Appears in *Machine Learning,* 19, 2, pp.95-131, 1995

Machine Learning, ??, 1–39 (??)

Automated Refinement of First-Order Horn-Clause Domain Theories

BRADLEY L. RICHARDS BRADLEY@LIA.DI.EPFL.CH
Artificial Intelligence Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland

RAYMOND J. MOONEY MOONEY@CS.UTEXAS.EDU
Department of Computer Sciences, University of Texas, Austin, Texas, 78712

Editor: Steve Minton

Abstract. Knowledge acquisition is a difficult, error-prone, and time-consuming task. The task of automatically improving an existing knowledge base using learning methods is addressed by the class of systems performing *theory refinement*. This paper presents a system, FORTE (First-Order Revision of Theories from Examples), which refines first-order Horn-clause theories by integrating a variety of different revision techniques into a coherent whole. FORTE uses these techniques within a hill-climbing framework, guided by a global heuristic. It identifies possible errors in the theory and calls on a library of operators to develop possible revisions. The best revision is implemented, and the process repeats until no further revisions are possible. Operators are drawn from a variety of sources, including propositional theory refinement, first-order induction, and inverse resolution. FORTE is demonstrated in several domains, including logic programming and qualitative modelling.

Keywords: Theory Revision, Knowledge Refinement, Inductive Logic Programming

1. Introduction

A number of recent machine learning projects have focussed on the task of refining incomplete and/or incorrect rule bases (*domain theories*) (Ginsberg, 1990; Ourston & Mooney, 1990; Towell & Shavlik, 1993; Craw & Sleeman, 1991; Wilkins, 1988). The goal of this work is to automate the laborious process of knowledge-base refinement and thereby speed the development of knowledge-based systems (Ginsberg, Weiss, & Politakis, 1988). Theory refinement normally integrates analytical and empirical machine learning methods in an attempt to leverage two sources of information: approximate rules obtained from an expert or a textbook, and empirical data on actual problems. A theory refinement system is successful to the extent that it can improve the accuracy of its initial domain theory and produce a more accurate and more comprehensible theory than purely inductive methods. Recent experiments have demonstrated such success in a few real-world domains (Ourston & Mooney, in press; Towell & Shavlik, 1993).

However, much existing work in theory refinement has dealt only with propositional rule bases. Such systems are primarily restricted to performing classification tasks for examples described as feature vectors. This paper describes FORTE (First

Order Revision of Theories from Examples), a system for automatically revising function-free first-order Horn-clause knowledge bases (i.e., pure Prolog programs without functions). This more powerful representation language allows FORTE to work in domains involving relations, such as computer programming, qualitative modelling, and natural language processing. Since it uses first-order Horn-clauses as a representation language, FORTE can be viewed as part of the growing body of work in *inductive logic programming* (ILP) (Muggleton, 1992). However, existing ILP research has primarily focussed on generalizing an existing theory by adding clauses, which does not address the issue of modifying incorrect knowledge. Existing ILP systems that modify incorrect knowledge generally require interaction with a user in order to isolate and correct faults (Shapiro, 1983; DeRaedt & Bruynooghe, 1992).

By contrast, FORTE is a fully automated system performing a hill-climbing search through a space of both specializing and generalizing operators in an attempt to find a minimal revision to a theory that makes it consistent with a set of training examples. FORTE's revision operators include methods from propositional theory refinement (Ourston & Mooney, 1990), first order induction (Quinlan, 1990), and inverse resolution (Muggleton & Buntine, 1988). The system has successfully been used to debug Prolog programs collected from students in a course on programming languages, to debug a decision-tree induction program, and to revise a qualitative model of a portion of the Reaction Control System of the NASA Space Shuttle.

The body of the paper is organized as follows. Section 2 defines the specific problem addressed by FORTE. Section 3 presents some background on theory refinement and inductive logic programming. Section 4 presents the details of the refinement algorithm. Sections 5 to 7 present empirical results on benchmark problems in relational learning, logic program debugging, and qualitative modelling, respectively. Section 8 discusses relationships to other work in the area, Section 9 discusses directions for future research, and Section 10 presents our conclusions. Richards (1992) provides more complete details on the system and the experimental results.[1]

2. Task Definition

The objective of this research has been to develop methods for revising first-order theories, and to implement and test the resulting methods in several domains. The specific task addressed is:

- **Given:** An incorrect initial theory and a consistent set of positive and negative instances.
- **Find:** A "minimally revised" theory that is correct on the given instances.

Our terminology is defined as follows:

Theory. A theory is a set of function-free definite program clauses. [2] FORTE views theories as pure Prolog programs. In the family domain, for example, a theory would be a set of clauses defining relationships such as: `father(X,Y) :- parent(X,Y), gender(X, male)`.

Concept. A concept is a predicate in a theory for which examples appear in the training set. Concepts need not be disjoint. In a family domain, concepts might include `father`, `aunt`, and `nephew`.

Instance. An instance is an instantiation (not necessarily ground) of a concept. For example, an instance of the concept `father` is `father(frank, susan)`. Each instance i has an associated set of facts F_i. A positive instance should be derivable from the theory augmented with its associated facts; the negative instances should not. In the family domain, the facts define a particular family, e.g., `parent(frank, susan), gender(frank, male)`.

Correctness. Given a set, P, of positive instances and a set, N, of negative instances, we say that a theory T is *correct* on these instances if and only if

$$\forall p \in P : T \cup F_p \vdash p \quad \text{and} \quad \forall n \in N : T \cup F_n \not\vdash n$$

Derivability is established using standard SLD-resolution, taking the instance to be the initial goal. A set of positive and negative instances is *consistent* if and only if there exists a correct theory for it.

"Minimally revised" theory. A correct theory for a set of instances can be produced trivially by deleting all existing clauses and asserting new clauses that memorize the positive instances, but such a theory is unlikely to be of interest. Ideally, we want the theory to generalize to unseen instances. Since the initial theory is assumed to be approximately correct, a revised theory should be as semantically and syntactically similar to it as possible. FORTE tries to ensure this by using operators that make small syntactic changes and attempting to minimize the number of operations performed.[3]

3. Background

This section provides background that is useful in understanding FORTE. A broader discussion of related work is left until Section 8. FORTE's development is an outgrowth of related work in propositional theory refinement, top-down first-order induction, and inverse resolution; each of which will be discussed briefly.

3.1. Propositional Theory Refinement

A number of researchers have developed propositional theory refinement systems. EITHER (Ourston & Mooney, 1990; Ourston & Mooney, in press) uses a combination of deduction, abduction, and induction to refine a propositional Horn-clause theory. It uses greedy set covering to identify a small set of rules that are responsible for the errors and then adds and retracts rules and antecedents to correct the theory. Although EITHER is limited to propositional domains, it is the conceptual predecessor of FORTE.

KRUST (Craw & Sleeman, 1991) generates a wide array of possible revisions to a knowledge base, and then filters and ranks the revisions to choose the most suitable one. Much of the filtering depends on the existence of certain canonical "chestnut" examples, which must be identified by a human expert. FORTE's overall approach is similar to KRUST's, in that it generates a number of possible revisions, and then selects the one which performs best.

3.2. Top-Down First-Order Induction

FOIL (Quinlan, 1990) is a recent, efficient algorithm for inducing first-order Horn-clause rules. Its outer loop is a greedy covering algorithm that learns one clause at a time. Each clause is constructed to maximize coverage of positive examples while excluding all negatives. Clauses are constructed one literal at a time using hill-climbing. At each step, the literal that maximizes an information-gain metric is added to the clause. Literals are added until all negative examples have been excluded. This hill-climbing technique is efficient, but vulnerable to local maxima. In order to reduce this problem, Quinlan (1991) added *determinate literals*. Given its input arguments, a determinate literal has only one possible binding for its output arguments. FOIL adds all possible determinate literals to a clause before beginning the normal induction process. This is a recursive process, as the new variables introduced by determinate literals can be used to define further determinate literals; hence, an arbitrary depth-bound is imposed. Excess determinate literals are deleted after learning is complete. One of FORTE's techniques for building new rules and specializing existing ones is based on the original FOIL algorithm.

3.3. Inverse Resolution

Inverse resolution is an inductive generalization method introduced by Muggleton and Buntine (1988). Suppose we have the resolution step:

$$\frac{\leftarrow \alpha, \beta \text{ (goal)} \qquad \alpha \leftarrow \delta \text{ (input clause)}}{\leftarrow \delta, \beta \text{ (resolvent)}}$$

If we know the resolvent and either the goal or the input clause, we can abduce the missing element. It is important to note that, when working in first-order logic, inverse resolution operations must take into account variable substitutions, so that any literal appearing in the goal or input clause is (non-strictly) more general than the corresponding literal in the resolvent. CIGOL (Muggleton & Buntine, 1988) used this technique to learn first-order theories from examples; however, it required the user to interactively verify certain steps.

GOLEM (Muggleton & Feng, 1992) is a more efficient, automated induction system based on Plotkin's (1971) framework of relative least-general generalization (RLGG), which Muggleton (1992a) shows to be closely related to inverse reso-

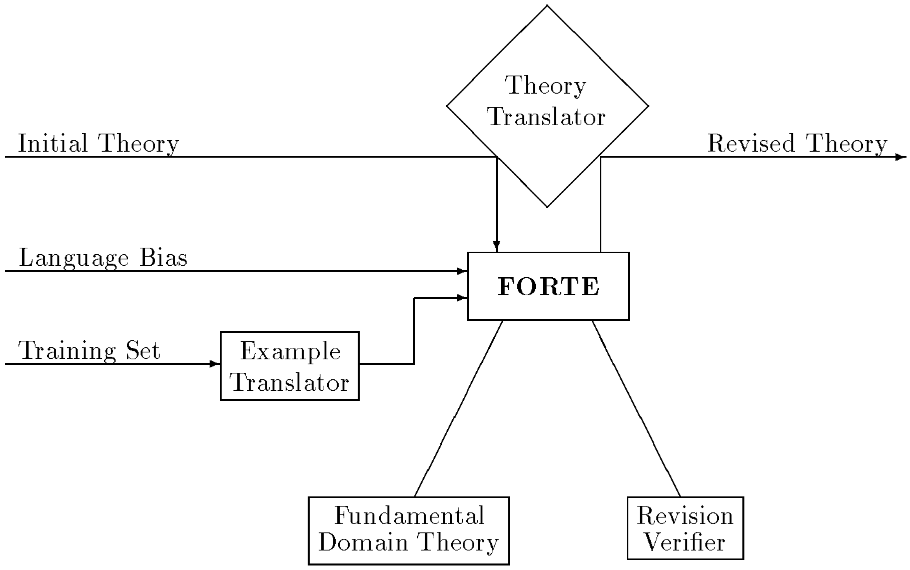

Figure 1. FORTE Interfaces.

lution. GOLEM learns first-order theories "bottom-up," generalizing the positive training instances while excluding the negative instances.

Two of FORTE's theory revision operators are based on inverse resolution. However, unlike CIGOL and GOLEM, FORTE's operators do not require input clauses[4] to be unit clauses.

4. System Description

This section describes the FORTE system. The first subsection looks at FORTE's interface to the outside world. The second subsection examines the theory refinement process itself—how FORTE specializes and generalizes clauses in a theory. The third subsection provides detailed algorithms for the revision operators.

4.1. Interfaces

Figure 1 shows FORTE's interface to the outside world. FORTE itself is represented by the central box. The language bias and the auxiliary modules shown are described below.

Theory translator. The theory translator is an optional module used to translate between the native representation of a theory and the representation required by FORTE. This is necessary when the native representation of a theory is not function-free pure Prolog. The most common use of the theory translator is to replace function symbols in a theory with calls to predicates which calculate the functions.

Example translator. FORTE requires examples to be provided as Prolog terms. As with theories, the FORTE representation may not be convenient in all domains. The example translator can be used to translate between a native domain representation and that required by FORTE.

Language bias. The language bias is used to limit FORTE's search space when the user knows that certain restrictions must apply to the output theory. For example, the user can require the theory to be conjunctive, or to be nonrecursive. Other options in the language bias are described more fully as they apply to the revision operators discussed below.

Revision verifier. The revision verifier is an optional module that allows the user to insert domain-specific consistency checks in the revision process. For example, when working in the domain of qualitative modelling, the revision verifier enforces dimensional consistency in the qualitative equations. To see how this works, suppose a revision operator proposes adding the constraint `derivative`(X, Y) to a clause containing `add`(X, Y, Z). The `derivative` constraint requires X and Y to have dimensions which differ by a factor of *1/time*, while the `add` constraint requires their dimensions to be the same. This is domain specific knowledge provided by the revision verifier for qualitative modelling; when the revision verifier is called to examine this revision, it will detect the dimensional conflict and reject the revision.

Fundamental domain theory. The fundamental domain theory is an optional module which provides a place for predicates which the user wishes to shield from FORTE's revision process. There are two reasons why this might be desirable. First, if some portion of the theory is known to be correct, shielding it in the fundamental domain theory will reduce the space of revisions, thereby speeding FORTE's execution. Second, the fundamental domain theory can provide intensional definitions of the fundamental relations used to define a domain (GOLEM uses an extensional definition of background information the same way). Since these definitions will not be revised by FORTE, they can be written using all the features of standard Prolog.

4.2. Top-Level Algorithm

FORTE revises theories iteratively, using a hill-climbing approach. Each iteration identifies points in the theory, called revision points, where a revision has the potential to improve the theory's accuracy. It then generates a set of revisions, based on the revision points, selects the best one, and implements it. The process iterates until no revision improves the theory. This top-level algorithm is shown in Figure 2.

In order to generate revision points, the current theory is tested on the training set. FORTE annotates failed proofs of positive instances and successful proofs of negatives. From these annotations it identifies points in the theory for possible revision (see Section 4.2.1). Each revision point has a *potential*, defined as the

```
repeat
     generate revision points
     sort revision points by potential (high to low)
     for each revision point
          generate revisions
          update best revision found
     until potential of next revision point is less
          than the score of the best revision to date
     if best revision improves the theory
          implement best revision
     end if
until no revision improves the theory
```

Figure 2. Top-level refinement algorithm.

maximum increase in theory accuracy which could result from a revision of that point. For example, if a particular clause was used in successful proofs of five negative instances, then specialization of that clause has a potential of five.

FORTE then generates a set of proposed revisions from the revision points, beginning with the point that has the highest potential and working down the list. Each revision receives a score, which is the actual increase in theory accuracy it achieves; FORTE retains the single best revision generated so far, where the best revision is the one increasing accuracy the most (in case of a tie, FORTE chooses the revision resulting in the smallest theory). FORTE stops generating revisions when the potential of the next revision point is less than the actual accuracy increase of the best revision generated to date. At that point, the best revision is implemented, and the cycle begins again. Since we require an increase in accuracy on each iteration, and accuracy is limited to 100%, this algorithm is guaranteed to terminate.

This process continues until FORTE is unable to generate any revisions which improve the theory. At this point, we hope to have developed a theory that is correct on the training set. However, since this is a hill-climbing process, FORTE can be caught in local maxima. We minimize this danger in two ways. First, revisions are developed and scored using the entire training set, rather than just a single instance; this global vision gives us better direction than if revisions were developed from single instances. Second, FORTE uses a variety of different operators to generate possible revisions. Since the operators have different strengths and weaknesses, they can escape different types of locality problems.

4.2.1. Generating revision points

Revision points are places in a theory where errors may lie. They are of two types: specialization points and generalization points. We identify revision points by annotating proofs or attempted proofs of misclassified instances. Points in the theory where proofs of positive instances fail are places where the theory may need to be generalized, and clauses used in successful proofs of negative instances are points where the theory may need to be specialized. The number of different instances which flag a particular point represents its potential, i.e., the maximum increase in theory accuracy that could be gained by revising the theory at that point.

Generating specialization revision points is simply the process of noting which clauses participate in proofs of negative instances; these clauses become the revision points.

Generating revision points for generalization is more complex because we have three kinds of generalization operators. Some generalization operators are antecedent-based, meaning that their revisions target a particular antecedent in a particular clause, some are clause-based, and some are predicate-based. We must generate revision points for each of these operator types. However, all of these revision points are generated from annotations made from failed proofs of positive instances.

The annotation process works as follows: Each time we backtrack, we note which antecedent in which clause failed; this antecedent is a *failure point*. In addition, we must consider which other antecedents may have contributed to this failure, perhaps by binding variables to incorrect values. These antecedents are called *contributing points*. As an example, consider the following program:

```
sister(A, B) :- daughter(A, C), parent(C, C).
daughter(A, B) :- gender(A, female), parent(B, A).
```

If we try to execute the `sister` predicate, the unprovable `parent` antecedent will be marked as a failure point. The `daughter` antecedent instantiates variable `C`, and so is marked as a contributing point. Within the `daughter` predicate, the parent predicate instantiates variable `B`, and is therefore also marked as a contributing point. The `gender` antecedent is neither a failure point nor a contributing point, and so is not marked and will not be subject to revision. No subsequent distinction is made between failure points and contributing points; all of the underlined antecedents become antecedent-based revision points.

We create clause-based revision points for all clauses in which we made an annotation. The potential of a clause-based revision point is the number of distinct instances that marked any antecedent within it. These revision points are used by clause-based operators, which revise a single clause without regard for any particular antecedent. In the above example we would have two clause-based revision points, since both clauses contain annotations.

Predicate-based revision points are the next step beyond clause-based revision points. A predicate-based revision point is created for each theory predicate that appears as a marked antecedent in the annotated theory. In other words, since we marked `daughter(A, C)` in the theory, we create a predicate-based revision point for `daughter`. Predicate-based revision points have a potential equal to the number of distinct instances that annotated a call to the predicate anywhere in the theory. These revision points are used by the operator identification, which seeks to generalize the definition of the predicate, without reference to any particular clause.

4.2.2. *Special provisions*

There are two types of theories, as specified by the language bias, for which FORTE makes special provisions: recursive theories and most-specific theories. These provisions are discussed below.

Recursive theories. Revising a recursive theory is substantially more difficult than revising a nonrecursive one. With nonrecursive theories, we can treat the predicate under revision in isolation from the rest of the theory. If the predicates appearing as antecedents contain slight errors, we will still be able to develop a revision for the chosen predicate. If the antecedents contain gross errors, the proposed revision may simply eliminate them as antecedents. When revising a recursive theory, we inevitably need to evaluate a recursive call to the very predicate we are revising. Since we are revising it, we can be almost certain that the results of evaluating the recursive call will be incorrect.

In order to solve this problem, we must decouple our evaluation of a recursive call from the definition of the predicate that we are revising. The training set provides us with a way to do this; we can use the positive instances in the training set as an extensional definition of the predicate. By using this extensional definition to evaluate recursive calls, we allow the revision process to work unhindered by the complications of recursion. GOLEM and FOIL also use extensional definitions to handle recursion. After the revision has been developed, we can test its effectiveness using normal resolution.

Unfortunately, using the training set as an extensional definition works only if the training set contains all instances that will be generated during well-founded recursion from other instances present. For example, if we are learning a definition of list reversal, and we wish to prove the example `reverse([a,b,c],[c,b,a])`, then the training set must contain the examples `reverse([b,c], [c,b])` and `reverse([c], [c])`. If either of these instances is missing, our proof will fail and we may not be able to develop a correct revision. Since the user is not expected to know what recursion scheme is appropriate for the theory, this means that the training set should contain a complete set of examples below a certain size. For example, our data set for `reverse` contains all permutations of all lists of length-2 and smaller, plus one example of a length-3 list, using the symbols `a, b,` and `c`.

If the recursive predicate we wish to revise is not a top-level predicate for which we have training data, FORTE derives a temporary training set for the predicate from the top-level predicates. This process works well if the higher-level predicates are correctly defined, but may develop different predicates than expected if the higher-level predicates contain errors.

To see how we derive a training set, suppose we have the following correct definition for `subset`:[5]

```
subset([], A).
subset([Elt|Elts], Set) :-
     member(Elt, Set),
     subset(Elts, Set).
```

To derive a training set for `member`, we start the proofs of all positive instances for the subset predicate, and collect the instantiated calls to `member` made at the top-most level (i.e., we do not descend into the recursive calls, since the results of doing so depend on the correct functioning of `member`, which is the predicate we are seeking to revise). These calls become the training set for the member predicate. We thus have the following correspondence between subset instances and derived member instances:

```
subset([a], [a])         --- member(a, [a])
subset([a], [a,b])       --- member(a, [a,b])
subset([b,c], [b,c])     --- member(b, [b,c])
subset([a,b], [a,b,c])   --- member(a, [a,b,c])
```

This process can be viewed as abduction, as in Wirth & O'Rorke (1991).

After revising a theory, it is tested by normal meta-interpretation (i.e., without intercepting recursive calls and using the training set as an extensional definition). Nontermination on an example is considered to be a false classification, and is detected by means of a depth limit.

FORTE's effectiveness in revising recursive theories depends on the theory being revised; refer to Section 6.3 for a more complete discussion of its limitations.

Most-specific theories. In some domains, negative examples are not available and we wish to develop the most-specific theory which fits the positive instances as tightly as possible. In order to prevent simple memorization, FORTE requires that most-specific theories be conjunctive (i.e., they must consist of a single clause).

An example of a domain requiring a most-specific theory is qualitative modelling. Given a set of observed system behaviors, we wish to develop a model that reproduces those behaviors; negative behaviors are not normally available.[6] Hence, we ask FORTE to develop the most constrained model that accounts for all of the given positive behaviors. Since a model is a conjunction of one or more constraints, this is naturally a conjunctive theory.

In order to develop a most-specific theory, FORTE follows the normal revision process to generalize the input theory as necessary to allow all positives to be provable. It then makes the theory as specific as possible by adding all possible antecedents which do not eliminate any of the positive instances. In order to ensure that this process is finite, we do not allow the antecedents added in the second step to introduce new variables.

4.3. Revision Operators

In order to be able to repair arbitrarily incorrect theories, revision operators must be able to transform any theory in the language into any other. This can be done with four basic revision operations: adding rules, deleting rules, adding antecedents to a rule, and deleting antecedents from a rule. A simple implementation of these basic operations would produce a workable theory revision system. However, such a system would often find itself trapped in local maxima or lost on local plateaus. FORTE's operators are designed to more quickly develop useful revisions to a theory, for example, by adding several antecedents at once until a desired goal is reached. However, FORTE's operators can often best be understood by remembering that they are ultimately composed of these basic revision operations.

The following subsections describe FORTE's revision operators. For complex operators, we also give explicit algorithms. Although the operators are described in terms of the changes they make to the theory, recall that each operator is developing a *proposed* revision, and that the revision will be implemented only if it is the best revision developed by any operator for any revision point in the current revision cycle.

Conceptually, each operator develops its revision using the entire training set. However, in practice, this is usually unnecessary. For example, when specializing a clause, we will not change the provability of any unprovable instance, or of any provable instance whose proof does not rely on the clause being specialized. Hence, we can develop the revision using a subset of the training set consisting only of those provable instances whose proofs rely on the target clause. Similar subsets are used for generalization as well.

The operators are illustrated using examples in the domain of family relationships. Part of one of the family data sets used by Hinton (1986) is shown in Figure 3. Horizontal lines denote marriage relationships and the remaining lines denote parental relationships.

4.4. Operators for Specialization

FORTE specializes clauses when they are used to prove negative instances. A clause may be specialized by being deleted (operator *delete-rule*), or by having antecedents added to it (operator *add-antecedent*). These operators are described below.

4.4.1. Operator delete-rule

The simplest way to specialize a clause is to delete it. There are two restrictions. First, if the clause is the only base case of a recursive predicate (i.e., a predicate that currently has one or more recursive clauses) then it cannot be deleted, as doing so would invalidate the recursive clauses as well. Second, if this is the only clause for a top-level concept, we replace the deleted clause with the rule

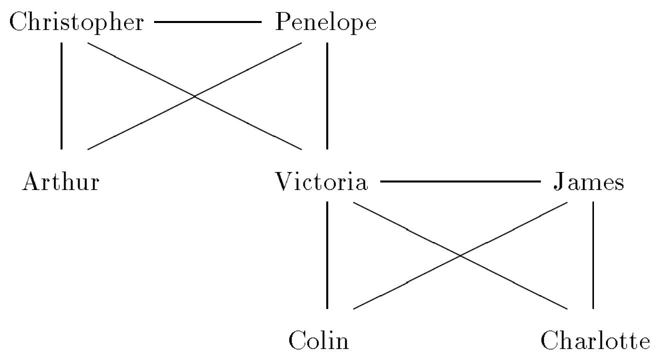

Figure 3. A portion of Hinton's family data.

```
concept :- fail.
```

This provides us with a starting point for later revisions to the predicate.

4.4.2. Operator add-antecedent

Another approach is to specialize a clause by adding antecedents to discriminate between positive and negative instances. FORTE adds antecedents to a clause in an attempt to make all negative instances unprovable. If adding these antecedents also makes some positive instances unprovable, FORTE adds the specialized clause to the theory and begins again with the original clause, looking for alternate specializations that retain the proofs of the other positive instances while still eliminating the negatives. This process continues until we have a set of clauses that retains the provability of all of the originally provable positive instances.

FORTE provides two separate algorithms for producing a specialized clause: hill-climbing antecedent addition and *relational pathfinding* (Richards & Mooney, 1992). As shown in Figure 4, both methods are used to develop specializations of a clause, and the one with the best performance is selected. In practice, these two methods of specializing clauses are complementary; certain types of revisions are performed well by one but not the other.

Hill-climbing antecedent addition. The hill-climbing method of antecedent addition is based on the original FOIL algorithm. Our implementation departs from FOIL in one respect: FORTE does not maintain "tuples" as FOIL does. FOIL's tuple-based approach counts the number of proofs of instances, whereas FORTE keeps track of the number of provable instances (ignoring the fact that one instance may be provable in several different ways).

```
repeat
     specialize clause by hill-climbing
     specialize original clause by relational pathfinding
     choose specialized clause covering the most positives
     add chosen clause to the revision
until all positives covered by original clause are covered
     or no specialized clause can be generated
```

Figure 4. Algorithm for add-antecedent.

The language bias may be used to limit the types of antecedents considered for addition, e.g., in a nonrecursive theory, recursive antecedents would not be allowed. Clearly invalid or redundant antecedents are also not generated, for example, relational antecedents must contain at least one variable that already appears in the current clause.

The FOIL approach is quite effective in many cases, particularly for developing recursive base-cases and for adding non-relational antecedents to a rule. However, as with any hill-climbing method, it can be caught by local maxima. Local plateaus can also occur when there are a number of antecedents that do not decrease accuracy; but in order to actually increase accuracy, several antecedents must be added at once. We can see the local plateau problem by trying to define the grandparent relation using only the instances below and the data shown in Figure 3.

```
(+)  grandfather(christopher, colin)
(-)  grandfather(christopher, arthur).
```

There is no single antecedent that we can add which will allow the positive instance to be proven while making the negative instance unprovable. Both Colin and Arthur have parents, neither has children, and neither is married. Even determinate literals would not help in this example, since all parents have two children and all children have two parents. In order to create a correct theory, we must simultaneously add both of the required parent relationships, i.e.,

```
grandparent(x, y) :- parent(x, z), parent(z, y).
```

To do this, we need a method which is capable of searching for relationships among the constants in a domain. Our method for accomplishing this is called *relational pathfinding*.

Relational pathfinding. Relational pathfinding (Richards & Mooney, 1992) is a method of antecedent addition designed to escape local maxima and local plateaus. The idea of pathfinding in a relational domain is to view the domain as a (possibly infinite) hypergraph of terms linked by the relations that hold among the terms. Our underlying assumption is that, in most relational domains, important concepts will be represented by a small number of fixed paths among the terms defining a positive instance. For example, in the "grandfather" example, constants satisfying the relation are joined by a single fixed path consisting of two parent relations.

```
instantiate rule with a randomly chosen positive instance
find isolated sub-graphs
for each sub-graph
     terms become initial end-values
end for
repeat
     for each sub-graph
          expand paths by one relation is all possible ways
          remove paths with previously seen end-values
     end for
until intersection found or resource bound exceeded
if one or more intersections found
     for each intersection
          add path-relations to original rule
          if the new rule contains new singleton variables
               add relations using the singleton variables
               if all singletons cannot be used
                    discard the rule
               end if
          end if
          replace terms with variables
     end for
     select most accurate rule
end if
if selected rule allows negatives
     specialize using hill-climbing
end if
```

Figure 5. Algorithm for relational pathfinding.

Relational pathfinding can be used any time a clause needs to be specialized and does not have relational paths joining all of its variables. If, after pathfinding, the rule is still too general, we do further specialization using hill-climbing. This arises, for example, when a rule requires non-relational antecedents.

Relational pathfinding as described in Figure 5 finds paths by successive expansion around the nodes associated with the terms in a positive example, in a manner reminiscent of Quillian's (1968) spreading activation. We arbitrarily choose one misclassified positive instance and use it to instantiate the initial rule. The terms in the instantiated rule are nodes in the domain graph, possibly connected by antecedents in the rule. We then identify isolated subgraphs among these terms; if the initial rule contains no antecedents, then each term forms a singular subgraph.

We view a subgraph as a nexus from which we explore the surrounding portion of the domain graph. Each exploration that leads to a new node in the domain graph is a path, and the term at the node it has reached is the path's *end-value*. Initially, each term in a sub-graph is the end-value of a path of length zero.

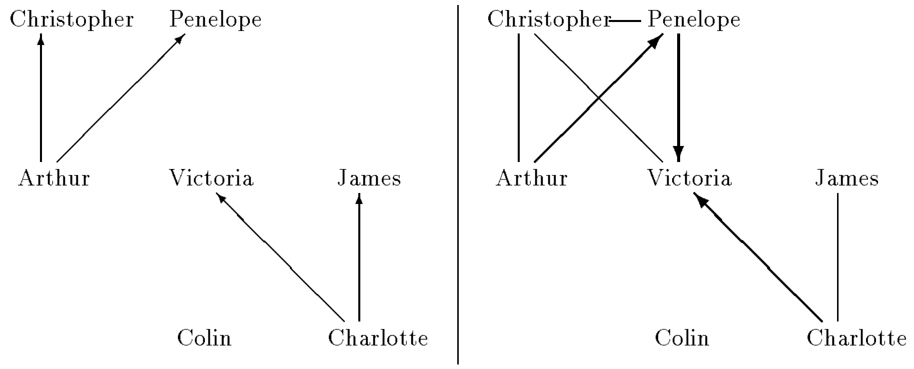

Figure 6. Learning the concept `uncle` with relational pathfinding.

Taking each subgraph in turn, we find all new terms that can be reached by extending any path with any defined relation. These terms form a new set of path end-values for the subgraph. We check this set against the sets of end-values for all other subgraphs, looking for an intersection. If we do not find an intersection, we expand the next node. This process continues until we either find an intersection or exceed a preset bound on the maximum path-length we will consider. There is also a (very high) limit on the number of new paths generated when expanding nodes, intended to prevent termination problems when working in infinite domains.

When we find an intersection, we add the relations in the intersecting paths to the original instantiated rule. If the new relations have introduced new terms that appear only once, we try to complete the rule by adding relations that hold between these singletons and other terms in the rule; these new relations are not allowed to eliminate any of the currently provable positive instances. If we are unable to use all of the new singletons, the revision is rejected.

Finally, we replace all terms with unique variables to produce the final, specialized theory clause. If we simultaneously discover several intersections, we develop clauses for each of them separately and choose the one that provides the best accuracy on the training set.

As an example, suppose we want to learn the uncle relationship, given an initially empty rule and the positive instance `uncle(arthur, charlotte)`. This process is illustrated in Figure 6. We begin by exploring paths from the node labelled `Arthur`, which leads us to the new nodes `Christopher` and `Penelope`. We then expand from the node labelled `Charlotte`, leading to the nodes `Victoria` and `James`. At this point we still do not have an intersection, so we lengthen all paths originating from node `Arthur`. We eliminate any end-values that we have already used (and which, therefore, do not give us an intersection). This leaves only one value: `Victoria`. Since `Victoria` is also an end-value of one of the paths originating from `Charlotte`, we recognize an intersection.

There are actually two paths leading from `Arthur` to `Victoria`, but in this case they are isomorphic (merely leading through different grandparents). If we had found several paths, we would select the one providing the best overall accuracy. The final path in this example is

```
uncle(X, Y) :- parent(Z, X), parent(Z, W), parent(W, Y)
```

Since relational pathfinding is only able to add *relations* as antecedents, it calls on the hill-climbing method of antecedent addition to complete its clauses. In this case, hill-climbing antecedent addition would need to add the literal `gender(X, male)` to remove any remaining negatives.

4.5. Operators for Generalization

FORTE generalizes a predicate when a positive instance is unprovable. It uses four operators to perform generalization. Two methods are similar to methods used in propositional theory revision: adding new rules and deleting antecedents from existing rules. The second two are variants of the inverse-resolution operators absorption and identification.

4.5.1. Operator delete-antecedent

In many cases, FORTE may be able to create a good revision simply by deleting antecedents from an existing clause. In order to develop a revision, we generalize the original clause to cover as many positives as possible, without allowing proofs of any negatives. We then add the generalized clause to the theory. If there are more positives to be covered, we begin again with the original clause and repeat the process. We stop when all of the positive instances listed in the revision point are provable or we are unable to generalize the original clause to allow proof of any of the remaining unprovable instances.

We have two methods at our disposal. First, we try a hill-climbing approach. This method deletes one antecedent at a time, selecting each time the antecedent whose deletion allows the most unprovable positives to be proven. As with any hill-climbing approach, this is efficient but vulnerable to local maxima. If this approach fails, we use a more general method that can delete multiple antecedents simultaneously.

Hill-climbing antecedent deletion. This method of deleting antecedents is iterative. It tries deleting each antecedent in the specified clause, and notes two things: how many unprovable positives can be proven when the antecedent is deleted, and whether any negatives become provable as a result of its deletion. We select the antecedent that allows proof of the largest number of positives while not allowing any negatives to be proven. This antecedent is deleted, and the process repeats. We stop when there are no more antecedents whose deletion gains us anything.

```
repeat
     for each antecedent in the clause
          if deletion does not allow provable negatives
               count number of positives deletion makes provable
          end if
     end for
     delete antecedent allowing the most positives
until no antecedent can be deleted without proving negatives
```

Figure 7. Algorithm for hill-climbing delete-antecedent.

This approach to deleting antecedents may fail for two reasons. First, it may be that we need to add new discriminating antecedents to the clause after generalizing it. In this case, the add-rule operator is likely to propose a useful revision. Second, the clause may be so over-specialized that we need to delete several antecedents at once in order to affect the provability of any instance. This local plateau problem is dealt with by the technique for deleting multiple antecedents.

Deleting multiple antecedents. This method is much more computationally expensive than the hill-climbing approach to antecedent deletion, since it must try deleting combinations of antecedents. Because of this expense, it is not used if hill-climbing antecedent deletion successfully develops a revision.

To generalize a clause, we first collect all antecedents whose (individual) deletion does not allow any negative instance to be proven. None of these deletions will, by itself, allow positive instances to be proven either, or the hill-climbing approach to antecedent deletion would have found them. We generate combinations of these antecedents, looking for a combination whose deletion allows proof of one or more positives but no negatives.

We build combinations of deletions one antecedent at a time, working left-to-right through the clause. When we delete an antecedent, we check to see if any negatives have become provable. This allows us to substantially prune the search space, as, if negatives have become provable, we discard not only this particular combination but all supersets of it. We do not stop when positives have become provable—we delete as many antecedents as we can, covering as many positives as possible.

4.5.2. Operator add-rule

Add rule is a clause-based generalization operator that develops one or more new versions of an existing rule, while leaving the original rule in the theory. Its objective is to create a new rule that allows proof of the positive instances that identified the original rule as a failure point. Building this rule is a two-step process.

First, we create a generalized rule containing only the core of antecedents essential to keep negatives from being proven, while not interfering with proofs of positives. To do this, we copy the original rule, delete antecedents whose deletion does not

```
find all antecedents whose deletion does not allow provable negatives
repeat
     consider an antecedent for deletion
     if negatives are provable
          prune this branch of the search space
     else
          delete this antecedent
     end if
until no antecedents left to try
if one or more positives have become provable
     propose generalized clause as a revision
end if
```

Figure 8. Algorithm for delete-multiple-antecedents.

allow any negatives to be proven, and also delete antecedents whose deletion allows one or more previously-unprovable positives to be proven (even if doing so allows proofs of negatives). This is done in the same way as hill-climbing antecedent deletion (see above).

Second, we create one or more specializations of this core rule, which will allow proofs of the desired positives while eliminating the negatives. We do this by passing the rule to the add-antecedent operator described earlier.

4.5.3. Operator identification

Identification is a predicate-based operator which attempts to generalize the theory by creating a new rule for an existing predicate. It constructs a new clause to generalize the definition of an antecedent that caused one or more proofs of positive instances to fail. Rather than developing the clause from scratch, it performs an inverse resolution step using two existing rules in the domain theory. For a complete definition of the inverse resolution operators, refer to Muggleton (1992a).

Suppose that our initial theory of family relationships includes the following rules, where `aunt_uncle` is intended to be a general rule for identifying aunts and uncles without regard to gender.

```
uncle(A, B) :- gender(A, male), aunt_uncle(A, B).
uncle(C, D) :- gender(C, male), sibling(C, E), parent(E, D).
aunt_uncle(A, B) :- married(A, C), sibling(C, D), parent(D, B).
aunt(A, B) :- gender(A, female), aunt_uncle(A, B).
```

When we are presented with an instance of an aunt who is a blood relative, this instance will not be provable. One of the failure points is the call to `aunt_uncle`. Identification looks for ways to provide another rule for this predicate, and finds one in the two rules for `uncle`. The proposed revision replaces the second `uncle` clause with the new rule

```
aunt_uncle(A, B) :- sibling(A, E), parent(E, B).
```

4.5.4. *Operator absorption*

Absorption is the complement of identification. Rather than constructing a new clause for the predicate corresponding to a failing antecedent, absorption looks for an existing clause whose antecedents subsume the failing antecedent (and possibly other antecedents in the clause), and which has alternate clauses that will allow the failing positive instances to be proven. For example, suppose our theory includes the rules

```
uncle(A, B) :- gender(A, male), sibling(A, C), parent(C, B).
aunt_uncle(D, F) :- sibling(D, E), parent(E, F).
aunt_uncle(A, B) :- married(A, C), sibling(C, D), parent(D, B).
```

When we are presented with an instance of an uncle who is not a blood relative, we will not be able to prove it using this theory. We will have a failure point either at `sibling` or `parent`. Absorption finds similar antecedents in the second `aunt_uncle` clause. Thus, it replaces the `uncle` rule with the new rule

```
uncle(A, B) :- gender(A, male), aunt_uncle(A, B).
```

5. Experimental Results in the Family Domain

In this section, we examine FORTE's performance in the domain of family relationships, a standard benchmark problem in relational learning (Quinlan, 1990). Richards (1992) also presents results on another standard benchmark problem, illegal chess positions for king-rook-king endgames. The results indicate that FORTE improves the accuracy of randomly corrupted theories and produces more accurate theories than pure inductive learning. Since FORTE's hill-climbing techniques make it vulnerable to local maxima, another important aspect of learning is the accuracy of revised theories on the training data. In practice, FORTE has very little trouble with local maxima. In over 1300 test runs, FORTE was caught in local maxima only nine times (0.69%); in all cases the accuracy of the revised theory on the training data was greater than 98%.

5.1. Description of Data and Initial Theories

Our earlier examples used the family data employed by Hinton (1986) and Quinlan (1990). While the simplicity of this data makes it suitable for examples, it includes a great deal of artificial structure (for example, all married couples have two children, one boy and one girl). In order to provide a more realistic test, we created a large, diverse family composed of 86 people across 5 generations. This domain uses

```
wife(X, Y) :- gender(X, female), married(X, Y).
husband(X, Y) :- gender(X, male), married(X, Y).
mother(X, Y) :- gender(X, female), parent(X, Y).
father(X, Y) :- gender(X, male), parent(X, Y).
daughter(X, Y) :- gender(X, female), parent(Y, X).
son(X, Y) :- gender(X, male), parent(Y, X).
sister(X, Y) :- gender(X, female), sibling(X, Y).
brother(X, Y) :- gender(X, male), sibling(X, Y).
aunt(X, Y) :- gender(X, female, au(X, Y).
uncle(X, Y) :- gender(X, male, au(X, Y).
niece(X, Y) :- gender(X, female), au(Y, X).
nephew(X, Y) :- gender(X, male), au(Y, X).
au(X, Y) :- sibling(X, B), parent(B, Y).
au(X, Y) :- married(X, A), sibling(A, C), parent(C, Y).
sibling(X, Y) :- parent(A, X), parent(A, Y), X \= Y.
```

Figure 9. A correct theory for family relationships.

the same twelve concepts as Hinton's data: husband, wife, mother, father, sister, brother, son, daughter, aunt, uncle, niece, and nephew.

The family data set includes 744 positive instances and 1488 randomly generated negative instances. Every test run used an independent, randomly selected subset of these instances as the training set, with the remaining instances used as the test set. The background facts provide the gender of each person, all marriages, and all parent-child relationships.

The theory revision tests used randomly corrupted versions of the correct theory shown in Figure 9. The number of errors introduced in each corrupted theory depended on the test being run (see below). Six types of errors could be introduced:

- Delete rule
- Add rule (1-3 antecedents)
- Delete antecedent
- Add antecedent
- Change antecedent (delete plus add)
- Change variable

When adding a new antecedent, there was a 50% chance that the antecedent used would be taken from elsewhere in the theory, and a 50% chance that it would be newly constructed. When changing a variable, there was a 50% chance that it would be changed to a variable appearing elsewhere in the same clause and a 50% chance that it would be a new variable.

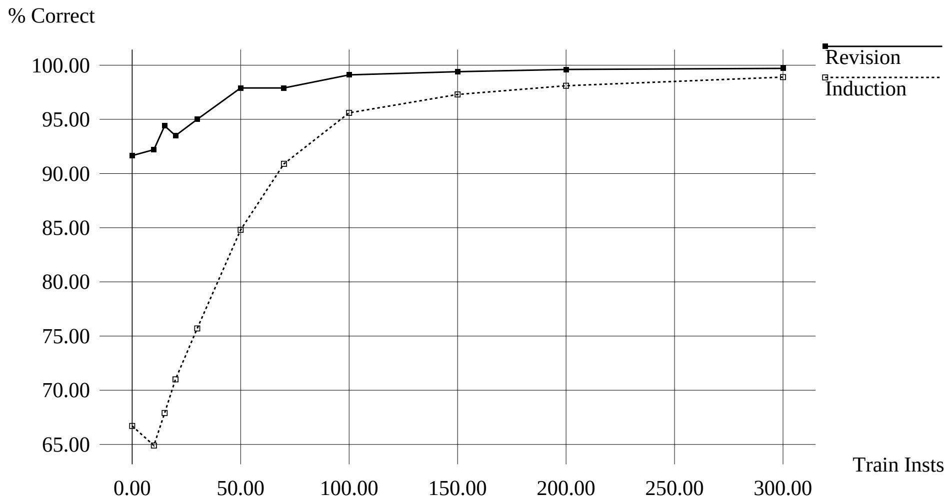

Figure 10. Refinement performance in the family domain.

5.2. Theory Refinement Performance

The basic premise of theory refinement is that it is better to revise a theory that is approximately correct than it is to induce a new theory from scratch. In order to verify this premise, we generated five corrupted theories, containing an average of 3.6 errors each. Their average initial accuracy was 91.65%. Figure 10 shows a revision learning curve, averaged across four runs on each of the five theories, and an induction curve (i.e., FORTE revising an empty initial theory) averaged over 20 trials. A statistical t-test revealed that the difference between the curves at all training-set sizes is statistically significant ($p < 0.01$). These results show that beginning with an approximate domain theory not only provides an initial boost in accuracy, but also that this advantage is maintained as the training set size increases.

Another performance issue in theory refinement is how a system responds to increasing degradation of the initial theory. A good system will degrade gracefully as the accuracy of the input theory decreases. To illustrate this characteristic of FORTE, we created five series of increasingly corrupted theories. Each series contains four theories, containing from two to eight errors each. We fixed the training set size, and ran FORTE four times on each corrupted theory in each series, and then averaged the results for each level of corruption (i.e., we averaged together the 20 runs on theories containing two errors, the 20 runs on theories containing four errors, and so forth). We repeated this experiment for training set sizes of 50 and 100 instances. The results appear in Figure 11.

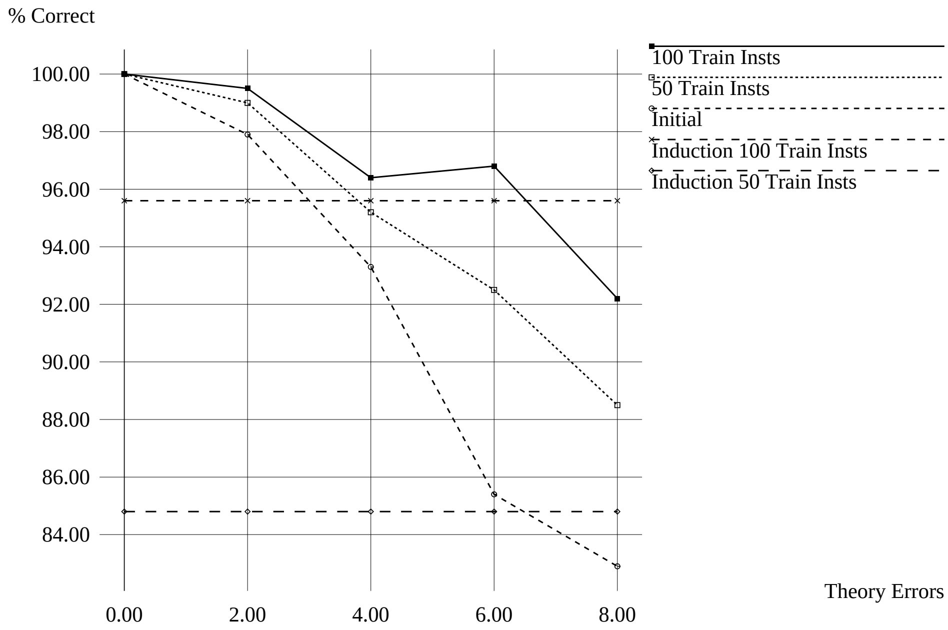

Figure 11. Degradation due to initial theory corruption in the family domain.

The lowest curve shows the accuracy of the initial corrupted theories. The center curve shows the accuracy of FORTE's theories when it is given 50 training instances. The highest curve shows FORTE's accuracy when it is given 100 training instances. As expected, increasingly inaccurate initial theories do lower the accuracy of FORTE's revised theories for a given training set size. However, the degradation is gradual, and FORTE's output theories are always significantly better than the input theories ($p < 0.01$). Also shown are accuracies of pure induction for 50 and 100 examples. With 50 examples, revision is always better than induction. With 100 examples, revision is better up to six theory errors. With eight theory errors, as many as half of the rules in the initial theory may be corrupted (the initial theory contains 15 rules), and FORTE performs better when allowed to induce a new theory.

5.3. Inductive Performance

The family domain is a prototypical first-order domain, in that it depends heavily on relations such as `parent(X, Y)` and `married(X, Y)` that cannot easily be translated into a propositional representation. Much of FORTE's performance in

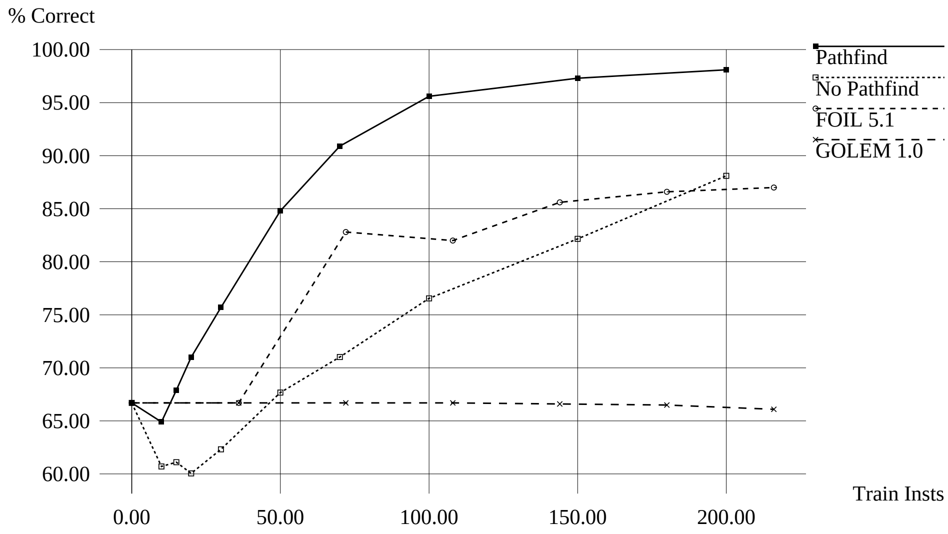

Figure 12. Inductive performance in the family domain.

highly relational domains of this sort comes from relational pathfinding (Richards & Mooney, 1992). To demonstrate this, Figure 12 shows FORTE performing inductive learning both with and without relational pathfinding. These curves are averaged over 20 runs for each data point. The difference between them is statistically significant ($p < 0.01$) at all points.

Figure 12 also includes learning curves for FOIL version 5.1 and GOLEM version 1.0α, also averaged over 20 trials. Since GOLEM and FOIL only learn one concept at a time, each trial actually consisted of a run on each of the twelve family concepts, using training sets one-twelfth of the size noted. FOIL's accuracy is ultimately limited by its inability to learn the concepts `aunt, uncle, niece,` and `nephew`. GOLEM performs poorly on all concepts, generally just memorizing the positive instances. Since the `parent` relation is not determinate, GOLEM is simply unable to learn in this domain.

Finally, we ran induction experiments in two domains, family and king-rook-king, to determine how FORTE scales with increasing amounts of data. As one would expect, FORTE's complexity is exponential in the size of the input theory, and in the arity of the theory predicates. For example, when FORTE considers new antecedents for addition to a rule, the number of permutations of arguments to a predicate is an exponential function of the predicate's arity. However, FORTE's complexity for a given learning problem, where these items are fixed, is at most quadratic in the size of the training set. This complexity result is demonstrated in Figure 13. This is a log-log graph, which means that polynomials show as lines,

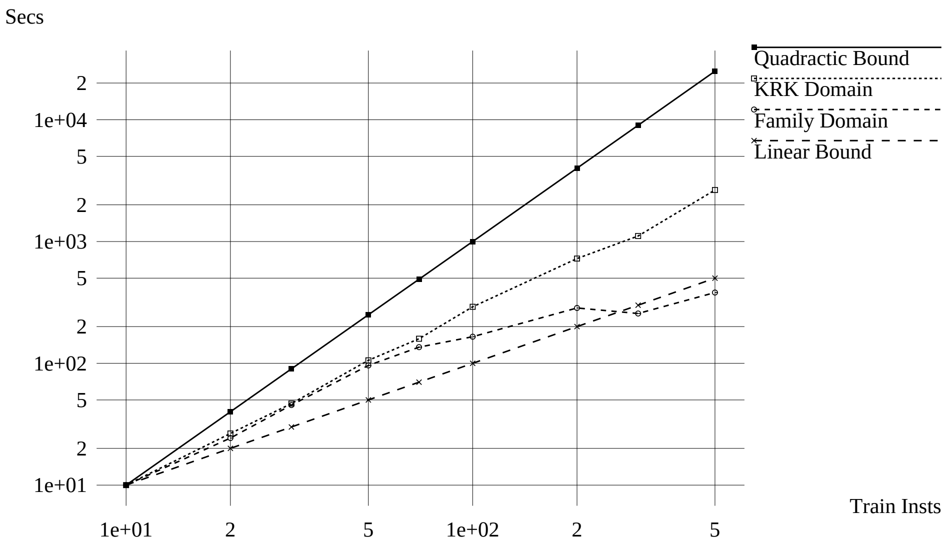

Figure 13. Time complexity of training.

with the slope of the line proportional to the degree of the polynomial. The lower and upper lines show linear and quadratic bounds respectively. The learning times for both of the test domains fall between these two bounds.

6. Application: Logic Programming

Since FORTE represents theories as Prolog programs, we can view theory induction and refinement as logic program synthesis and debugging, respectively. However, the logic programming domain is substantially different from most machine learning domains. First, logic programs are highly recursive. Second, while we may be satisfied with a highly accurate classification theory, one is usually not happy with a "mostly correct" program. Consequently, we test FORTE's performance in this domain differently. Instead of producing learning curves showing increasing accuracy with larger training sets, we show that, given sufficient training data, FORTE will produce a completely correct program.

6.1. Program Synthesis

Although designed as a theory refinement system, FORTE is able to inductively synthesize simple logic programs from examples of desired behavior. As discussed

Table 1. Summary of program synthesis results.

Program	Training Set	No Pathfinding	Normal FORTE	GOLEM 1.0	FOIL 5.1
member	21 instances	4 seconds	4 seconds	7 seconds	1 second
append	39 instances	*failed*	21 seconds	*failed*	3 seconds
directed path	121 instances	25 seconds	24 seconds	*failed*	1 second
insert after	35 instances	30 seconds	50 seconds	*failed*	*failed*
merge sort	60 instances	*failed*	199 seconds	*failed*	17 seconds
naive reverse	38 instances	*failed*	207 seconds	*failed*	2 seconds

in Section 4.2.2, in order to correctly synthesize or revise a recursive theory, FORTE requires the training set to provide a complete extensional definition for a subset of the problem domain.

Table 1 presents a summary of several standard program synthesis problems to which FORTE has been applied.[7] In all of these cases, correct definitions were given for any necessary lower-level predicates. For example, the `merge-sort` problem provides definitions for `split` and `merge`.

The first column in the table identifies the program to be synthesized. The second column shows the size of the training set that was provided. The third column gives the run-time required for the synthesis with relational pathfinding disabled. The fourth column gives the run-time for the synthesis with relational pathfinding enabled. The fifth and sixth columns give run-times for GOLEM version 1.0α and FOIL version 5.1 respectively. All run-times are for a SPARCstation 2.

Where run-times are shown, a correct program was synthesized. Where the annotation *failed* appears, the system did not generate a correct program. FORTE learned correct programs for all six problems. Relational pathfinding was essential for FORTE to correctly synthesize the recursive clause for three of the six programs. FOIL also performed quite well, missing only one of the problems and executing more quickly due to its efficient implementation and more limited search. GOLEM fared especially badly, as it tended to either memorize the positive instances or to produce infinite recursions. For example, for naive reverse, GOLEM produced the recursive clause: `reverse(A, B) :- reverse(B, A).`

6.2. Debugging Student Programs

In order to provide a realistic test of FORTE's logic program debugging capabilities, we asked students in an undergraduate class on programming languages to hand in their first attempts at writing simple Prolog programs. They gave us their programs after they had satisfied themselves on paper that the programs were correct, but before they tried to run them. The student programs were distributed among three problems: find a path through a directed graph, insert an element into a list, and merge-sort a list. We collected 23 distinctly different incorrect programs, representing a wide variety of errors ranging from simple typographical mistakes to complete misunderstandings of recursion.

Table 2. Summary of program debugging results.

Program	# of Programs	Training Set	Mean Revision Time	% Correct
directed path	4	121 instances	87 seconds	100%
insert after	9	35 instances	82 seconds	100%
merge sort	10	60 instances	199 seconds	100%

FORTE was able to debug all of these programs (see Table 2). The training sets were the same as those FORTE used to synthesize these same programs. Since FORTE is able to synthesize all of these programs, it is perhaps no surprise that it was able to debug them as well. However, what is noteworthy is FORTE's ability to debug the programs while preserving the basic structure provided by the program author. For example, consider the correct program for finding a path through a directed graph (this is the program FORTE synthesizes):

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

One student's attempt at writing this program was:

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

Even though this program is highly inaccurate, FORTE was able to preserve both the one correct base case and the unusual recursion scheme. FORTE's revised program is:

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

6.3. Debugging Deeply Recursive Programs

FORTE is able to repair top-level recursive predicates effectively by treating the positive instances in the training set as an extensional definition of the correct predicate, and using this extensional definition to evaluate recursive calls while the predicate is being revised. In order to use the same technique on lower-level recursive predicates, FORTE derives temporary extensional definitions from proofs (or attempted proofs) of the positive instances in the training set (see Section 4.2.2).

This approach is not foolproof, but is often effective. The method fails in three circumstances. The first occurs when the top-level predicates are so incorrect that they do not provide a meaningful set of calls to the lower-level predicate. Second, the calls to the lower level predicate may not be ground. In this case, the derived extensional definition will be overly general, and FORTE is likely to develop an unintended definition for the lower-level predicate. Third, the lower-level predicate may be called with a restricted set of arguments. In this case, FORTE may again learn an unintended definition for the predicate.

An unintended definition often results in a correct program, but the lower level predicate does not have the expected meaning. For example, a predicate for naive reverse always calls `append` with lists of length one in the second argument. Suppose we begin with the incorrect program:

```
reverse([], []).
reverse([A|B], C) :- append(D, [A], C), reverse(B, D).

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

FORTE successfully revises this program to be a correct implementation of reverse:

```
reverse([], []).
reverse([A|B], C) :- append(D, [A], C), reverse(B, D).

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

The definition of `append` is correct for its role in this program. However, it is not a general-purpose append predicate, as the first clause is only correct when the second argument is a list of exactly one element.

In order to demonstrate the potential of FORTE's techniques, we presented it with incorrect versions of a realistic logic program. The program we used is a variation of Bratko's (1991) decision-tree induction program.

As a concession to efficiency, we placed most of the program's lower-level predicates in the fundamental domain theory. The portion of the program FORTE was asked to revise consisted of a top-level non-recursive predicate (which served as an interface to the program proper), a second level recursive predicate containing two base cases and three recursive clauses (which actually builds the decision trees), and a third-level recursive predicate containing one base case and two recursive clauses (which chooses the correct attribute to split on at a given level in the decision tree). This was a total of 31 lines of code.

The task given to the decision tree program was to construct a decision tree to correctly classify twelve blocks as positive or negative based on attributes of color, shape, and size. An instance to FORTE included the attributes and instances given to the decision-tree program along with the decision tree expected as output. FORTE received 14 positive instances, corresponding to the full decision tree constructed from the twelve instances, and all subtrees of the full tree (including leaves). FORTE also received 12 negative instances, which were trees or subtrees which might be constructed if the decision-tree program were to select the wrong attribute to split on at some point.

FORTE was able to repair most single errors introduced into the program, even in lower-level recursive predicates. The limits of our ability to repair the program were reached when we simultaneously introduced errors in two nested, recursive program predicates. In this case, Forte was able to correctly revise the program only if the error in the outer predicate did not prevent the derivation of a correct training set for the inner predicate. In practice, this meant that we placed the outer error in a base-case. In two such tests, the revision time averaged 3850 seconds. This relatively long revision time is due to the high arity of the predicates and the large number of variables present in several program clauses.

7. Application: Qualitative Modelling

To demonstrate FORTE's ability to work in diverse domains, we have also applied it to qualitative modelling. When supplied with appropriate domain knowledge, through the fundamental domain theory and the revision verifier, FORTE is able to synthesize and revise qualitative models suitable for use by QSIM (Kuipers, 1986).

7.1. Background

Qualitative modelling uses constraint-based models to predict and explain the behavior of dynamic systems in intuitive terms. For example, when trying to understand the effect of heating a pot of water, it may be more useful to simply know that the pot may boil over rather than to understand the numerical thermodynamic equations. Qualitative models can be given to simulators like QSIM (Kuipers, 1986) to determine all possible qualitative behaviors of the system.

Traditionally, qualitative models have been constructed by hand. This works for simple, well-understood systems. For complex systems, the approach of compositional modelling (Falkenhainer & Forbus, 1991) allows a system model to be built up from predefined components. Although this makes constructing models easier, it still requires the user to understand the system being modelled. Often, however, users want a model precisely because the target system is not well-understood.

An alternative approach is to induce a qualitative model directly from observations of a system's behavior. Coiera (1989) presents a method which, given a qualitative description of one or more system behaviors, derives a qualitative model that reproduces those behaviors. MISQ, a system independently developed by Richards, Kraan, and Kuipers (1992), uses some of the same techniques, but can synthesize qualitative models from qualitative or quantitative behavioral data. MISQ learns maximally constrained models and can handle incomplete behavioral descriptions.

FORTE uses components of MISQ to provide the domain knowledge it needs to work in the domain of qualitative modelling. However, FORTE substantially extends MISQ's capabilities by allowing the introduction of new system variables. FORTE can also be used to revise an imperfect qualitative model supplied by the user.

A qualitative model is represented as a single, conjunctive clause. Furthermore, one generally wants tightly constrained models that produce only the desired behaviors, so the language bias is most-specific. This means that FORTE will produce

Table 3. Summary of qualitative model induction results.

Model	Training Set	Number of Constraints	Execution Time
thrown ball	1 behavior	2 constraints	4 seconds
simple bathtub	1 behavior	3 constraints	2 seconds
two independent bathtubs	2 behaviors	6 constraints	35 seconds
cascaded tanks	2 behaviors	7 constraints	43 seconds
reaction control system	1 behavior	55 constraints	114 seconds

a single clause containing all constraints consistent with the input behaviors. As discussed by Richards, Kraan, and Kuipers (1992), when given complete behavioral information, a model generated in this manner is guaranteed to be unique, complete, and correct.

One instance specifies a complete system behavior over time; for each system variable we have a list specifying the variable's qualitative value, sign, and direction-of-change at a series of points in time. This information is interpreted by the QSIM constraint definitions provided in the fundamental domain theory. For example, in order to prove the constraint `m_plus(Amount, Outflow)`, FORTE provides the information on the variables `Amount` and `Outflow` to the fundamental domain theory predicate `m_plus`. If a monotonically increasing function holds between the two behavior terms, `m_plus` succeeds; otherwise it fails. A more complete description of QSIM constraints is beyond the scope of this paper, and is discussed by Kuipers (1986, 1989). FORTE's implementation of the QSIM constraints is taken from MISQ (Richards, Kraan, & Kuipers, 1992), and includes: `constant`, `M+`, `M−`, `add`, `multiply`, and `derivative`. From FORTE's point of view, the constraints in the fundamental domain theory are simply predicates that succeed or fail in the course of a proof.

Table 3 provides a summary of several models FORTE induced from behavioral data, ranging from the very simple model of a thrown ball to the much more complex Reaction Control System (RCS) on the space shuttle. As illustrations, we discuss the two cascaded tanks and the RCS below.

7.2. Two Cascaded Tanks

Cascading two tanks so that the drain from one provides the inflow to the next provides a moderately complex second order system. In order to provide a more difficult test, we omitted two system variables that a user might realistically forget: we supposed the user measured all the flows and amounts but did not realize that the calculated netflow for each tank would be important. Thus, we provided behaviors for the five other variables, but omitted the netflows entirely. Given two examples of system behavior, FORTE produces the model we would expect:

```
model(In_A, Out_A, Amt_A, Out_B, Amt_B) :-
     add(Out_A, Net_A, In_A),
```

```
derivative(Amt_A, Net_A),
m_plus(Amt_A, Out_A),
add(Out_B, Net_B, Out_A),
derivative(Amt_B, Net_B),
m_plus(Amt_B, Out_B).
```

The netflow variables are reintroduced by relational pathfinding, as a way of satisfying the requirement (enforced by the revision verifier) that the model be connected.

7.3. Reaction Control System

The Space Shuttle Reaction Control System (RCS) (Kay, 1992) is substantially more complex than the system of cascaded tanks, and provides a more realistic test of FORTE's capabilities in this domain. The RCS consists of a number of identical, parallel components; our test domain consisted of one of these components with its valves in fixed positions. Although space prevents us from giving a complete description of the RCS, a simplified view would contain three interconnected tanks, plus the thruster outlet. The first tank contains Helium, which is provided at constant pressure to the fuel tank. The Helium forces fuel out of the fuel tank and into the manifold. From the manifold, the fuel enters the thruster and ignites to provide thrust.

For the purposes of this section, we assume that the valve leading to the thruster is closed (i.e., the thruster is off), the Helium regulator valve is open and providing a constant-pressure supply of Helium, and the valve between the fuel tank and the manifold has just been opened. If the initial pressure in the manifold is lower than the initial pressure in the fuel tank (so that the system is not immediately at equilibrium), then fuel flows into the manifold until the pressures equalize. Providing this single behavior to FORTE allows FORTE to induce a correct system model for the RCS, with the addition of several correct but redundant constraints.

However, since FORTE is a theory refinement system, we can use it in a more sophisticated way. Suppose that the user has a correct system model, but that the system is behaving incorrectly. In this case, we can use theory refinement to revise the correct system model to reflect the actual system behavior. The resulting changes in the model can be viewed as a diagnosis.

One of the failures that can occur in the RCS is a leak in one of the manifolds leading from the fuel tank. In order to isolate the leak, the astronauts shut the valve leading from the fuel tank into the manifolds. They then isolate the suspected manifold and reopen the valve connecting the fuel tank and the manifolds. If the leak has been eliminated, the system will quickly reach equilibrium. If the leak has not been isolated, the system will not reach a pressure equilibrium (at least, not before all of the fuel has drained out through the leak).

If FORTE begins with a correct system model along with the system behavior caused by a leak in the manifold, FORTE revises the model by deleting the constraint `minus(D_Amt_Fuel, D_Amt_Man)`. The variable `D_Amt_Fuel` is the amount of fuel leaving the fuel tank and flowing into the manifold. Variable `D_Amt_Man` is the net change in the amount of fuel in the manifold. Normally, the amount of fuel flowing out of the fuel tank should be the same, except for sign, as the net amount of fuel being added to the manifold. Since FORTE deletes this constraint, there must be another influence on the amount of fuel in the manifold, namely, a leak.

8. Related Work

8.1. Propositional Theory Refinement

As previously mentioned, much existing work in theory refinement has dealt with propositional theories and does not handle relations or recursion. EITHER (Ourston & Mooney, in press) is FORTE's conceptual predecessor. It revises propositional theories using a combination of abduction and induction. Unlike FORTE, EITHER does not hill-climb and is guaranteed to fit an arbitrary theory to any set of noise-free data. However, EITHER's approach of computing all abductive proofs of unprovable positive examples was deemed computationally intractable for first-order theories.

RTLS (Ginsberg, 1990), KBANN (Towell & Shavlik, 1993), DUCTOR (Cain, 1991), and KRUST (Craw & Sleeman, 1991) are other theory refinement systems that are restricted to propositional theories. Of these, KRUST is most closely related to FORTE, since it follows the approach of generating many possible revisions to theories, and then choosing among them based on their performance. However, KRUST's approach suffers from a number of weaknesses. For example, even though KRUST requires multiple examples to be available, it generates revisions from only a single example. Also, revisions are evaluated based on rule-belief factors which ignore the difference between specialization and generalization errors; hence, a specialization error can be used to justify further specialization of a rule.

8.2. First-Order Learning

Most work in inductive logic programming (Muggleton, 1992) concerns generalizing an existing first-order Horn-clause theory by adding clauses, but does not address the problems of generalizing existing clauses or removing or specializing *incorrect* clauses. Shapiro's (1983) MIS system was capable of specializing theories; however, it required an oracle to answer membership queries for any predicate in the theory. His Prolog debugging system, PDS6, required even more interaction with the user. It required the user to judge the correctness of predicate calls, determine which clause in a predicate to change, and to actually write missing clauses. Other interactive systems include MARVIN (Sammut & Banerji, 1986) and CLINT (DeRaedt & Bruynooghe, 1992). CLINT generalizes and specializes theories and also creates new predicates via analogy. CLINT's revisions only include adding and removing clauses; it does not attempt to modify existing clauses. Unlike MIS and CLINT, FORTE automatically revises theories without any user interaction.

A number of recent knowledge-based learning systems use a first-order domain theory to bias the learning of an operational concept description but do not modify the actual domain theory. ML-SMART (Bergadano & Giordana, 1988) was the first system to take this approach. FOCL (Pazzani & Kibler, 1992) is a more recent approach based on FOIL. FOCL successively operationalizes a theory by either including portions of the domain theory or adding new literals via induction. The addition that provides the best information gain is chosen at each point. This general approach is at a disadvantage when the initial theory is missing lower-level rules (Cohen, 1992) since it must learn a completely new disjunct at the top level. Also, it can unnecessarily eliminate large portions of the theory that are consistent with the data but happen not to be needed to explain the training examples. By contrast, FORTE preserves as much of the initial theory as possible and can learn rules at any level in the theory. FOCL has been used as the basis for an interactive theory refinement system, KR-FOCL (Pazzani & Brunk, 1990). However, this system requires the user to determine where to make most theory changes.

GRENDEL (Cohen, 1992) is a recent system to use domain knowledge to guide induction. GRENDEL is a FOIL-like inductive learner that allows the user to provide an explicit bias in the form of a grammar. By providing different bias grammars, Cohen has shown that GRENDEL can simulate other systems such as FOCL. However, the user must encode the domain knowledge in the appropriate form and, as with FOCL, GRENDEL does not actually refine an initial theory. When a good initial theory is available, true theory refinement should have an advantage over systems like ML-SMART, FOCL, and GRENDEL. Also, theory refinement can improve the accuracy of subconcepts and related concepts for which no explicit training examples have been provided (Ourston & Mooney, 1991; Tangkitvanich & Shimura, 1992).

The GOLEM system has been applied to a number of learning problems. However, since it works by generalization, it may overgeneralize a theory. Bain and Muggleton (1992a, 1992b) present a method of specialization for GOLEM using "closed-world specialization" (i.e., negation as failure). GOLEM first executes normally, generalizing from examples to an induced theory. Any exceptions to this theory generated by a closed-world specialization algorithm are then recursively generalized. Using this technique in the KRK domain, GOLEM learns a correct theory after 10,000 examples. FORTE integrates specialization operators from the start, and it is therefore no surprise that its performance compares favorably with GOLEM's, learning a correct theory after 5000 examples. Wrobel (1993) presents an alternative definition of minimal semantic specialization, but does not address the issue of generalizing the resulting exceptions.

A few other automated refinement systems for first-order theories have also recently been developed. AUDREY (Wogulis, 1991) first specializes a theory by deleting clauses and then generalizes it using an abductive method that makes a single fault assumption. Consequently, its range of revisions is limited compared to FORTE. RX (Tangkitvanich & Shimura, 1992) first produces a revised operational definition and then translates this into changes to the original theory (as in the RTLS propositional system (Ginsberg, 1990)). The basic learning mechanism is very similar to FOCL, and therefore has some of the same problems discussed above. In addition, complete operationalization can result in an exponential expansion of the theory and can duplicate work when revising rules for a subconcept that appears in multiple places in the theory.

Aben and van Someren (1990) use a method of annotating and repairing incomplete or incorrect logic programs which is similar to FORTE's, in that they meta-interpret the execution of an example set and accumulate evidence which directs them to revise certain parts of the input program. However, they revise the program by means of syntactic transformations; for example, they might replace the constant "steev" in a failing rule with the similar constant "steve", found elsewhere in the program. This allows them to identify and correct many syntactic errors, such as typographical mistakes, which FORTE would have to treat as semantic errors (i.e., FORTE would not recognize the typographical similarity between "steev" and "steve"). However, their success depends on the initial theory being very nearly correct, since the corrections are expected to be found elsewhere in the program.

Finally, it should be noted that FORTE's relational pathfinding is similar to an early method for learning production rules developed by Langley (1980). Important differences are that Langley's method used unidirectional search and required the path to form the entire rule. Relational pathfinding uses spreading activation and can specialize paths by adding additional literals.

8.3. Belief Revision

Research in belief revision addresses the problem of finding a minimal retraction of beliefs required to consistently incorporate a new belief (Gardenfors, 1992). However this work does not address the inductive problem of generalizing a theory or specializing it by adding constraints (e.g., by adding additional antecedents to rules). Also, this work tends to focus on minimal semantic change which requires memorizing exceptions to the theory rather than producing a specialization that generalizes to new cases.

8.4. Qualitative Modelling

Bratko, Mozetic, and Lavrac (1989) did the some of the earliest work on learning qualitative models; however, it was not based on a general purpose simulation language like QSIM. Coiera (1989) presents GENMODEL, a method for inducing a QSIM qualitative model from qualitative behaviors. His approach is limited by the fact that behaviors must be completely specified, and his output models may contain incorrect constraints, due to the absence of dimensional analysis. A more powerful system, MISQ, was developed independently by Richards, Kraan, and Kuipers (1992). MISQ uses dimensional analysis, and is also able to work with incomplete behavioral information. The Misq model-building techniques are subsumed by Forte, and Forte's relational pathfinding allows correct models to be learned even when essential system variables have been omitted.

Golem has also been applied to the problem of learning qualitative models by Bratko, Muggleton, and Varsek (1991). However, their method requires hand-generated negative information (i.e., examples of behaviors that the system does not exhibit), it does not completely implement the Qsim constraints (e.g., corresponding values are ignored), and it does not use dimensional information. Golem also requires extensionally defined background knowledge, whereas Forte's fundamental domain theory allows background knowledge to be defined intensionally.

There has also been some recent work in constructing and revising models based on Forbus's (1984) qualitative process theory (Falkenhainer & Rajamoney, 1988). However this work uses analogy (Falkenhainer, 1990) and experimentation (Rajamoney, 1990) rather than induction from a fixed set of behaviors.

9. Future Work

Although Forte performs hill-climbing search, it considers a large number of operations at each step. Significant speedup could be obtained if a method could be developed for reducing the branching factor by only producing and testing the most promising revisions at each cycle. Forte also spends a great deal of time reproving many examples for each revision. A truth maintenance system that kept track of which examples would be affected by which changes could potentially eliminate much of this continual reproving.

Unlike some ILP systems (Muggleton & Feng, 1992; Quinlan, 1991), Forte does not exploit mode information, i.e., knowledge of which predicate arguments are input and which are output. The system could be enhanced to use available mode information to prune revisions and order literals in a clause.

Forte could also be enhanced to deal with negation as failure. Negation complicates revision since it switches the effect of generalizing and specializing operators. For example, learning a new rule for a predicate specializes rules in which the predicate appears negated. Bain and Muggleton (1992a, 1992b) present a general approach for inducing theories containing negation, and this approach could be adapted to theory revision.

The problems of modifying deeply recursive rules, discussed in Section 6.3, need to be addressed. Most current methods, such as Foil and Golem, also require complete extensional definitions of recursive predicates. However, a couple of recent papers address this issue (Muggleton, 1992b; Lapointe & Matwin, 1992; Cohen, 1993), and these ideas may lead to better techniques for revising recursive programs.

Another major problem is that Forte, like many ILP systems, cannot invent new predicates. The invention of new recursive predicates is a particularly difficult and important problem. Using general inverse resolution methods (Muggleton & Buntine, 1988) to invent new predicates without an oracle is computationally intractable. However, several efficient methods for inventing new predicates in restricted cases have recently been developed (Wirth & O'Rorke, 1991; Kijsirikul, Numao, & Shimura, 1992), and would be useful to add to FORTE.

In many domains, some form of uncertain or probabilistic reasoning is desirable; however, current theory refinement systems like FORTE are restricted to purely logical domain theories. RAPTURE (Mahoney & Mooney, 1993) is a recent system that combines connectionist and symbolic methods to refine propositional certainty-factor rule bases (Shortliffe & Buchanan, 1975). However, its basic approach should be applicable to first-order theories.

10. Conclusions

This paper has described and evaluated a completely automated approach to revising imperfect first-order Horn-clause domain theories by incorporating methods from propositional theory refinement and inductive logic programming. The ability to revise relational and recursive theories greatly increases the range of application of automated knowledge-base refinement. In particular, it allows for the automatic refinement of logic programs and qualitative models.

Our implemented system, FORTE, uses a hill-climbing algorithm with a diverse collection of generalization and specialization operators in an attempt to find a minimally revised theory that is consistent with a set of training examples. Its operators include simple propositional ones such as delete-rule and delete-antecedent, inverse resolution operators like absorption and identification, and a FOIL-like learner for adding antecedents and learning new rules. In addition, we introduce a powerful new operator, *relational pathfinding,* that helps overcome local maxima when learning relational concepts.

Experiments on standard relational benchmarks, such as the family domain, demonstrate FORTE's ability to effectively revise randomly corrupted domain theories and produce more accurate results than purely inductive learning. In the family domain, an ablation study reveals the particular effectiveness of relational pathfinding, which increases accuracy up to 20 percentage points.

Results in logic program debugging demonstrate that FORTE can correctly debug simple logic programs written by students for a programming languages course. The system was also able to correct small bugs in a decision-tree induction program. Finally, unlike previous Prolog debugging systems like Shapiro's PDS6, FORTE requires no user interaction.

In the domain of qualitative modelling, FORTE has been used to induce QSIM models of a number of simple systems from only a single positive qualitative behavior. It has also been used to induce, revise, and diagnose a fairly complex qualitative model of the Space-Shuttle Reaction Control System. The relational pathfinding operator is particularly important in automated qualitative modelling since it allows FORTE to introduce new system variables.

We believe that our results in these diverse domains demonstrate that relatively efficient automated refinement of complex relational theories is possible using existing methods in theory refinement and inductive logic programming. Continued research will hopefully improve the efficiency of these methods and incorporate advanced features such as predicate invention, negation as failure, uncertain reasoning, and better methods for revising deeply recursive programs.

Acknowledgements

Thanks to Ross Quinlan for making FOIL 5.1 available, to Steve Muggleton for making GOLEM 1.0 available, and to Josh Konvisser for helping us run these programs. Also, many thanks to the anonymous reviewers for their helpful comments on the initial draft of this paper. The first author was supported by the Air Force Institute of Technology. This research was also supported by the National Science Foundation under grant IRI-9102926, by the NASA Ames Research Center under grant NCC 2-629, and by the Texas Advanced Research Program under grant 003658114.

Notes

1. The Quintus Prolog implementation of FORTE, along with sample theories and test data, is available by anonymous FTP from cs.utexas.edu in the directory /pub/mooney/forte.
2. A definite program clause is a clause of the form $\alpha \leftarrow \beta_1, \ldots \beta_n$ where $\alpha, \beta_1, \ldots \beta_n$ are atomic formulae (Lloyd, 1987).
3. Mooney (in press) presents a formal definition of minimal change based on the notion of *syntactic distance* and shows that it guarantees convergence to a probably approximately correct (PAC) theory if the initial theory is guaranteed to be within a fixed distance of the true theory. Unfortunately, it appears computationally intractable to guarantee minimal syntactic change for any realistic theory language.
4. The *input clause* in resolution is the clause whose literal appears positively in the resolution step.
5. For readability, we display lists using functional notation. The actual representation used by FORTE uses explicit destructor predicates in place of function symbols.
6. The user can readily observe what the system does, but there are an infinite number of things that the system does *not* do, most of which do not provide useful information to the revision process.
7. The `insert-after` program may not be familiar. This program adds a new element into a list after the first occurrence of a specified marker element, e.g., `insert_after([a,m,b], m, n, [a,m,n,b])`.

References

Aben, M., & van Someren, M. (1990). Heuristic Refinement of Logic Programs. *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 7-12). London: Pitman.

Bain, M., & Muggleton, S. (1992a). Non-monotonic learning. In S. Muggleton (Ed.), *Inductive Logic Programming.* New York, NY: Academic Press.

Bain, M., & Muggleton, S. (1992b). Experiments in Non-monotonic First-Order Induction. In S. Muggleton (Ed.), *Inductive Logic Programming.* New York, NY: Academic Press.

Bergadano, F., & Giordana, A. (1988). A knowledge intensive approach to concept induction. *Proceedings of the Fifth International Workshop on Machine Learning* (pp. 305-317). San Mateo, CA: Morgan Kaufman.

Bratko, I. (1991). *Prolog programming for artificial intelligence*. Reading: MA, Addison Wesley.

Bratko, I., Mozetic, I., & Lavrac, N. (1989) *KARDIO: A study in deep and qualitative knowledge for expert systems*. Cambridge, MA: MIT Press.

Bratko, I., Muggleton, S., & Varsek, A. (1991). Learning qualitative models of dynamic systems. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 385-388). San Mateo, CA: Morgan Kaufman.

Cain, T. (1991). The DUCTOR: A theory revision system for propositional domains. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 485-489). San Mateo: CA: Morgan Kaufman.

Cohen, W. (1992). Compiling prior knowledge into an explicit bias. *Proceedings of the Ninth International Conference on Machine Learning* (pp. 102-110). San Mateo, CA: Morgan Kaufman.

Cohen, W. (1993). Pac-learning a restricted class of recursive logic programs. *Proceedings of the Eleventh National Conference on Artificial Intelligence* (pp. 86-92). San Mateo, CA: Morgan Kaufman.

Coiera, E. (1989). Generating qualitative models from example behaviors. Technical Report DCS 8901, Department of Computer Science, University of New South Wales.

Craw, S., & Sleeman, D. (1991). The flexibility of speculative refinement. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 28-32). San Mateo, CA: Morgan Kaufman.

DeRaedt, L., & Bruynooghe, M. (1992). Interactive concept learning and constructive induction by analogy. *Machine Learning*, 8, 107-150.

Falkenhainer, B. (1990). A unified approach to explanation and theory formation. In J. Shrager and P. Langley (Eds.), *Computational Models of Scientific Discovery and Theory Formation.* San Mateo, CA: Morgan Kaufman.

Falkenhainer, B., & Forbus, K. (1991). Compositional modelling: Finding the right model for the job. *Artificial Intelligence*, 51, 95-144.

Falkenhainer, B., & Rajamoney, S. (1988). The interdependencies of theory formation, revision, and experimentation. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 353-366). San Mateo, CA: Morgan Kaufman.

Gärdenfors, P. (1992) (Ed.). *Belief Revision*. Cambridge, England: Cambridge University Press.

Ginsberg, A. (1990). Theory reduction, theory revision, and retranslation. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 777-782). Cambridge, MA: MIT Press.

Ginsberg, A., & Weiss, S. and Politakis, P. (1988). Automatic knowledge-base refinement for classification systems. *Artificial Intelligence*, 35, 197-226.

Hinton, G.E. (1986). Learning distributed representations of concepts. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (pp. 1-12). Hillsdale, NJ: Erlbaum.

Kay, H. (1992). A qualitative model of the space shuttle reaction control system. Technical Report AI92-188, Artificial Intelligence Laboratory, University of Texas at Austin.

Kijsirikul, B., Numao, M., & Shimura, M. (1992). Discrimination-based constructive induction of logic programs. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 44-49). Cambridge, MA: MIT Press.

Kuipers, B. (1986). Qualitative simulation. *Artificial Intelligence*, 29, 289-338.

Kuipers, B. (1989). Qualitative reasoning: Modelling and simulation with incomplete knowledge *Automatica*, 25, 571-585.

Langley, P. (1980). Finding common paths as a learning mechanism. CIP Working Paper 419, Carnegie-Mellon University.

Lapointe, S., & Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. *Proceedings of the Ninth International Conference on Machine Learning* (pp. 273-281). San Mateo, CA: Morgan Kaufman.

Lloyd, J.W. (1987). *Foundations of logic programming*, second, extended edition. Berlin: Springer-Verlag.

Mahoney, J., & Mooney, R. (1993). Combining connectionist and symbolic learning to refine certainty-factor rule bases. *Connection Science,* 5, 339-364.

Mooney, R.J. (in press). A preliminary PAC analysis of theory revision. In T. Petsche, S. Judd, and S. Hanson (Eds.) *Computational Learning Theory and Natural Learning Systems, Vol. 3.* Cambridge, MA: MIT Press.

Muggleton, S. (1992a). Inductive logic programming. In S. Muggleton (Ed.), *Inductive Logic Programming* (pp. 3-27). New York, NY: Academic Press.

Muggleton, S. (1992b). Inverting implication. *Proceedings of the Second International Workshop on Inductive Logic Programming.* Tokyo.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 339-352). San Mateo, CA: Morgan Kaufman.

Muggleton, S & Feng, C. (1992). Efficient induction of logic programs. In S. Muggleton (Ed.) *Inductive Logic Programming* (pp. 281-298). New York, NY: Academic Press.

Ourston, D., & Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 815-820). Cambridge, MA: MIT press.

Ourston, D., & Mooney, R. (1991). Improving shared rules in multiple category domain theories. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 534-538). San Mateo, CA: Morgan Kaufman.

Ourston, D., & Mooney, R. (in press). Theory refinement combining analytical and empirical methods. *Artificial Intelligence.*

Pazzani, M., & Brunk, C. (1990). Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning. *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop.*

Pazzani, M., & Kibler, D. (1992). The utility of prior knowledge in inductive learning. *Machine Learning*, 9, 57-94.

Plotkin, G. D. (1971). Automatic Methods of Inductive Inference. Ph.D. Thesis. Edinburgh University, Edinburgh, Scotland.

Quillian, M.R. (1968). Semantic memory. In M. Minsky (Ed.) *Semantic Information Processing.* Cambridge, MA: MIT Press.

Quinlan, J.R. (1990). Learning logical definitions from relations. *Machine Learning*, 5, 239-266.

Quinlan, J.R. (1991). Determinate literals in inductive logic programming. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 442-446). San Mateo, CA: Morgan Kaufman.

Rajamoney, S. (1990). A computational approach to theory revision. In J. Shrager and P. Langley (Eds.), *Computational Models of Scientific Discovery and Theory Formation.* San Mateo, CA: Morgan Kaufman.

Richards, B. (1992). An operator-based approach to first-order theory revision. Ph.D. Thesis. Department of Computer Sciences, University of Texas, Austin, TX. Also appears as Technical Report AI 92-181, Artificial Intelligence Laboratory.

Richards, B., Kraan, I., & Kuipers, B. (1992). Automatic abduction of qualitative models. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 723-728). Cambridge, MA: MIT Press.

Richards, B., & Mooney, R. (1992). Learning relations by pathfinding. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 50-55). Cambridge, MA: MIT Press.

Sammut, C., & Banerji, R.B. (1986). Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.) *Machine Learning: An Artificial Intelligence Approach, Volume II.* San Mateo, CA: Morgan Kaufman.

Shapiro, E. (1983). *Algorithmic program debugging.* Cambridge, MA: MIT Press.

Shortliffe, E.H., & Buchanan, B.G. (1975). A model of inexact reasoning in medicine. *Mathematical Biosciences*, 23, 351-379.

Tangkitvanich, S., & Shimura, M. (1992). Refining a relational theory with multiple faults in the concept and subconcepts. *Proceeding of the Ninth International Conference on Machine Learning* (pp. 436-444). San Mateo, CA: Morgan Kaufman.

Towell, G., & Shavlik, J. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13, 71-101.

Wilkins, D. (1988). Knowledge base refinement using apprenticeship learning techniques. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 646-651). San Mateo, CA: Morgan Kaufman.

Wirth, R., & O'Rorke, P. (1991). Constraints on predicate invention. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 457-461). San Mateo, CA: Morgan Kaufman.

Wrobel, S. (1993). On the proper definition of minimality in specialization and theory revision. In Brazdil (Ed.) *Machine Learning - ECML-93* (pp. 65-81). Springer Lecture Notes on Artificial Intelligence Vol. 667.

Wogulis, J. (1991). Revising relational domain theories. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 462-466). San Mateo, CA: Morgan Kaufman.