PERTURBATION: A MEANS FOR GUIDING GENERALIZATION

Dennis Kibler
Bruce Porter

Information and Computer Science Department
University of California at Irvine
Irvine, California[1]

Abstract

Learning problem solving from examples suffers from three problems. First, there is a strong dependency on the order of the presented examples. Second, each example has its own peculiarities which must be overcome. Third, the size of the generalization space can be huge, even if the instance language is small. By adding perturbation operators to the concept tree each of these problems can be alleviated. This is demonstrated in a system which learns, through interaction with a teacher, to solve simultaneous linear equations.

1. Introduction

With the aid of a teacher, junior high school students can learn to solve simultaneous linear equations. A program, given the same information, has numerous problems to overcome. One problem is focussing attention: the solutions presented may contain spurious associations, hiding the essential characteristics or features. Another problem is the extremely large space of possible rules (candidate generalizations) that match a given instance of a set of linear equations and the given appropriate operation. For a natural representation of equations there are more than one million rules one might infer. We avoid these problems by examining the effect of the same operator on a "near" example created by perturbing the given example. In this manner we can focus attention on the essential features and reduce the size of the search space to several thousand possibilities. Once this is done we can apply standard generalization techniques, such as described by Vere [10, 12, 13], Michalski [5], or Mitchell [6,8]. As a side benefit this technique also mitigates the effect of the particular sequence of examples that the teacher presented.

[1]This research was supported by the Naval Ocean System Center under contract N00123-81-C-1165.

2. Related Work

Winston [15] showed the importance of "near" misses in learning concepts about the blocks world. By using perturbations "near" examples are generated automatically. We use a relational production system, somewhat like Vere's [12] except that we use a bag of conditions rather than a set, to represent the program's knowledge of when to apply operators. Production systems have been successfully used to model the acquisition of skill for poker playing [14], puzzle solving [1], algebra problems [9], arithmetic problems [2], and symbolic integration [7]. Of these, Neves's [9] 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 [7] 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 [10] of the examples.

3. Perturbation Method

Before discussing the general technique of perturbations, we will illustrate its use in learning to solve simultaneous linear equations. We adopt a relational description of each equation, so equation(a) 2x-3y=-7 is stored as: {tenn(a,2*x) ,term(a,-3*y),term(a,7)}.

Following Mitchell [7] and Michalski [5] we have concept trees for integers, variables, and

equation labels. Our concept tree for integers is shown in figure 3-1.

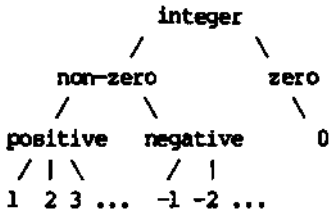

Figure 3-1: Concept tree for integers

For variables the concept tree is simpler. An algebraic variable stands for itself or can be generalized to var(X), where X is a variable in the pattern language. Similarly labels are either the particular label or a label variable. Basically we are using the typed variables of Michalski [5].

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 [5] 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 label(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 about by deleting terms.

The program is set the task of learning when to apply opaque operators, i.e. operators that are hard-coded and unanalyzable by the program. The operators are: add(a,b), sub(a,b), and cross(a,b,c1,c2). where a and b are equation labels and c1, and c2 are integers. The add(a,b) operator replaces equation (b) by the sun of equation(a) and equation(b). Simplification takes place as part of the application of an operator. The operator sub(a,b) is defined similarly. The operator cross(a,b,c1,c2) replaces equation(b) by c1*equation(a) - c2*equation(b)• We show how these powerful operators can be learned from simpler operators using episodic segmentation [4]. In this paper we concentrate on reducing the size of the search space.

3.1. Learning Cycle

Initially the program has no rules for applying its operators. A high-level description of the program[2] is given in figures 3-2 and 3-3.

```
repeat
  get problem from teacher
  repeat
    if some rule matches problem then
      apply it (no learning)
      else get operation from teacher and
        call: integrate operation into rule base
  until problem solved
  display current set of rules
until teacher satisfied
```

Figure 3-2: Main driver

```
Subroutine: integrate operation into rule base
  set candidate rule to instance
  repeat
     perturb problem
     if operator still effective, minimally
        generalize current (perturbed) instance
        with candidate rule
  until no more perturbations
  if a member of rule base can be generalized to
    cover current candidate rule, then replace
    member by generalization
   else add candidate rule to rule base.
```

Figure 3-3: Perturbation subroutine

For example, given the problem:

a: 2x+3y=7
b: 2x-5y=3

the teacher advice to sub(a,b), and an empty rule base, the system first describes the rule as:

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

Now the program "perturbs" the instance by modifying each of the coefficients individually. This is done by zeroing, incrementing and decrementing each coefficient. Some of the equations generated by perturbation are:

(i)	(ii)	(iii)	(iv)	(v)
3y=7	2x =7	2x+3y=7	2x+3y=0	2x+3y=7
2x-5y=3	2x-5y=3	2x =3	2x-5y=3	2x-6y=3

Notice that sub(a,b) is still effective in examples ii, iv and v but is not effective in examples i and iii. By effective we mean that not only is the operator applicable, but also that it simplifies the problem state. We were surprised to discover that example iii is not a positive instance for sub(a,b). Note that by applying

---

[2]Implemented in Prolog on DEC-2020. Available upon request.

sub(a,b) the resulting equations are:
a: 2x+3y7
b: -3y-4
which is not simplier (sub(b,a) would be effective however).

Since the operator is effective in example ii, the system generalizes (minimally) its current rule conditions with this example yielding the new rule:

{term(a,2*x),term(a,-7),
term(b,2*x),tenn(b,-5*y),term(b,-3)} => sub(a,b)

The major effect is to delete the condition on the y-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. After generalizing with example iv_f the rule becomes:

{term(a,2*x) ,term(b,2*x) ,term(b,-5*y),
term(b,-3)} «> sub(a,b).

The effect of generalizing with example v is to allow any negative coefficient for the y-term of equation(b):

{tem(a,2*x),term(b,2*x),term(b,neg(N)*y),
$term(b_f$-3)} => sub(a,b).

Futher perturbations yield the candidate rule:
{term(a,2*x) ,term(b,2*x) ,term(b,neg(N)*y) }=>sub(a,b).
Since each term in this rule has about 16 possible generalizations, this rule has more than 4000 possible generalizations.

Sometime later, probably in the context of another example, the program may have to incorporate a rule of the form:
{term(a,3*y) ,term(b,3*y) ,term(b,4*x)}«>sub(a,b).
A minimal generalization of this candidate with the previous rule yields the rule:
{tenn(a,pos(N) *var (X)) ,term(b,pos(N) *var ((X)),
term(b,nonzero(M))} «> sub(a,b).
Eventually, depending on the examples chosen by the teacher, an effective set of rules will be generated. Each rule can be fully learned with only two instances, if the instances are choosen to be maximally dissimilar but still requiring the same operator. A more casual and usual set of examples, such as those found in [3] require 4-6 examples to generate the rule. At most five instances or interactions with a teacher are required to learn the rule:

{term(label(A) ,nonzero(N)*var(X),
term(label(B) ,nanzero(N)*var(X),
term(label (B) ,nonzero(M)*var (Y)}
-> sub (label (A), label (B))

A rough english translation of this rule is: If two equations have equal terns, then subtract those equations, replacing equation(label(B)) with the result (provided equation(label(B)) has two terms) •

The rule for the add operator is similar to the rule for subtract:

(tenn(labeKA) ,nonzero(N)*var(X),
term(label (B) ,nonzero(-N)*var(X),
term(label(B) ,nonzero(M)*var(Y)}
=> add (label (A), label (B))

which can be paraphrased as: If two equations have like terms such that the coefficient of one is the negation of the coefficient of the other, then add those equations, replacing equation(label(B)) with the result (provided equation (label (B)) has two terms) •

A rule for the cross-multiply operator that the system forms given typical examples is:

{term(label(A) ,nonzero(M)*var(X)),
term(label (B),nonzero (N) *var (X)),
term(labeKB) ,nonzero(0)*var(Y))}
=> cross (label (A), label (B), nonzero (N) ,nonzero(M))

which roughly means: If two equations have like terms with non-zero coefficients then replace the equation (label (A)) with the result of cross multiplying and substracting the equations (provided equation(label(B)) has two terms).

3.2. Perturbation Operators

The perturbation operators map instances into new sibling instances. In figure 3-4 we illustrate the concept tree for integers augmented with a possible set of perturbation operators, indicated in braces.

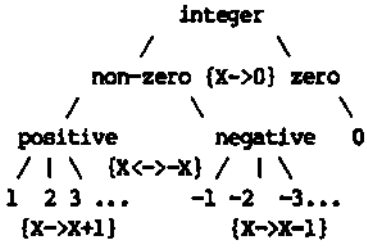

Figure 3-4: Concept tree with perturbation operators

Perturbation breaks up the generalization process into two steps. Each example is perturbed multiple times to create near examples and near misses. Minimal generalizations are formed from this set. Primarily this has the effect of sifting out the essential conditions from the non-essential ones. No further teacher assistance is required for this sifting. Additional teacher assistance is required to refine the

generalization formed. Because of the active nature of problem solving, as opposed to standard concept learning, the system is capable of relying less on the teacher for appropriate examples.

Perturbations classified as near-misses are distinguished from near examples by applying a test of whether the operator which simplified the example also simplifies the perturbation. Since the cost of trying the operator is small (as is the cost of generating the perturbation) there is no need to guide the selection of which perturbations to try. If there is a high cost associated with testing an example, as in re-applying the problem solver in LEX [7], heuristics for guiding the generation of perturbations are needed.

4. Limitations and Extensions

As with most learning programs we require that the concept to be learned be representable in our generalization language. In addition the system 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 is not malicious and gives only appropriate advice.

Instead of generating all perturbations of an instance, though it is practical in this domain, we plan to allow the current generalization of the rule determine the appropriate perturbations. We also plan to allow sequences of non-improving operations. Currently no perturbation can modify more than one coefficient. By adding the capability to modify several coefficients simultaneously, perturbations could be used to direct the tree generalization. Reconsider the example:

a: 2x+3y=7
b: 2x-5y=3

We showed that by perturbing one coefficient at a time, the rule

{term(a,2*x) ,term(b,2*x) ,term(b,neg(M)*y)«>sub(a,b)}

could be derived. If we could simultaneously modify the 2's in the x-terms to 3's, we could generate the new example:

a: 3x+3y*7
b: 3x-5y*3

Since sub(a,b) is effective here, the new rule

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

could be formed. Lastly we intend to apply the technique to other problem solving domains.

5. Conclusions

For learning operators, we have shown that perturbations form an effective means for eliminating nonessential features from the search space. For learning how to solve simultaneous linear equations, the search space was reduced from approximately a million candidates to several thousand. To learn the correct conditions for applying an operator, the system requires at most five interactions with the teacher. Essentially, perturbation is a technique for creating near examples and near misses upon which standard generalization techniques can be applied.

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. Keedy and Bittenger. Introductory Algebra. Addison-Wesley, 1979.
4. Kibler, D.F., and Porter, B.W. Episodic Learning. 194, University of California, Irvine, 1983.
5. 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.
6. Mitchell, T.M. Version spaces: a candidate elimination approach to rule learning. IJCAI 5 (1977), 305-310.
7. Mitchell, T.M., Utgoff, P.E., Nudel, B, and Banerji, R. Learning Problem-Solving Heuristics Through Practice. IJCAI 7 (1981), 127-134.
8. Mitchell, T.M. Generalization as Search. Artificial intelligence 18 (1982), 203-226.
9. Neves, D.M. A computer program that learns algebraic procedures by examining examples and working problems in a textbook, CSCSI II (1978), 191-195.
10. Vere, S.A. Induction of concepts in the predicate calculus. IJCAI 4 (1975), 281-287.
11. Vere, S.A. induction of Relational Productions in the Presence of Background Information. IJCAI 5 (1977), 349-355.
12. Vere, S.A. Inductive learning of relational productions. In Waterman, D.A. and Hayes-Roth, F., Ed., Pattern-Directed inference System, Academic Press, 1978.
13. Vere, S.A. Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions. Artificial intelligence 11 (1980), 138-164.
14. Waterman, D.A. Generalization learning techniques for automating the learning of heuristics. AI 1 (1970), 121-170.
15. Winston, P.H. Learning structural description from examples. In Winston, P.H., Bd., The Psychology of Computer vision, McGraw-Hill, 1975.