition creator does a breadth-first search from the equation node of the first of the two lines looking for the argument. When it finds the argument it places it, along with some of its context (the path from the equation node to the argument) in the condition side. This procedure is outlined below.

1. Find a path from the current line to the argument. This path will be included in the production. Often this path leads from the equation node to a term within the equation. Sometimes the argument may occur elsewhere in the example (such as at the start of the example in the adding-to-both sides example). The program now tries to specify enough of the path to make it *unique,* i.e. to distinguish this path from other potential paths from the equation node.

   a. If there is no other path ol this length (from the current equation node to the argument) then quit.
   b. Otherwise check to see if the type (i.e. ISA property) of the argument is unique. If so, include the ISA property in the condition and quit
   c. Otherwise, check to see if the argument is in a special location in its list (i e. first location, last). If so, include that information and quit.
   d. Otherwise, put in absolute location information.
2. Note which nodes in the network are equal and include this information in the condition.
3. Add any constraints of the operator to the condition. Each operator needs a certain state configuration to operate successfully, litis information is included in the difference production for that operator and is added to the working forward production being built. For example, canceling two terms requires that thev be equal but of opposite signs (+ 5 + -5).
4. Generalize the condition by deleting all constants.

The performance system

ALEX uses two methods to work problems in the textbook. When it is given a problem to solve it first tries to use its working forward productions to solve the problem If the goal is reached then the problem is solved and the system halts. It the working forward rules are not sufficient to solve all the problem it will halt at some point, short of the goal Here the problem solving (means ends) component takes over and suggests an operator to try next. If this operator leads to the goal then a working forward production is built so that problem solving will not have to be done in the future. ALEX starts out with slow problem solving behavior, gradually working its wav to fast expert (working forward) behavioi with experience in working problems The perfor mance system will be discussed in more detail later

Traces of algebra learning

This section describes how ALEX dealt with the 5 examples in the textbook.

Learning to add to both sides

ALEX first learns the legal Algebra operations, such as adding a number to both sides. It is given an example like:

```
x = 2 (argument is 5)
x + 5 = 2 + 5
```

It looks at the two lines and notes that two (+ 5)'s have been introduced. It does not have an operator that will do this transformation so it makes one with its primitive list operations. It creates a single production procedure A1.

```
A1. If the goal is add-to-both-sides
      arg1 is -equation, arg2 is =loadd
      = Lterm is a left side term
      = Rterm is a right side term
    Then insert (+ =toadd) after = Lterm
         insert ( + =loadd) aftei = Rteim
         return from add-to-both-sides
```

ALEX also creates a difference production (1)1) with the difference in the condition side and the add-to-both-sides name in the action side.

```
D1. If the goal is get-action
      (Add (+ =Lterm)) (Add (+ =Rterm))
      ( = Lterm equals = Rterm)
      = Lterm is a left side term, -Rterm is a right side term
    Then the operator is add-to both-sides
         the argument is =Lterm
```

Learning to divide both sides

ALEX next learns how to divide both sides by a number with the following example.

```
x = 2 (argument is 3)
x / 3 = 2 / 3
```

No operator is found that will transform the first line to the second so the system must construct an operator (A2).

```
A2. If the goal is divide-both-sides
      arg2 is =div
      = Lterm is a left side term
      = Rterm is a right side term
    Then replace =Lterm with (/ "Lterm =div)
         replace = Rterm with (/ =Rterm -div)
         return from divide-both-sides
```

The problem with production A2 is thai it is dividing terms on both sides and not the sides themselves. The example is ambiguous because there is only one term on a side. The production will not generate an error until it is applied to a side with more than one term. In that case it will divide just one term, as in:

```
3x - 4 = 5      ;divide by 3
3x/3 - 4 = 5/3
```

It turns out this is an error generated by novice Algebra solveis as well (Davis & Cooney, 1978).

The difference production created for dividing both sides (D2) shows some propositions (under consitions) on the action side. These are pio positions that are needed in order for the operator to successfully apply (i.e. in order to divide you need something to divide). When a working forward production is created from this rule these propositions are added to the condition side of that newly created rule. This is step 3 of the condition creation routine explained above.

```
1>2. II the goal is get-action
     (Add (/ =LtermR =div)) (Add (/ =RtermR =div))
     (Remove =LtermR) (Remove =RtermR)
     =LtermR is a left side term
     =RtermR is a right side term
   Then the operator is divide-both-sides
     the argument is =div
     the conditions are
          =LtermR is a left side term
          =RtermR is a right side term
```

ALEX also learns how to subtract both sides of an equation by a number and how to multiply both sides b\ a number. These productions are similar to the ones above for addition and division.

Learinig to solve for \

ALEX now knows the legal operators of Algebra. It next is given an example to learn *when* to use these operators in solving for the unknown.

```
1. 3x + 4 = 6
2. 3x + 4 - 4 = 6 - 4
3. 3x = 6 - 4
4. 3x = 2
5. 3x/3 = 2/3
6. x = 2/3
```

Lines J and 2 are processed first. The difference production for subtract-both-sides fires, suggesting a number was subtracted from both sides of the equation. The operator is applied to line 1 and line 2 is leached. The hypothesis is confirmed. To compute the condition side it searches the first line for a "4", the argument to the operator it found. It finds 4 on the left side of the equation and builds A4.

```
A4. If the goal is solve-for-x
     =Lterm is (+ =number1)
     =Lterm is a left side term
   Then subtract-both-sides (- =number1)
```

ALEX has learned *when* to use subtract-both-sides when solving for x.

ALEX now examines lines 2 and 3. A difference production for delete-terms fires. This difference production was learned in previous chapters as part of instruction on simplification (it is given to the ALEX program). Delete-terms takes as input two equal terms, but having different signs, and deletes them. A5 is built from lines 2 and 3.:

```
A5. If the goal is solve-for-x
     =Lterm1 is (+ =thing1)
     =Lterm1 is a left side term
     =Lterm2 is (- =thing1)
     =Lterm2 is a left side term
   Then delete-terms(=Lterm1, =Lterm2)
```

Note here that the production is a bit loo specific. It will not f it two like terms are on the right hand side of the equation.

Lines 3 and 4 show another arithmetic operation in progress. A production that is veiy similar to A.5 above is created

Lines 4 and 5 demolish ate when to divide both sides by a number. A difference production for divide-both-sides fues and A6 is built.

```
A6. If the goal is solve-for x
     =Lterm is (+ =num1 * x)
     =Lterm is a left side term
     there is a term on the left and a term in on the right
          ;dividing both sides requites there be something on the
          ;right hand side to divide
   Then divide-both-sides( + =num1)
```

The problem with A6 is that it is too general. There is a possibility that the rule could fire if there were moie than I term in either side of the equation.

The last two lines of the example suggest another simplification operator. Two numbei s are canceled. Production A7 is built.

```
A7. If the goal is solve-for-x
     =Lterm is (=t1 * =t2) / =t1
     =Lterm is is a left side term
   Then cancel(=Lterm)
```

After finishing this example ALEX has 6 (including the stopping pioduction) working forward productions that will solve the same problem shown to it as well as other similai problems. In addition it has the 4 working forward pioductions which are legal operations in Algebra and their 4 ditfeience productions. Solve lor x also has a difference pioduction associated with it.

Learning b\ doing

After going through the example ALEX is given six problems to solve. These are: a. 6x - 5 = 0 b. x/7 + 4 = 0 c. 4x = 7 d. 5x + 7 = 0 e. x/2 = 3/8 f. 1/x = 2. Some of these problems can be solved with just the working forward productions built from the example. Others will require some problem solving. During that problem solving new learning can take place.

Problem c is very similar to the example and is solvable with the existing productions. Both sides of the equation are divided by 4, then the two 4's on the left are canceled to yield x = 7/4.

Problem d is very nearly solved also. The problem comes with the too general production that divides both sides by a number. For the first step either both sides can be divided by 5 or 7 can be subtracted from both sides. Because OPS production system conflict resolution prefers productions with larger condition sides the incorrect (division) pioduction is picked. If the subtracting-from-both-sides routine is executed (i.e. after backing up from the other production application) the problem is solved. The division production needs some discrimination tuning (Anderson, Kline, & Beasley, 1980) which would compare the problem state during a successful invocation of the production with the failure state. See the discussion at the end of the paper.

Problem a is the first which requires some problem solving. It is similar to the example except that a number is subtracted on the left side instead of being added. There is no production that fires, so the system enters into problem solving. ALEX now tries to reduce the difference between 6x - 5 = 0 and the goal (x = number). One difference is that the 5 needs to be deleted on the left. An operator is retrieved (add inverse) that will do that but it is not a legal Algebra operator. A legal operator is searched for that is similar to add-inverse and add-to both-sides is found and returned.

Note that the process here is somewhat different from the table of connections in GPS. The difference is that the operator retrieved might not be legal in the domain being worked on. The reason a non legal operator was found above was that the operator was sensitive to the changes it causes - i.e. one can get rid of a number by adding its inverse. This knowledge came about through training in simplification (perhaps even when steps were skipped). The Algebra operators (adding-to both-sides, etc.) have not yet been indexed by the changes they ultimately produce (i.e. moving a term from one side to another).

Since adding-to both-sides was retrieved during problem solving a new production is built that will apply add-to-both sides if there is a negative number on the left hand side of the equation. Problem a is finished up by dividing both sides by 6 and simplifying.

Problem b is x/7 + 4 = 0. The program starts out by subtracting 4 from both sides and simplifying. It is now at x/7 = -4 and starts to problem solve. ALEX compares its current state to the goal and sees that it should get rid of the 7. As in the previous problem it retrieves an inverse operator that will do it (to multiply by 7) but it isn't legal in Algebra. It then searches its memory for a legal Algebra operator that is similar and finds the operator multiply both-sides. It successfully carries this out and builds the working forward production "If x/n1 = n2 Then multiply both sides by n1".

Problem e (x/2 = 3/8) demonstrates what is gained in learning by doing. If no other test problems had been done before getting to e then some problem solving behavior would have been needed to solve it. However solving problem b has created a production that can be used in this problem. The first thing done is to fire a production, created in Problem b, that multiplies both sides by 2. Then simplification is done to give x = 6/8.

Problem f (1/x = 2) is too hard for the problem solving system. It starts out by multiplying both sides by x^2 to get x on the left hand side. After simplification it has $x = 2x^2$. It cannot get any closer to the goal from here. Later on in this chapter a whole section is devoted to this kind of problem. The suggested solution is:

1/x = 2
x * 1/x = 2x
1 = 2x
1/2 = x or x = 1/2

ALEX would have to wait for such an example to be able to solve it.

Summary

ALEX learned to solve simple equations from two pages in a textbook. It first learned the torn legal operators by looking at examples showing what they do. ALEX then learned when to use those operators (as well as when to use simplification operators) by going through an example solving for x. ALEX was then given six problems to solve. Two of the six problems can be solved with the working forward productions built during example learning.

ALEX needed to use problem solving to solve two of the other three problems. As it found an operator that would get it to the goal it created a working forward production so that it would not have to problem solve again in similar situations. The third problem used a production built during a previous problem and did not need problem solving.

The last problem (1/x = 2) was too hard for ALEX. The solution involves some steps that take one away from the goal of solving for x. This problem is difficult for human beginners as well.

Problems

Creating the condition side

There were both good and bad points on the development of the condition side of the productions. The worst part was that there was no facility for changing the condition side given feedback on how the production did. This resulted in many productions that were too specific and a couple that were too general. One way to correct the general rules is to use a discrimination procedure (Anderson, 1983) on the general rule. In order for discrimination to work the aberrant rule must be first be identified. One way of identifying the aberrant rule is to use a check procedure (substitute the correct answer for the unknown and see where equation equality is violated). Unfortunately, checking is not taught until the next section of the equation chapter. The rule can also be identified by comparing its application with one in an example. If it is fired in a very different situation then it is a candidate for change. In the case of dividing both sides above, the rule flies when there are two terms on the left hand side. This looks different from the equation in the example which has a single term on the left.

Once a bad rule has been identified it must be changed by comparing the situation of incorrect application with a situation of correct application. In our case we have an instance of correct application in the worked-out example. If we compare the example of dividing both sides (3x = 2 to 3x/3 = 2/3) with the incorrect application (5x + 7 = 0 in Problem d) we notice that the left side has an extra term (the 7). We change our rule (An) by adding that constiaint (see the proposition in italics below).

A6' If the goal is solve-for x
= Lterm is (+ = num1 * x)
= Lterm is a left side term
there is a term on the left and a term on the right
there is only one left hand side term
Then divide-both-sides(+ = num1)

The best part of how condition creation was done was that the program did not need many examples in order to form rules that would work. The textbook only provides a single example of solving an equation in this section. It is not unreasonable to expect that students (or program) have learned something about generalization in the mathematics domain before coming into equation solving. Experience with simplification shows that particular number constants are not important and neither are terms not directly involved in the simplification.

Goal-sensitive differences

During problem solving ALEX made use of the fact that arithmetic operators were sensitive to the changes they eventually produced. For example, to get rid of a number one can add its inverse. ALEX can learn to index operators this way it skipped steps are used. Another way of learning to index operators in this manner is to observe how well each operator gets one closer to the goal when either going through an example or problem solving. It an operator takes one away

(torn the goal (such as adding to both sides ol an equation) then follow the example until at a state that is closet to the goal. The is analogous to an inquisitive student saying "Why was that operator applied? It doesn't seem to help get me to the goal." As the example is followed one of the steps will be closer to the goal than the original step (see the skipped steps example above for such a sequence). The original operator is then indexed by the change it eventually brings about.

Conclusion

One of the main results ol this research is to show how learning, problem solving, and performance can be combined into a single system. Learning from examples creates working forward productions that can be used in the performance system. While learning one must sometimes problem solve to fill in skipped steps. Also, as new procedures are being learned they are indexed by their changes (i.e. a difference production is created). This means that the learning system can then recognize more complex examples and the problem solving system can bring to bear high level operators to work on difficult problems.

Footnote

Special thanks go my Ph.D. committee at Carnegie-Mellon (H.A. Simon, chairman; D. Klahr, J. Greeno, and J. Larkin) for all their help in the area of learning and instruction as well as encouragement throughout. Thanks also to Jola Jakimik and to an anonymous reviewer. The ALEX program was run on a KL10 computer and was made up of about 70 OPS productions and 25 pages of UCI Lisp code.

References

Anderson, J.R. T*he architecture of cognition.* Cambridge, MA: Harvard University Press, 1983.

Anderson, J.R., Kline, P., & Beasley, C M. Complex learning processes. In R.E. Snow, P.A. Federico, and W.E. Montague (Eds.), *Aptitude, learning, and instruction. Cognitive process analyses.* Hillsdale, NJ: Lawrence Erlbaum, 1980.

Davis, E.J. & Cooney, T.J. Identifying errors in solving certain linear equations: Some findings and some suggestions. AERA paper, 1978.

Dietierich, T.G. Learning about systems that contain state variables. *Proceeding of the 1984 AAAI,* 96-100.

Forgy, C. The OPS4 Reference Manual. Department of Computer Science, Carnegie-Mellon University, 1979.

Neves, D.M., A computer program that acquires algebraic procedures by examining examples and by working on test problems. Proceedings *ol the Second National Conference of the Canadian Society (or Computational Studies ol Intelligence,* 1978, 191-195.

Neves, D.M., Learning procedures from examples. Ph.D. Dissertation, Department of Psychology, Carnegie-Mellon University, 1981.

Newell, A. & Simon, H. *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall, 1972.

Stein, S.K., & Crabill, C D. *Elementary algebra. A guided inquiry.* Boston, Houghton Mifflin Company, 1972.

Waterman, D.A. Adaptive production systems. *Proceeding of the Fourth IJCAI,* 1975. 296-303.

VanLehn, K., Felicity conditions for human skill acquisition: validating an Al-based theory. Rep. No., CIS-21, Xerox Palo Alto Research Center, 1983.