Episodic Learning

Dennis F. Kibler
Bruce W. Porter

Technical Report 194

May 1983

Department of Information and Computer Science
University of California, Irvine

This research was supported by
contract N00123-81-C-1165 from the
Naval Ocean Systems Center.

ABSTRACT

A system is described which learns to compose sequences of operators into episodes for problem solving. The system incrementally learns when and why operators are applied. Episodes are segmented so that they are generalizable and reusable. The idea of augmenting the instance language with higher level concepts is introduced. The technique of perturbation is described for discovering the essential features for a rule with minimal teacher guidance. The approach is applied to the domain of solving simultaneous linear equations.

Keywords: machine learning, episodic segmentation, perturbation, augmentation, problem solving.

1. Introduction

With the aid of a teacher, junior high school students can learn to solve simultaneous linear equations. Operators that are applied in solving these problems include multiplying an equation by a constant and combining like terms. The students are already familiar with how these operators are applied. Moreover, the teacher assumes that the students understand basic concepts about numbers, such as a number being positive, negative, or non-zero.

Our system, nicknamed PET, incrementally induces correct rules from the training instances presented. Incremental learning is defined to be [7]:

> multistage learning, in which information learned at one stage is modified to accomodate new facts provided in subsequent stages.

In PET this includes modification of existing knowledge by adding more rules as well as generalizing existing rules. The rules are correct in the sense that, at any point in the learning process:

- the knowledge is consistent with all past training instances.
- sequences of rules (episodes) are guaranteed to simplify the problem state if they apply.

Learning rules for applying operators involves two stages of learning:

- Stage 1 learning involves understanding when each available operator should be applied. The concern here is with learning the enabling conditions for individual operators, without knowledge of the other operators in the solution path to provide context.
- Stage 2 learning involves understanding why each operator is applied with emphasis on the sequencing of operators. We refer to this as episodic learning. Episodic segmentation is the grouping of operators to form an episode. Episodes are discrete, reusable components for plan generation and each simplifies the problem state.

We developed an approach to stage 1 learning which solved some of the problems of learning when to apply operators. Basically, we reduced the size

of the generalization space for learning when an operator should be applied, while not constraining the order of the training instances presented by the teacher. PET could create its own training instances by perturbing those given by the teacher. Each perturbation is a minor variant of the original and allows PET to discover the set of features which are essential for a given operator to succeed.

Stage 2 learning builds sequences of rules that are used to move from the initial state of a problem to a goal state. Again, a major concern is reducing the size of the search space. A traditional problem solver [4] simply explores the search space for a path to a goal state. However, a learning system substitutes knowledge derived from experience for this search.

The main features of our approach to episodic learning are:

- segmentation of rule sequences into meaningful, re-usable episodes.
- augmentation of the instance language to include higher-order concepts not present in the training instance itself.
- perturbation of a training instance to create new instances.

Stage 1 learning (incorporating perturbation) and stage 2 learning of episodes are combined into a system for learning to solve systems of simultaneous linear equations from examples.[1]

2. Related Work

Our knowledge representation scheme and goals are similar to those of other researchers. We use a relational production system, somewhat like Vere's [16] except that we use a bag of conditions rather than a set, to represent the

---

[1] implemented in Prolog on Dec2020. Available upon request.

program's knowledge of when to apply operators. Production systems have been successfully used to model the acquisition of skill for poker playing [17], puzzle solving [1], algebra problems [13], arithmetic problems [2], and symbolic integration [10]. Of these, Neves's [13] system learned to solve one equation in one unknown from textbook traces. The system learned both the context (preconditions) of an operator as well as which operator was applied, although the operator had to be known to the system. His generalization language was simpler than ours in that a constant could only be generalized to a variable. Anzai [1] gradually refined weak general problem solving methods into strong ones by acquiring strategies for the tower-of-Hanoi problem. Weak methods, without some heuristics, would leave our program with too large a space to search. The program LEX [9, 10, 11, 12] uses version spaces to describe the current hypothesis space as well as concept trees to direct or bias the generalizations. As it is not the main point of our work, we keep only the minimal (maximally specific) generalization [14] of the examples.

MACROPS [4] is an example of stage 2, or episodic learning. This system remembers robot plans that have been generated so that the plan can be reused without re-generation. The plans are stored in triangle tables which record the order of application of operators in the plan and how their pre-conditions are satisfied. The plans are generalized to be applicable to other instances (as are episodes).

While effective in learning plans, MACROPS has difficulty applying its acquired knowledge [3]. The central problem is that the operators in a MACROPS plan are not segmented into meaningful sequences. Any sequence of operators can be extracted from the triangle table and re-used as a macro

operator. A sequence of length N defines N(N-1)/2 macros. However, few of these sequences are useful. MACROPS offers no assistance in selecting the useful sequences from a plan. If sequences are not extracted from the triangle table then the entire plan must be considered an episode. This results in a large collection of opaque, single-purpose, macro operators. Branching within an episode is made impossible. In either case, combinatorial explosion makes planning with the macros impractical.[2]

3. Operators for Solving Linear Equations

The operators applicable to solving simultaneous linear equations are described in figure 3-1.

Operator	Semantics
combinex(Eq)	Combine x-terms in equation Eq.
combiney(Eq)	Combine y-terms in equation Eq.
combinec(Eq)	Combine constant terms in equation Eq.
deletezero(Eq)	Delete term with 0 coefficient or 0 constant from equation Eq.
sub(Eq1,Eq2)	Replace Eq2 by the result of subtracting Eq1 from Eq2
add(Eq1,Eq2)	Replace Eq2 by the result of adding Eq1 and Eq2
mult(Eq,N)	Replace equation Eq by the result of multiplying Eq by N

Figure 3-1: Operators

4. Description Languages

This section describes three languages used by PET: the instance language, the generalization language and the rule language.

---

[2]It should be recognized that MACROPS was designed to control a physical robot, not a simulation. For this reason, the designers thought it important to permit the planner to skip ahead in a plan if situation permits or to repeat a step in a plan if the operation failed due to physical difficulties.

4.1. Instance Language

The instance language serves as "internal form" for training instances. We adopt a relational description of each equation, so the training instance:

```
a: 2x-5y=-1
b: 3x+4y=10
```

is stored as:

```
{term(a,2*x),term(a,-5*y),term(a,1),
 term(b,3*x),term(b,4*y),term(b,-10)}
```

where a and b are equation labels and x and y are variables in the instance language.

4.2. Generalization Language

Following Mitchell [10] and Michalski [6, 8] we have concept trees for integers, equation labels, and variables (figure 4-1). Basically we are using the typed variables of Michalski [6].

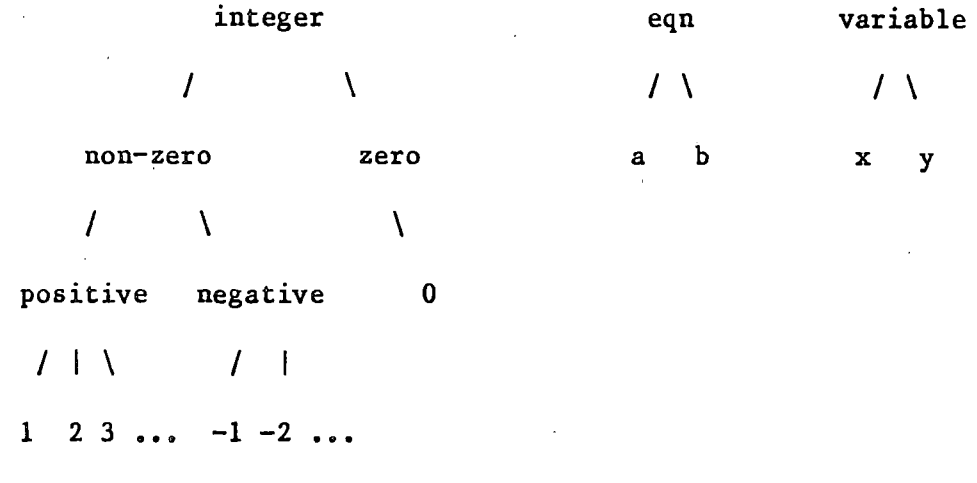


Figure 4-1: Concept trees

We permit generalizations by 1) deleting conditions, 2) replacing constants by variables (typed), and 3) climbing tree generalization. Disjunctive generalization is allowed by adding additional productions or rules. This

covers all the generalization rules discussed by Michalski [6] except for closed interval generalization.

To be more specific, generalizations of equation(a) are achieved by generalizing any term according to its concept tree or by deleting any term. term(a,2*x) has two generalizations of a (a and eqn(X)), four generalizations of 2 (2, positive(N), nonzero(N), and integer(N)), and two generalizations of x (x and var(Y)), giving a total of 16 possible generalizations. Two equations may have four such terms as well as two constant terms, yielding a total of 16*16*16*16*4*4 or more than a million possible generalizations! Note we have not counted the additional generalizations that come abut by deleting terms.

4.3. Rule Language

Knowledge is encoded in rules which suggest operators to apply. Rules are of the form:
<score>--<bag of terms expressed in generalization language> => <operator>
(The score of a rule is described in section 6.2.) PET does not learn "negative" rules to prune the search tree, as in [5].

5. Perturbation

Perturbation is a technique for stage 1 learning which enables a learning system to discover the essential features of a rule with minimal teacher involvement. A perturbation of a training instance is created by:

- deleting a feature of the instance to determine whether its presence is essential.
- if a feature is essential, modifying it slightly to determine if it can be generalized. Perturbation operators, which are added to the concept tree used for generalization, make these minor modifications.

For example, given the problem (a) 2x+3y=7 (b) 2x+3x-5y=5, the teacher advice to combinex(b), and an empty rule base, PET first describes the rule

as:

```
{term(a,2*x),term(a,3*y),term(a,-7),
 term(b,2*x),term(b,3*x),term(b,-5*y),term(b,-5)} => combinex(b).
```

Now PET perturbs the instance by modifying each of the coefficients individually. This is done by zeroing, incrementing and decrementing each coefficient. Some of the instances created by perturbation are:

(i)	(ii)	(iii)	(iv)	(v)
3y=7	2x =7	2x+3y=7	2x+3y=0	2x+3y=7
2x+3x-5y=5	2x+3x-5y=5	3x-5y=5	2x+3x-5y=5	2x+4x-5y=5

Notice that combinex(b) is still effective in examples i,ii,iv, and v but is not effective in example iii. By effective we mean that not only is the operator applicable, but also that it simplifies the problem state. Since the operator is effective in example i, PET generalizes (minimally) its current rule conditions with this example yielding the new rule:

```
{term(a,3*y),term(a,-7),
 term(b,2*x),term(b,3*x),term(b,-5*y),term(b,-5)} => combinex(b).
```

The major effect is to delete the condition on the x-term of equation(a). Perturbed examples for which the operator is not effective are disregarded. In other domains this negative information might be useful, but it is not necessary for this domain. Skipping ahead and generalizing with example v, the rule becomes:

```
{term(a,3*y),term(a,-7),
 term(b,2*x),term(b,pos(N)*x),term(b,-5*y),term(b,-5)} => combinex(b).
```

And after all positive instances of the operator combinex(b) have been generated by the perturbation technique and generalized, the rule is formed:

```
{term(b,pos(N)*x),term(b,pos(M)*x)} => combinex(b).
```

Essentially, perturbation is a technique for creating near-examples and near-misses [18] with minimal teacher involvement upon which standard generalization techniques can be applied.

6. Episodic Learning

6.1. Importance for Learning

As we noted in the MACROPS use of operator sequences, unless the system can select meaningfully useful sequences from the set of candidate sequences, combinatorial explosion makes re-use of generalized plans infeasible. To be useful, a model of episodic learning must have a clear definition of "episode." We define an episode to be a sequence of rules which, when applied, simplifies a problem state. Our episodes are "loosely packaged" to allow branching. Rather than storing an entire plan for reaching a goal state from the start state, we segment the solution path into small, re-usable, generalizable episodes, each accomplishing a simplification of the problem.

Episodic, or stage 2, learning is concerned with these sequences of rules and their connections. These sequences are learned incrementally. Learning a rule for an operator depends on an understanding of why the operator is applied.[3] PET understands two reasons for selecting an operator:

1. By applying the operator, the problem state is simplified. In the domain of algebra problems, a state is simplified if the number of terms in the equations is reduced.

2. By applying the operator, the preconditions of an existing rule are satisfied. The rule being formed for the operator is then loosely linked with the rule that the operator enables. If more than one rule is enabled, then multiple branches through the episode are allowed.

To understand (2), note that problem states in the solution path for algebra problems (and other domains) do not monotonically improve. That is,

---

[3] "Understand" is used here to mean "know-how" encoded in production rules. We do not mean to infer any deep model of understanding which might include causality and analogy.

with any natural definition of "simpler" (estimate of distance to a goal), PET cannot select a rule to apply. For example, multiplying an equation by a constant does not simplify a problem state, even though it is a necessary operation. Only by learning the connection between rules and their grouping into episodes can a student learn to solve algebra problems.

PET adds a rule to the rulebase when the purpose of the rule's action (the "why" component of the operator) is understood. Initially PET does not have rules for any operators. The first operators for which rules can be learned are those which simplify the problem state, such as combining like terms. Any operators applied before combine cannot be understood and PET must "bear with" the teacher. After rules are formed for the combine operators, subtract can be learned. For instance, sub(a,b) applied to:

```
        a: 2x+3y=5
        b: 2x-1y=1
yields:
        a: 2x+3y=5
        b: 2x-2x -1y-3y=1-5
```

Now PET can learn sub(a,b) for reason (2) above: a rule which is already understood (for a combine operator) is enabled by subtract. An episode can be formed connecting the rules for sub(a,b) and combine(b).

6.2. Scoring Operators

A simple scoring scheme connects rules into episodes and resolves conflicts when more than one rule is enabled. A natural scheme is to score each rule by its position in an episode. The rules for the combine operators are given a score of 0. The rule for subtract, which enables a combine operator, is given a score of 1 (0+1). Intuitively, the score is the length of the episode before something good happens (i.e. the equations get simplified). For conflict resolution, PET selects the rule with the lowest score among those

enabled. Ties are resolved arbitrarily.

Scoring is also used for perturbation. For each instance generated, PET must determine whether it is positive or negative for the operator being learned. For this determination, and to assign scores to new rules, PET considers the following three cases:

1. The rule is assigned a score of 0 if, when applied to the instance given by the teacher, the number of terms in the equations is reduced. In this case a perturbation is a positive instance if and only if the rule can be applied and it simplifies the instance.

2. The rule is assigned a score of 1 if, when applied to the instance given by the teacher, it enables a score 0 rule. In this case a perturbation is a positive instance if and only if the rule can be applied and, after some sequence of score 0 operators are applied, the instance is simplified.

3. The rule is assigned a score of N(>1) if, when applied to the instance given by the teacher, it enables a score N-1 rule. In this case a perturbation is a positive instance if and only if the rule can be applied and it enables the same score N-1 rule[4].

7. Augmentation

A description in the instance language is basically a translation of a training instance. This description is more appropriate to computation than the surface language used to input the instance. This is adequate for learning in some domains. That is, the relevant objects and operations in a training instance necessary for learning it are retained in the instance language. However, in more complex domains, more knowledge needs to be represented than is captured in a literal translation of a training instance into the instance language.

---

[4]The LEX system [12] must re-start the problem solver for each training instance created by the problem generator to determine if it is positive or negative for the concept.

Augmentation of the instance language with additional knowledge is useful in these complex domains. In the domain of backgammon, for example, merely recording the location of the pieces on the board in the instance language is inadequate. In addition, we need to have knowledge of pip count, presence of primes, near primes and steppingstones, etc. These higher order concepts are computable from the instance language and form a more appropriate language for learning in the domain.

In the domain of algebra problems, augmentation serves to relate terms in the instance language. For example, a relevant relation between coefficients is productof(N,M,P) (the product of N and M is P). This relation augments the instance language. The augmentation represents necessary pieces of knowledge (not available at the surface level of the training instance) which a student must have in order to solve problems.

Augmentation of the instance language is necessary when the terms or values necessary for an operation (the RHS of a rule) are not present in the pre-conditions for the operation (the LHS of the rule). For example, the training instance:

a: 2x-5y=-1
b: 3x+4y=10

might be presented with teacher advice mult(a,3). This yields:

a: 6x-15y=-3
b: 3x+4y=10

From this training instance, PET forms the rule (after perturbation):

{term(a,2*x),term(b,3*x)} => mult(a,3).

Here the 3 in the RHS operation mult(a,3) appears on the LHS in term(b,3*x). In this case, we say that the LHS of the rule is predictive of the operator on the RHS and no augmentation is needed.

In contrast, the teacher advice to apply mult(b,2) to the last pair of equations cannot generate a predictive rule. The operation is useful and yields:

a: 6x-15y=-3
b: 6x+8y=20

The problem is that the 2 in the RHS operation mult(b,2) is not contained in the instance language description of the equations. Therefore, it could not be on the LHS of any rule in this language.

An augmentation of the instance language is needed to relate the 2 on the RHS with some term on the LHS. In this case, the additional knowledge needed is the 3-ary predicate productof, specifically productof(2,3,6). Now the rule to cover the training instance can be formed:

{term(a,6*x),term(b,3*x),productof(2,3,6)} => mult(a,2)

This can be generalized (with more training instances) to:

{term(a,N*x),term(b,M*x),productof(L,N,M)} => mult(a,L).

Concepts in the augmentation language form a second-order search space for generalizing to the correct rule for an operator. Figure 7-1 is the augmentation search space for algebra problems. The space consists of a (partial) list of concepts that a student might rely on for understanding relations between numbers. When a predictive rule cannot be found in the first-order search space then PET tries to form a rule using the augmentation as well. Concepts are pulled from the list and added to a developing rule. If the concept makes the rule predictive, then it is retained. Otherwise, it is removed and another concept is tried. If no predictive rule can be found then PET ignores the training instance.

Vere [15] has also addressed the problem of learning in the presence of

```
sumof(L,M,N)          (sum of L and M is N)
productof(L,M,N)      (product of L and M is N)
squareof(M,N)         (square of M is N)
```

Figure 7-1: augmentation search space

"background information." For example, learning a general rule for a straight in a poker hand requires knowledge of the next number in sequence. This is considered background to the knowledge in the poker domain. Vere describes an "association chain" which links together each term in a rule. If a term in the rule is not linked in the chain (analogous to our test for predictiveness), then more background information must be "pulled in" until it is associated.

Augmentation is similar to selecting background knowledge. One problem with both approaches is determining how much background knowledge to incorporate. Incorporating too little knowledge, which results in an over-generalized rule, can be detected by an association chain violation or, in PET, by a non-predictive rule. However, detecting when too much knowledge has been pulled in is difficult. In this case, the rule formed will be over-specialized. We overcome this problem (to a large extent) by perturbation. Vere relies solely on forming a disjunction of rules (each overly specialized) for the correct generalization.

Vere allows only one concept in the background knowledge. This further simplifies the task of knowing how much knowledge to pull in. However, as the complexity of problem domains increase, more background knowledge must be brought to bear. Our augmentation addresses some of the problems of managing this knowledge.

8. The Learning Cycle

The learning cycle algorithm is described in figures 8-1 - 8-5. The rulebase is initially empty and, as PET learns, rules are added, generalized, and supplanted. PET requests advice whenever the current rules do not apply to the problem state. Both stage 1 and stage 2 learning is required.

```
repeat
  get problem from teacher
  repeat
    if some rule matches problem then apply_episode (no learning)
      else get operation from user and
       if understand_why_operation_used then
           call: learn_operation
       else no learning
  until problem solved
  display current set of rules
until teacher satisfied
```

Figure 8-1: The Learning Cycle

```
function: understand_why_operation_used
  if operation simplifies state (reduces number of terms)
    then rule for operation gets score of 0 and return true
  orif effect of operation enables a rule in rulebase
    then rule for operation gets score of 1 plus score of
    rule enabled and return true
  else return false
```

Figure 8-2: Can operation be learned?

```
Subroutine: Learn_operation
  perturb instance, removing nonessential conditions of operation,
    forming candidate rule
  if LHS of rule is not predictive of operation then
    augment instance to find generalization and re-perturb
  call: integrate_operation_into_current_rule_base
```

Figure 8-3: Learn operation

```
Subroutine: integrate_operation_into_current_rule_base
  if a member of rule base can be generalized to cover current
    candidate then supplant member by generalization
  else
    add candidate rule to rule base.
```

Figure 8-4: Integration subroutine

```
Subroutine: apply_episode
  Apply the rule with minimum score S
  loop       (apply rules in remainder of episode)
  while S>0
     S<—S-1
     apply rule with score S
  repeat
  loop       (apply score-0 rules)
     select score-0 rule from those enabled
     and apply it
  while a score-0 rule is enabled
```

Figure 8-5: Apply Episode

9. Examples of System Performance

This section discusses highlights from PET's episodic learning for problem solving in the domain of linear equations.

9.1. Example 1--Learning Combine

The rulebase is initially empty and the teacher presents a training instance:

```
a: 2x+3y=5
b: 2x+4y=6
```

with the advice sub(a,b). PET applies the operator which yields:

```
a: 2x+3y=5
b: 2x-2x+4y-3y=6-5
```

PET must understand why an operator is useful before a rule is formed. The operator failed to simplify the equations (in fact the number of terms in the equations went from six to nine) and did not enable any other rules (since the rulebase is empty). Unable to understand why sub(a,b) was suggested by the teacher, PET cannot form a rule for the operator and waits for something understandable to happen.

The teacher now suggests that combinex(b) be applied, yielding:

```
a: 2x+3y=5
b: 0x+4y-3y=6-5
```

Since the number of terms is reduced from nine to eight, PET understands the

purpose of combinex(b) (to simplify the state) and proceeds to form a rule. This involves stage 1 and stage 2 learning.

Stage 1 learning involves forming a rule for when the operator is applied. Perturbation tests each term in the equations to determine which are essential and which can be generalized. PET forms the rule:

{term(b,pos(N)*x), term(b,neg(M)*x)} => combinex(b)

which means:

> given a problem state, whenever equation b contains an x-term with a positive coefficient and an x-term with a negative coefficient, then combine the two terms.

The new set of equations is:

a: 2x+3y=5
b: 0x+4y-3y=6-5

PET is unable to apply current knowledge (i.e. the rule for combinex(b)) so the teacher suggests combiney(b)[5] which yields:

a: 2x+3y=5
b: 0x+1y=6-5

Stage 1 learning produces the rule:

{term(b,pos(N)*y), term(b,neg(M)*y) => combiney(b)

This rule cannot be generalized with the current rulelist and is simply added.

Learning rules for the operators combinec(b) and deletezero(b) are similar and will be assumed to be completed.

Stage 2 learning of the combine operators involves relating them to episodes, or sequences of operators. Since combine simplifies a problem state immediately, the operators are given a score of zero. The current rulelist

---

[5] deletezero(b) could also be suggested, but we continue with a combine operator for continuity.

(with scores) is:

```
0 -- {term(b,pos(N)*x), term(b,neg(M)*x)} => combinex(b)
0 -- {term(b,pos(N)*y), term(b,neg(M)*y)} => combiney(b)
0 -- {term(b,pos(N)), term(b,neg(M)} => combinec(b)
0 -- {term(b,0*x)} => deletezero(b)
0 -- {term(b,0)} => deletezero(b)
```

With further training instances for the combine operators, PET forms the rules:

```
0--{term(eqn(L),int(N)*x), term(eqn(L),int(M)*x)} => combinex(eqn(L))
0--{term(eqn(L),int(N)*y), term(eqn(L),int(M)*y)} => combiney(eqn(L))
0--{term(eqn(L),int(N)), term(eqn(L),int(M)} => combinec(eqn(L))
0--{term(eqn(L),0*var(X))} => deletezero(eqn(L))
0--{term(eqn(L),0)} => deletezero(eqn(L))
```

9.2. Example 2--Learning Subtract

The teacher now gives another training instance for subtract:

```
a: 3x+4y=10
b: 3x+5y=11
```

Since the combine operators in the rulebase do not apply, the teacher suggests that sub(a,b) be applied, yielding:

```
a: 3x+4y=10
b: 3x-3x+5y-4y=11-10
```

Now combinex(b) applies, so sub(a,b) can be learned (since it enables a rule with score 0). PET learns the rule (after perturbation):

```
{term(a,3*x), term(b,3*x), term(b,pos(N)*y)} => sub(a,b)
```

With further training instances the generalized rule for sub is:

```
{term(eqn(L1),nonzero(N)*var(X)), term(eqn(L2),nonzero(M)*var(X)),
 term(eqn(L2),nonzero(0)*var(Y))} => sub(eqn(L1),eqn(L2))
```

This rule is given a score of one (one plus score of rule enabled=1+0).

PET has learned its first episode with the scored rules:

```
0--{term(eqn(L),int(N)*x), term(eqn(L),int(M)*x)} => combinex(eqn(L))
0--{term(eqn(L),int(N)*y), term(eqn(L),int(M)*y)} => combiney(eqn(L))
0--{term(eqn(L),int(N)), term(eqn(L),int(M)} => combinec(eqn(L))
0--{term(eqn(L),0*var(X))} => deletezero(eqn(L))
1--{term(eqn(L1),nonzero(N)*var(X)), term(eqn(L2),nonzero(M)*var(X)),
    term(eqn(L2),nonzero(0)*var(Y))} => sub(eqn(L1),eqn(L2))
```

It is important to note that the episode learned is "loosely packaged." That is, rules from the rulebase can be applied in any order so long as the scores of the rules in the sequence are non-increasing. This enables PET to proceed with learning subtract while the rules for the combine operators are being refined. It is important that PET be able to use partial-knowledge with assurance that:

- rules formed using the partial-knowledge as a basis will not be incorrect (refer to our definition of correct rules in section 1.)
- the partial-knowledge can be refined independent of other rules in the rulebase.

9.3. Example 3--Learning Multiply

The teacher presents PET with an example of multiply with the training instance:

3x+4y=7
6x+2y=8

Current knowledge does not apply, so PET requests advice. mult(a,2) is suggested which yields:

6x+8y=14
6x+2y=8

Now sub(a,b) applies so the rule for mult(a,2) can be learned. Mult(a,2) is given a score of 2 (one more than the score of the rule enabled). Stage 1 learning requires perturbing each term of the pair of equations and testing for essential features. Example perturbations and resulting conclusions are:

```
Perturbation 1: Delete term(a,3*x)
      4y=7  mult(a,2)     8y=14
   6x+2y=8    ==>      6x+2y=8
 No score<2 rule is enabled so presence of
 non-zero term is essential.

Perturbation 2: Increment term(a,3*x)
   4x+4y=7  mult(a,2)  8x+8y=14
   6x+2y=8    ==>      6x+2y=8
 No score<2 rule is enabled so term(a,3*x) is
 essential.

Perturbation 3: Delete term(a,4*y)
   3x   =7  mult(a,2)  6x   =14  sub(a,b)  6x   =14
   6x+2y=8    ==>      6x+2y=8      ==>    6x-6x+2y=8-14

 Mult(a,2) enabled the sub(a,b) rule.
 Therefore, term(a,4*y) is non-essential
 and is deleted from the generalization of the rule.

Perturbations 4-12 are similar.
```

From this stage 1 analysis, PET forms the rule:

```
{term(a,3*x),term(b,6*x),term(b,pos(N)*y)} => mult(a,2)
```

At this point PET realizes that it has over-generalized since the rule is non-predictive (the 2 on the RHS does not occur on the LHS). PET augments the instance description and forms the candidate rule:

```
{term(a,3*x),term(b,6*x),term(b,pos(N)*y),productof(2,3,6)} => mult(a,2)
```

After additional examples, PET forms the correct rule:

```
2 -- {term(a,pos(K)*x),term(b,pos(L)*x),term(b,pos(M)*y),
      productof(pos(N),pos(K),pos(L)}          => mult(a,pos(N))
```

which supplants the more specific rule in the rulebase.

A troublesome (first) training instance for multiply is:

```
a: 3x+4y=9
b: 9x+3y=21
```

Assuming PET does not have a rule for multiply, the teacher suggests mult(a,3). After perturbation, PET forms the rule:

```
{term(a,3*x),term(b,9*x),term(b,pos(N)*y)} => mult(a,3)
```

The problem is that the rule is "falsely" predictive. Although the rule

passes the test for predictiveness, the 3 on the LHS does not explain (or account for) the 3 on the RHS. Therefore, the rule will not correctly generalize.

Removing these spurious concepts is the role of perturbation theory. However, our perturbation operators are too weak to generate a positive instance of the operator mult(a,3) from the example given. This would require perturbing two terms (or features) of the example simultaneously (e.g. the x-terms of each equation). Unable to find a positive instance through perturbation, PET cannot discover the spurious association which resulted in the error.

The difficulty arises due to the simplicity of the test for predictiveness. Since the test is syntax-based it lacks an understanding of how terms are related and why they are important to a rule. We beleive that a deeper representation of knowledge in rules is essential and are currently addressing this issue.

9.4. Example 4--Learning "Cross Multiply"

The teacher presents the training instance:

2x+6y=8
3x+4y=7

Since no rule is enabled, the teacher advice to apply mult(a,3) yields:

6x+18y=24
3x+ 4y=7

Since mult(b,2) is enabled, mult(a,3) can be learned. After perturbation, PET acquires the rule:

{term(a,2*x),term(b,3*x),term(b,pos(N)*y)} => mult(a,3)

This rule is given a score of 3 since it enables a rule with score 2. The rule will be generalized (after perturbation and subsequent training

instances) to:
3 -- {term(a,pos(N)*x),term(b,pos(M)*x),term(b,pos(L)*y)} => mult(a,pos(M))

10. Limitations and Extensions

As with most learning programs we require that the concept to be learned be representable in our generalization language. In addition PET has to be supplied with some coarse notion of when an operator has been effective in simplifying the current state. Furthermore we assume that the teacher gives only appropriate advice and there is no "noise."

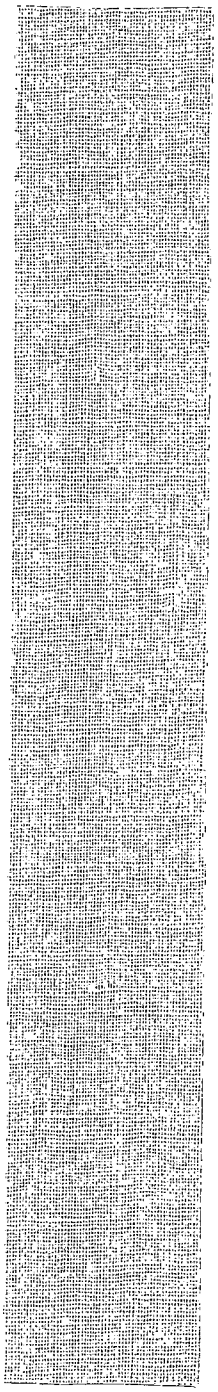
Extensions that we are considering are:

- Learning from negative instances as well as positive ones.
- Improving the use of augmentation by introducing structured concepts. These would permit climbing tree generalizations for this second-order knowledge. Another improvement would be allowing multiple concepts to be pulled into a rule from the augmentation search space. This requires a requisite change in the test for predictiveness.
- Applying the theory to learning operators in other domains. Integration problems have been attempted [10]. We would like to try our approach in the calculus problem domain.

11. Conclusions

A system has been described which learns sequences of rules, or episodes, for problem solving. The learning is incremental and thorough. The system learns when and why operators are applied. Although the system starts with an extremely general and course notion of why an operator should be applied, it's representation becomes increasingly fine and complete as it forms rules from examples. The idea of augmenting the generalization language with higher-level concepts was introduced. Moreover, we described why the augmentation is natural and when it must be used. Due to the power of perturbation, our system can learn episodes with minimal teacher interaction. The episodes are segmented into discrete, re-usable segments, each

accomplishing a recognizable simplification of the problem state. The approach is shown effective in the domain of solving simultaneous linear equations.



REFERENCES

1. Anzai,Y. Learning strategies by computer. CSCSI II (1978), 181-190.

2. Brazdil, P. Experimental learning model. AISB Conference Proceedings (1978), 46-50.

3. Carbonell, J.G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

4. Fikes, R.E. and Nilsson, N.J. STRIPS: A new approach to the application of theorem proving to problem solving. AI 2 (1971), 189-208.

5. Kibler, D.F, and Morris, P.H. Dont be Stupid. IJCAI (1981), 345-347.

6. Michalski, R.S., Dietterich, T.G. Learning and Generalization of Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods. IJCAI 6 (1979), 223-231.

7. Michalski,R.S., Carbonell,J.G., Mitchell,T.M. Machine Learning. Tiogo Publishing, 1983.

8. Michalski, R.S. A Theory and Methodology of Inductive Learning. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

9. Mitchell, T.M. Version spaces: a candidate elimination approach to rule learning. IJCAI 5 (1977), 305-310.

10. Mitchell, T.M., Utgoff, P.E., Nudel, B, and Banerji, R. Learning Problem-Solving Heuristics Through Practice. IJCAI 7 (1981), 127-134.

11. Mitchell, T.M. Generalization as Search. Artificial Intelligence 18 (1982), 203-226.

12. Mitchell, T.M., Utgoff, P.E., Nudel, B, and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem- Solving Heuistics. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

13. Neves, D.M. A computer program that learns algebraic procedures by examining examples and working problems in a textbook. CSCSI II (1978), 191-195.

14. Vere, S.A. Induction of concepts in the predicate calculus. IJCAI 4 (1975), 281-287.

15. Vere, S.A. Induction of Relational Productions in the Presence of Background Information. IJCAI 5 (1977), 349-355.

16. Vere, S.A. Inductive learning of relational productions. In Waterman, D.A. and Hayes-Roth, F., Ed., Pattern-Directed Inference Systems, Academic Press, 1978.

17. Waterman, D.A. Generalization learning techniques for automating the learning of heuristics. AI 1 (1970), 121-170.

18. Winston, P.H. Learning structural description from examples. In Winston, P.H., Ed., The Psychology of Computer Vision, McGraw-Hill, 1975.