Appears in *Proceedings of the Second International Workshop on Inductive Logic Programming,* Tokyo, Japan, June, 1992

Automated Debugging of Logic Programs via Theory Revision *

Raymond J. Mooney and Bradley L. Richards
Department of Computer Sciences
University of Texas
Austin, TX 78712
mooney@cs.utexas.edu, bradley@cs.utexas.edu

February 14, 1992

Abstract

This paper presents results on using a theory revision system to automatically debug logic programs. FORTE is a recently developed system for revising function-free Horn-clause theories. Given a theory and a set of training examples, it performs a hill-climbing search in an attempt to minimally modify the theory to correctly classify all of the examples. FORTE makes use of methods from propositional theory revision, Horn-clause induction (FOIL), and inverse resolution. The system has has been successfully used to debug logic programs written by undergraduate students for a programming languages course.

1 Introduction

Most of the recent work on inductive logic programming has focused on the synthesis of logic programs from examples, e.g. FOIL [Quinlan, 1990] and GOLEM [Muggleton and Feng, 1990]. There has been very little work on automated debugging of logic programs since Shapiro's original work on the subject [Shapiro, 1983]. Although systems like FOIL and GOLEM generally make use of existing subroutines, e.g. learning `reverse` given `append`, the supplied definitions are extensional rather than intensional and these systems can only

*This research was supported by the National Science Foundation under grant IRI-9102926 and by the NASA Ames Research Center under grant NCC 2-629. The second author was supported by the Air Force Institute of Technology faculty preparation program.

learn new theories and cannot revise existing ones. However, unlike Shapiro's system, recent systems do not rely on an omniscient oracle that is capable of determining the truth of arbitrary ground atomic formulae. Consequently, current systems require much less user interaction.

This paper presents results on using methods for first-order theory revision to automatically debug logic programs without the use of an oracle. We are developing a theory revision system, FORTE [Richards and Mooney, 1991], that modifies an incomplete and/or incorrect Horn-clause domain theory to fit a set of training examples. When the domain theory is viewed as a logic program, theory revision corresponds to automated debugging from I/O pairs. The design of FORTE has been influenced by a number of previous developments. Many of its basic revision operators (delete antecedent, add antecedent, delete rule, and add rule) are based on similar operators in EITHER, a predecessor of FORTE for propositional Horn-clause theories [Ourston and Mooney, 1990; Mooney and Ourston, 1991]. FORTE also makes use of a FOIL-like algorithm to add new rules and antecedents, and several inverse-resolution operators [Muggleton and Buntine, 1988] to generalize and compress the theory. The system has recently been augmented with a path-finding method for overcoming local-maxima [Richards and Mooney, 1992] and modified to handle recursion.

To test the resulting system, we collected buggy Prolog programs from students in an undergraduate course on programming languages. Students were asked to hand in their initial versions of programs for finding a path in a graph and inserting a new element after a specified element in a list. The collected programs included a total of 12 distinctly different buggy programs. FORTE was able to correctly debug all but one of these programs based on a relatively small set of training examples.

Since induction of complete programs is a very difficult problem, we believe automated debugging is a more realistic goal with greater potential for practical utility. Induction of programs in other languages has not been particularly successful. Even the most sophisticated systems have been able to produce only relatively toy programs [Barstow, 1988]. Using inductive techniques to revise buggy programs has not been explored nearly as well, and we believe our initial results in oracle-free inductive logic-program debugging are quite promising.

The remainder of the paper is organized as follows. Section 2 presents an overview of FORTE's basic revision algorithm. Additional details on the revision algorithm are presented in [Richards and Mooney, 1991; Richards and Mooney, 1992]. Section 3 presents results on automatically debugging actual student programs. Section 4 presents results on debugging subroutines using only examples for the main program. Section 5 discusses related work and Section 6 presents directions for future research. Section 7 presents our conclusions.

2 FORTE's Theory Revision Algorithm

In order to revise a logic program, one must first detect an error. FORTE attempts to prove all positive and negative instances in the training set using the current program. As in most ILP systems, positive (negative) instances are tuples of constants that should (should not) satisfy the goal predicate. When a positive instance is unprovable, some program clause needs to be generalized. All clauses that failed during the attempted proof are candidates for generalization. When a negative instance is provable, some program clause needs to be specialized. All clauses that participated in the successful proof are candidates for specialization.

When an error is detected, FORTE identifies all clauses that are candidates for revision. The core of the system consists of a set of operators that generalize or specialize a clause to correctly classify a set of examples. Based on the error, all relevant operators are applied to each candidate clause. The best revision, as determined by classification accuracy on the complete training set, is implemented. This process iterates until the theory is consistent with the training set or until FORTE is caught in a local maximum, i.e. none of the proposed revisions improve overall accuracy.

FORTE maintains a list, for each program clause, of all errors for which the clause may be responsible. After trying to prove all instances, the clauses are ordered by the number of errors for which they may be responsible. Note that we treat specialization and generalization separately, so each clause may appear twice in this ordered list. Revision begins with the clause that provides the greatest potential benefit.

2.1 Specializing the theory

When one or more negative instances are provable, the theory needs to be specialized. There are two basic ways to do this: delete a clause or add new antecedents to an existing clause. Deleting a clause is a simple operation. The clause identified in the revision point is deleted from the theory, and the remaining theory is tested on the training set.

Adding antecedents to a clause is more complex. Our goal is to eliminate incorrect proofs of negative instances without eliminating many (or any) of the correct proofs of positive instances. It may be necessary to create several specializations of an existing clause. For example, we might specialize the propositional rule

```
a :- b, c
```

in two different ways

```
a :- b,c,d,e
a :- b,c,f
```

FORTE employs two competing methods for adding antecedents. The first is similar to FOIL in that it adds antecedents one at a time, choosing at any point the antecedent that provides the best information gain. Our information metric is different from FOIL's in that it considers only the number of positive and negative instances – not the number of proofs of positives and negatives.

The second algorithm is called relational pathfinding [Richards and Mooney, 1992]. We designed this method based on the assumption that relational concepts are usually represented by a small number of fixed relational paths connecting the arguments of a predicate. Relational pathfinding is reminiscent of Quillian's spreading activation [Quillian, 1968]. In essence, the variables already present in the clause are treated as nodes in a graph. Relations in the predicate are treated as edges (predicates with arity greater than two are simply edges with more than two ends). THe graph associated with the overly-general clause is frequently disconnected into separate subgraphs. Relational pathfinding seeks to find a path of additional relations that can be used to join the distinct subgraphs into a coherent whole.

2.2 Generalizing the theory

When one or more positive instances cannot be proven, the theory needs to be generalized. There are several ways to do this in FORTE: deleting antecedents from an existing clause, adding a new clause, or using the inverse resolution operators identification or absorption.

As with clause specialization, several generalizations of a clause may need to be generated in order to provide proofs for all of the positives without letting negatives become provable. There are two approaches to deleting antecedents from a rule. The first approach, which works well in simple cases, is to delete antecedents singly, based on a simple information metric. However, in some cases, we may need to delete several antecedents simultaneously in order to gain anything. A simple example of this is an "m of n" type problem. Given the clause

```
a :- b, c, d, e, f
```

it may be that all of the antecedents are important, but positive instances are distinguished by satisfying any three of them. In this case, we would need to create the rules

```
a :- b, c, d
a :- b, c, e
a :- b, c, f
a :- b, d, e
```

and so forth. However, deleting a single antecedent may not help, since none of the positives may satisfy a four antecedent rule like

```
a :- b, c, d, e
```

In order to add a new clause, FORTE copies the clause identified in the revision point. It then deliberately over-generalizes this clause by deleting all antecedents whose deletion allows the proof of one or more previously unprovable positives – even though their deletion may also allow proofs of some negative instances. This overly general rule is then given to the antecedent addition algorithms described in the section on specialization.

Lastly, FORTE can use the inverse resolution operators identification and absorption to generalize a program. Identification allows alternate definitions of a particular literal to come into play. For example, suppose we have the following two clauses in our program.

```
   a :- b, c, d
   a :- b, x
  --------------
   x :- c, d
```

Assume `c` in the first clause was a failure point in an attempted proof of a positive instance. Identification replaces the first clause with the third clause shown. This does not affect current proofs, but it allows other definitions of `x` to generalize proofs that use the first clause.

Absorption is, in effect, the complement of identification. Suppose we have, in our program, the first two clauses shown below.

```
   a :- b, c, d
   x :- c, d
  --------------
   a :- b, x
```

Absorption notes the common antecedents and replaces the first clause with the third one. Again, this does not endanger any existing proofs, and has the effect of allowing alternate definitions of `x` to come into play. Absorption is particularly useful for developing recursive clauses, since `x` and `a` may be the same predicate.

3 Debugging Student Programs

In order to test FORTE's debugging ability, sample logic programs were collected from students taking an undergraduate course on programming languages. Students were given an assignment to write several short logic programs. The first problem involved finding a path in a directed graph, a standard test problem for FOIL [Quinlan, 1990]. Below is a correct program for this problem.

```
path(A,B):-edge(A,B).
path(A,B):-edge(A,C),path(C,B).
```

The second involved inserting an element in a list immediately following a specified item. Below is a correct program for this problem: [1]

```
insert_after([A|B],A,C,[A,C|B]).
insert_after([A|B],C,D,[A|E]) :- C\=A, insert_after(B,C,D,E).
```

where `insert_after(X,A,B,Y)` inserts `B` after the first occurrence of `A` in the list `X`. Students were asked to hand in the initial versions of their programs before running and debugging them. We received 3 distinctly different buggy programs for `path`, and 9 for `insert_after`.

FORTE was given each of these 12 programs to debug together with a training set of correct input-output pairs. The training set for `path` was a complete set of positive and negative examples for an 11-node graph (15 positive, 106 negative). The training set for `insert_after` contained 10 positive examples for lists up to size 3 and 23 selected negative examples. For this problem, FORTE is also given a definition for `components`.

The system was able to correctly debug all but one of the 13 programs. Debugging time took an average of 68 seconds for the `path` programs and 350 seconds for the `insert_after` examples running in Quintus Prolog on a Sun Sparcstation. One program was not properly debugged due to a local maximum. The revisions included deleting incorrect clauses, adding additional literals to clauses, and adding totally new clauses. Below is one of the buggy student programs for `path`:

```
path(A,B):-edge(B,A).
path(A,B):-edge(A,B).
path(A,B):-edge(A,C),edge(D,B),path(A,C).
```

The first clause is incorrect since a directed path is desired. Since it covers a number of negatives and no positives, FORTE retracts this clause as its first revision. Since the student wrote the recursive clause with two edges and a path, the case of a path of length two is not handled. As a result, FORTE adds the rule:

```
path(A,B) :- edge(A,C), path(C,B).
```

Finally, FORTE decides to delete the student's original recursive clause since the new clause covers all of the examples it covers. The result is the simple correct program presented earlier.

One of the buggy student programs for `insert_after` is shown below:

```
insert_after([A|B],C,D,[A|E]):-insert_after(B,C,D,E).
insert_after([A|B],A,C,[A,C|D]):-insert_after(B,A,C,D).
```

[1]It should be noted that FORTE uses a function-free representation like FOIL. A term shown as [A|B] must actually represented by an additional literal, `components(X,A,B)`, in the body of the clause. FORTE translates its results into normal Prolog notation for readability.

This program is missing a base case, so the first revision FORTE makes is to add the clause:

```
insert_after([A|B],A,C,[A,C|B]).
```

Next, the system adds the antecedent $C \neq A$ to the student's first clause to prevent answers that never insert the desired item. Finally, the student's second clause is deleted because the instructions were to insert `C` only after the first occurrence of `A`. The result is the correct version of `insert_after` previously presented.

4 Additional Debugging Examples

Since the students in the previous experiment were writing their first Prolog programs, the examples could not be too difficult. Like many ILP systems, FORTE can actually induce complete programs for such simple problems. Given an empty initial theory and the same data used to refine the buggy programs, FORTE can construct complete and correct definitions for `path` and `insert_after`. Consequently, it is not particularly surprising that it can also debug programs for these problems.

However, FORTE can also debug programs which, due to fundamental limitations or resource constraints, existing ILP systems cannot induce without an oracle. Since they cannot create new predicates, systems like FOIL and GOLEM cannot induce programs with recursive subroutines unless they are given extensional definitions of these subroutines. For example, they cannot produce a program for `reverse` given only a background definition for `components` of a list – they need an extensional definition of `append`. Although FORTE is also unable to induce programs with recursive subroutines from scratch, it can debug many incorrect definitions.

As an example of fixing a recursive subroutine, consider the following buggy definition of `subset`.

```
member(A,[A|B]).
member(A,[B|C]).
subset([],A).
subset([A|B],C) :- member(A,C), subset(B,C).
```

The second clause for `member` is missing the recursive literal `member(A,C)` from its body. FORTE successfully added this antecedent given only 64 positive and negative examples of `subset` and a definition of `components` – explicit examples of `member` were not necessary.

5 Related Work

As previously mentioned, most recent work in ILP has concerned the complete induction of logic programs from examples [Quinlan, 1990; Muggleton and Feng, 1990; Kijsirikul *et* *al.*, 1991]. Shapiro's Prolog debugger, PDS6 [Shapiro, 1983], uses many techniques from his learning system, MIS; however, it requires a great deal of user interaction. The user must be available to answer membership queries as well as provide other detailed information. FORTE requires only a sufficient supply of examples; no oracle or additional user interaction is required.

Work in automated debugging for other programming languages has primarily employed *static* methods that compare a program to a formal specification [Katz and Manna, 1976] abstract program plan [Johnson, 1986], or existing correct program [Murray, 1988]. By comparison, PDS6 and FORTE are *dynamic*. They run a program on specific examples, detect errors, and use them to revise the program.[2] Consequently, dynamic methods require only partial, extensional definitions of programs. This is an important advantage since formal specifications are frequently unavailable. Systems that require an existing correct program (e.g. TALUS [Murray, 1988]) are primarily useful in tutoring environments, since a correct program is rarely available in other situations.

Most other work in theory revision is propositional in nature, and therefore inapplicable to logic programming [Ginsberg, 1990; Towell and Shavlik, 1991; Cain, 1991]. FOCL [Pazzani *et al.*, 1991] uses an initial theory to guide a FOIL-based system; however, it produces a flat, operationalized definition instead of a revised theory. A version of FOCL that performs theory revision has been developed [Pazzani and Brunk, 1990]; however, it requires significant user interaction. Finally, FOCL has not been tested on logic programming problems and it is unclear how its operationalization procedure would handle recursion.

6 Future Work

As with all existing ILP systems, the problems currently used to test FORTE are quite simple. Consequently, we plan to test FORTE's ability to debug more difficult programs. An interesting problem we are considering is debugging a Prolog implementation of ID3 [Bratko, 1990]. Since debugging is normally much easier than program synthesis, we believe FORTE should be able to handle larger problems than purely inductive systems. We also plan to test FORTE on other real-world problems such as revising qualitative models of complicated systems [Bratko *et al.*, 1991; Richards and Mooney, 1992] and revising the Chou-Fassman theory for protein folding [Maclin and Shavlik, 1991].

Like many other ILP systems (e.g. GOLEM, FOIL), FORTE is unable to create new predicates. Current predicate-invention methods such as inverse resolution [Muggleton and Buntine, 1988] are computationally very demanding and usually employ an oracle. Efficient oracle-free methods for predicate invention are needed to revise programs that require additional recursive subroutines.

[2]The terms *static* and *dynamic* are borrowed from [Murray, 1988].

We are also developing techniques for learning search heuristics [Mitchell, 1984; Cohen, 1990] to improve the efficiency of logic programs. Meta-rules for when to use a particular clause can be empirically learned using sample calls for which the clause ultimately failed or succeeded in leading to a final solution. Such examples can be extracted from the search conducted during the execution of a logic program [Cohen, 1990]. Existing ILP systems should be useful for learning search heuristics from these examples. As an example, consider the following exponential-time sorting program:

```
naivesort(X,Y) :- permutation(X,Y), ordered(Y).

permutation([],[]).
permutation([X|Xs],Ys) :- permutation(Xs,Ys1), remove(X,Ys,Ys1).

remove(X,[X|Xs],Xs).
remove(X,[Y|Ys],[Y|Ys1]) :- remove(X,Ys,Ys1).

ordered([_X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

The predicate `remove(X,Y,Z)` is true if removing *one* of the occurrences of the item `X` from the list `Y` leaves the list `Z`. The predicate `permutation` actually uses `remove` to insert an element into a list at every possible position. Using examples for when each of the clauses for `remove` leads to a success or failure, it should be possible to learn the following heuristics: the base case ultimately leads to a solution when `Xs` = [] or when `Xs` = [Y|Ys] and $\texttt{X} \leq \texttt{Y}$; the recursive clause leads to a solution when $\texttt{X} > \texttt{Y}$. If these constraints are folded into the existing rules, the resulting definition for `remove` is:

```
remove(X,[X],[]).
remove(X,[X,Y|Ys],[Y|Ys]) :- X =< Y.
remove(X,[Y|Ys],[Y|Ys1]) :- X > Y, remove(X,Ys,Ys1).
```

When this new procedure is used by `permutation`, it always inserts the element `X` before the first element of `Ys1` that is greater than it. Upon inspection, it is clear that the result is an insertion sort, where `permutation` always returns a sorted permutation and the `ordered` check is redundant. Consequently, by learning heuristics for when to use each of the clauses for `remove`, we have turned an $O(n!)$ sorting algorithm into a $O(n^2)$ one! We are currently developing an ILP system that learns such heuristics using a FOIL-based inductive learner.

7 Conclusions

Automated program debugging is an area of ILP that has not been extensively explored. Shapiro's original work in this area has not been followed-up nearly as well as his work on induction of complete programs. We believe that recent developments in first-order induction and theory revision hold great promise in developing dynamic automated debuggers for logic programming. Initial results on using our theory revision system, FORTE, to debug logic programs is quite promising. It is already capable of debugging actual student programs for simple problems without any user interaction. We plan to extend our tests to larger, more realistic problems and to develop effective learning methods for improving the speed as well as the accuracy of logic programs.

Acknowledgements

We would like to thank John Zelle for providing the details of the naive-sort example.

References

[Barstow, 1988] D. Barstow. Artificial intelligence and software engineering. In H.E. Shrobe, editor, *Exploring Artificial Intelligence*. Morgan Kaufman, San Mateo, CA, 1988.

[Bratko *et al.*, 1991] I. Bratko, S. Muggleton, and A. Varsek. Learning qualitative models of dynamic systems. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 385–388, Evanston, IL, June 1991.

[Bratko, 1990] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, Reading:MA, 1990.

[Cain, 1991] T. Cain. The DUCTOR: A theory revision system for propositional domains. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 485–489, Evanston, IL, June 1991.

[Cohen, 1990] W. W. Cohen. Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 268–276, Austin, TX, June 1990.

[Ginsberg, 1990] A. Ginsberg. Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 777–782, Detroit, MI, July 1990.

[Johnson, 1986] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Pitman Publishing, London, 1986.

[Katz and Manna, 1976] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the Association for Computing Machinery*, 19(4):188–206, 1976.

[Kijsirikul *et al.*, 1991] B. Kijsirikul, M. Numao, and M. Shimura. Efficient learning of logic programs with non-determinant, nondiscriminating literals. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 417–421, Evanston, IL, June 1991.

[Maclin and Shavlik, 1991] R. Maclin and J. W. Shavlik. Refining domain theories expressed as finite-state automata. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 524–528, Evanston, IL, June 1991.

[Mitchell, 1984] T. M. Mitchell. Toward combining empirical and analytic methods for learning heuristics. In A. Elithorn and R. Banerji, editors, *Human and Artificial Intelligence*. North-Holland, Amsterdam, 1984.

[Mooney and Ourston, 1991] R. J. Mooney and D. Ourston. A multistrategy approach to theory refinement. In *Proceedings of the International Workshop on Multistrategy Learning*, pages 115–130, Harper's Ferry, W.Va., Nov. 1991.

[Muggleton and Buntine, 1988] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, Ann Arbor, MI, June 1988.

[Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990. Ohmsha.

[Murray, 1988] W. R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing, London, 1988.

[Ourston and Mooney, 1990] D. Ourston and R. Mooney. Changing the rules: a comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 815–820, Detroit, MI, July 1990.

[Pazzani and Brunk, 1990] M. Pazzani and C. Brunk. Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning. In *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, October 1990.

[Pazzani *et al.*, 1991] M. Pazzani, C. Brunk, and G. Silverstein. A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 432–436, Evanston, IL, June 1991.

[Quillian, 1968] M. R. Quillian. Semantic memory. In M. Minksy, editor, *Semantic Information Processing*. MIT Press, Cambridge, MA, 1968.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Richards and Mooney, 1991] B. Richards and R. Mooney. First-order theory revision. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 447–451, Evanston, IL, June 1991.

[Richards and Mooney, 1992] B.L. Richards and R.J. Mooney. Learning relations by path finding. In *submitted paper*, 1992.

[Shapiro, 1983] E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[Towell and Shavlik, 1991] G. Towell and J. Shavlik. Refining symbolic knowledge using neural networks. In *Proceedings of the International Workshop on Multistrategy Learning*, pages 257–272, Harper's Ferry, W.Va., Nov. 1991.