An Approach to Repairing and Evaluating First-Order Theories Containing Multiple Concepts and Negation

James Lee Wogulis
wogulis@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, CA 92717

Technical Report 94-3

February 21, 1994

Dissertation submitted: December 17, 1993

UNIVERSITY OF CALIFORNIA

IRVINE

An Approach to Repairing and Evaluating First-Order Theories Containing Multiple Concepts and Negation

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

James Lee Wogulis

Dissertation Committee:

Professor Michael J. Pazzani, Chair

Professor Dennis F. Kibler

Professor Paul V. O'Rorke

1994

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my advisor, Michael Pazzani, for taking me on as a student late in the game, for letting me continue with the work I had been doing, and for giving me the encouragement and guidance to finish my dissertation. I would also like to thank Pat Langley, my first advisor, for getting me started in graduate school and for the years of support and guidance he provided. I also thank the other members of my doctoral committee, Dennis Kibler and Paul O'Rorke for their guidance.

Having been a graduate student for too many years, I have come to know a great many wonderful people who have helped and encouraged me along the way. In particular, I would like to thank my friends, Jim Kipps, Bernd Nordhausen, David Schulenburg, Dave Surmon, and Hadar Ziv for all the years of fun, travel and friendship. Special thanks go to David Schulenburg for his very helpful comments on an earlier draft of my dissertation. Almost too numerous to mention are all the other people I've known who have made my life as a graduate a little more tolerable and a lot more fun: Mats Heimdahl, Randy Jones, Karl Kilborn, David and Janice Levine, Owen O'Malley, John Self, Craig Snider, and Harry Yessayan.

I would also like to thank the following colleagues/friends for their insightful discussions and for making graduate school an enjoyable place to be: David Aha, Kamal Ali, John Allen, Cliff Brunk, Tim Cain, Piew Datta, John Gennari, Rogers Hall, Tim Hume, Wayne Iba, Chris Merz, Patrick Murphy, Don Rose, David Ruby, Stephanie Sage, and Kevin Thompson.

I would also like to thank Behavioral Technology Labs, Allen Munro, and Doug Towne for keeping me employed during most of my graduate career. Their patience, flexibility, and understanding of my needs as a graduate student have helped me out immensely.

Special thanks go to all the staff and support people here at ICS especially Caroline Ehrlich, Mary Day, Susan Moore, and Tim Morgan. Also thanks to Tim Morgan for his help with formatting this dissertation.

My biggest thanks goes to those people most important in my life: Kari Forester, my parents Carol Wenzlau, Stanley Wogulis, Russ Wenzlau and Joanna Wogulis, my brother Mark Wogulis and my entire family for all their love and support through these long and trying years as a graduate student.

Abstract of the Dissertation

An Approach to Repairing and Evaluating First-Order Theories Containing Multiple Concepts and Negation

by

James Lee Wogulis

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1994

Professor Michael J. Pazzani, Chair

This dissertation addresses the problem of theory revision in machine learning. The task requires the learner to minimally revise an initial incorrect theory such that the revised theory explains a given set of training data. A learning system, A3, is presented that solves this task.

The main contributions of this dissertation include the learning system A3 that can revise theories containing multiple concepts expressed as function-free first-order Horn clauses, an approach to repairing theories containing negation, and the introduction of a distance metric between theories to evaluate the degree of revision performed. Experimental evidence is presented that demonstrates A3's ability to solve the theory revision task.

Assumptions commonly made by other approaches to theory revision such as whether a theory needs to be generalized or specialized with respect to misclassified examples are shown to be incorrect for theories containing negation. A3 is able to repair theories containing negation and demonstrates a simple, general approach to identifying types of errors in a theory using a single mechanism for handling positive and negative examples as well as examples of multiple concepts.

The syntactic distance between two theories is proposed as an evaluation metric for theory revision systems. This distance is defined in terms of the minimum number of edit operations required to transform one theory into another. This allows for a precise measurement of how much a theory has been revised and allows for comparison of different systems' abilities to perform minimal revisions. This distance metric is also used by A3 in order to bias it towards finding minimal revisions that accurately explain the data.

The distance metric also leads to insights about the theory revision task. In particular, it is shown that the theory revision task is underconstrained if the additional goal of learning a particular correct theory is to be met. Without additional constraints, there are potentially many accurate revisions that are far apart syntactically. It is shown that providing examples of multiple concepts in the theory can provide some of these constraints.

Chapter 1
Theory Revision

1.1 The Problem

Imagine that you are given the job of maintaining an expert system written by someone else in your company. Now, assume that the system had been successfully used on a number of problems it was designed to solve when, one day, someone shows you a new problem it solves incorrectly. Because you know the system had worked in the past, you study the problem and are able to identify an error in the system. You correct the error, and the problem is solved. Contrast this with the strategy of discarding the system and rebuilding it from scratch in order to solve this new problem. All of the work, experience, understanding, and knowledge that your predecessor built into the system would be lost. Unfortunately, this is the approach taken by most machine learning systems.

Most machine learning methods are designed to acquire single concepts based on a set of training examples. A few systems are able to learn multiple concepts at one time but very little work has been done that allows a learner to *revise* a given set of concepts if they are found to be incorrect or incomplete. A class of learning systems that can build upon and revise its knowledge includes those that perform *theory revision.*

Such systems accept an initial, possibly incorrect, theory along with a set of training examples of concepts in the theory and output a revised version of the input theory that can explain the set of training examples. The output theory is required to be as similar as possible to the input theory and should differ only to the extent that errors in the input theory are corrected. A system that discarded the input theory and learned a new one from scratch would not be performing theory revision.

This dissertation presents a machine learning system that performs theory revision. The system identifies useful, irrelevant, and incorrect background knowledge and can modify and use this knowledge to improve its ability to classify a set of training examples correctly. The system's capabilities are evaluated on a number of domains and it is compared to other systems designed to address a similar task.

1.2 The Learning Task

One of the most commonly studied problems in machine learning is learning from examples (Quinlan, 1986; Michalski, 1983). In this approach, a learning system is presented with a set of examples of a concept and is expected to identify if a previously unseen example belongs to the concept. This is called *supervised* learning because the system is provided with examples and is told to which concepts they belong. This dissertation is concerned with the supervised learning task in which the learner is given an approximate concept description, called a theory, along with examples of concepts in the theory and is expected to revise the theory in order to classify correctly the set of training examples as well previously unseen examples.

A *theory* is viewed as a set of interrelated concepts that can be used for classifying examples. The theory revision task is similar to learning from examples. However, instead of building concept descriptions from scratch, the system revises the initial theory based on the examples provided. The types of revision may include discarding, modifying, and adding to portions of the initial theory.

Many learning systems are designed to take advantage of a user supplied, approximate concept definition (also called an incomplete or incorrect domain theory) to guide or constrain the learning process. Pazzani and Kibler (1992) state that the reason for providing a domain theory is "to increase the accuracy of learned rules." A variety of systems with different learning mechanisms share this objective, e.g., ML-SMART (Bergadano & Giordana, 1988), A-EBL (Cohen, 1992a), GRENDEL (Cohen, 1992b), IOE (Flann & Dietterich, 1989), IVSM (Hirsh, 1989), and EBL-ANN (Shavlik & Towell, 1989). These systems are typically evaluated by comparing the accuracy of the learned rules with and without an initial theory.

Some learning systems, called theory revision systems, impose an additional constraint on the learner. Mooney and Richards (1992) state that the goal of theory revision systems is "to minimally modify the theory to correctly classify all of the examples." The first theory revision systems such as EITHER (Ourston & Mooney, 1990), RTLS (Ginsberg, 1988), and KBANN (Towell, 1991) were restricted to revising theories in propositional logic. More recently, several systems such as KR-FOCL (Pazzani & Brunk, 1991), FORTE (Richards & Mooney, 1991; Richards, 1992), RX (Tangkitvanich, Numao & Shimura, 1992), and AUDREY (Wogulis, 1991) have been introduced that revise first-order theories expressed as Horn clauses.

1.2.1 Representation

In order to build a computational learning system, one must have a representation for examples and concepts. This dissertation is concerned with learning concepts

that can be represented using first-order Horn clauses (Lloyd, 1984; Plotkin, 1970; Vere, 1975; Quinlan, 1990). A concept in first-order logic defines a relationship between entities. The first-order formalism allows one to handle a large and interesting class of learning problems. For example, classes of concepts that require a first-order representation are family relationships, time relationships, spatial relationships (e.g., positions of pieces on a chess board), and physical relationships (e.g., chemical structures).

For example, the literal `uncle(X, Y)` can be used to define the uncle relation such that the literal is true whenever the person referred to by `X` is the uncle of the person referred to by `Y`.[1] The relation name `uncle` is used to refer to the concept for uncle.

An example of the `uncle` relation specifies values for the variables along with information about whether it is an example or non-example of the concept. If Walt is the uncle of Mark, `uncle(walt, mark)` is considered to be a *positive* example of the `uncle` relation. If Lee is not the uncle of Jim, `uncle(lee, jim)` is a *negative* example of the relation.

To be of any use, a concept must have a definition. Typically, this definition is expressed in terms of other known relations. For example, assume there exists a database of relationships between people that is only expressed in terms of `parent`, `spouse`, and `sibling` relations and all that is known about an individual is whether they are `male` or `female`. This database might include the literals `male(walt)`, `sibling(walt, carol)`, and `parent(carol, mark)`. Without a definition for the `uncle` relationship there is no way to determine from this database of relations whether one person is the uncle of another. However, the `uncle` relation can be expressed in terms of the those relations found in the database in a form known as a Horn clause:

```
uncle(U, N) :- male(U), sibling(U, S), parent(S, N).
uncle(U, N) :- male(U), married(U, M),
               sibling(M, S), parent(S, N).
```

The above definition for `uncle` has two parts, each represented by a clause. The first clause states that if person `U` is a male and there exists a person `S` who is a sibling of `U` and if `S` is also the parent of `N`, then `U` is the uncle of `N`. The second clause states that if person `U` is a male and there exists a person `M` who is married to `U` who is also a sibling of a parent of `N`, then `U` is the uncle of `N`. These two clauses define the two circumstances for which the `uncle` relation holds.

[1] Upper case symbols are used to designate variables and lower case symbols denote objects and relation names.

The first-order learning task then is to construct such a definition for the `uncle` relation given only those relations available in the database and positive and negative examples of the `uncle` relation.

1.2.2 Performance

At the highest level, the learner's task is to construct a concept definition that will correctly classify previously seen and unseen examples of the concept being learned. This ability is typically measured on a test set of positive and negative examples as the ratio of correctly classified examples to the total number of examples. Other important measures of a learning system's performance are its learning rate (number of examples required to learn a concept) and computational complexity. However, the theory revision task is to make *minimal modifications* to an existing incorrect theory in order to improve classification performance. It is therefore expected that the resulting theory bear as close a relationship in form to the original theory as possible.

1.3 An Approach to Theory Revision

This dissertation presents an approach to the theory revision task as described above. The following is a brief overview of the approach.

Examples of concepts to be learned are represented as ground literals, e.g., `uncle(frank, fred)`, that are labeled as either positive or negative examples of the concept to be learned. The initial theory given to the system is expressed as a set of Horn clauses. The system classifies an example by determining if it is a logical consequence of the theory. When an example is incorrectly classified there are two possible types of errors. If the example is a positive example of the concept being learned but is not a consequent of the theory, the system has made an "error of omission" and the theory is said to be "incomplete." If a negative example of the concept being learned is a consequent of the theory, the system has made an "error of comission" and is said to be "incorrect."

For example, the following theory for `uncle` is both incorrect and incomplete:

```
uncle(U, N) :- sibling(U, S), parent(S, N).
```

It is incorrect because it will classify a person `U` who is an aunt of `N` as an uncle because the theory is missing the condition that the uncle be a male. This theory is also incomplete because it does not express the relation that a person `U` can be the uncle of `N` if `U` is married to a sibling of a parent of `N`.

A theory is revised by locating and repairing errors until no more errors are found. A theory is found to be in error if it misclassifies any of the training examples. Each misclassified example will lead to the identification of several possible faults in the theory. The set of incorrectly classified examples is then viewed as a whole and the most frequently occurring fault is repaired first. A fault identifies a concept in the theory that is either incomplete or incorrect. Depending on the type of fault, the system applies a set of operators that attempt to repair the fault.

Returning to the above incorrect and incomplete theory for uncle, assume that the most common error among the examples is that aunts are classified as uncles. One possible fault in the theory is that the clause for uncle is too general because it classifies negative examples as being an uncle. One way to make the clause more specific is to add additional constraints to the definition of the clause. The system searches for additional constraints that allow positive examples of the uncle relationship to be classified and which prevents negative examples from being misclassified. In this case, the system would find the additional constraint that the person `U` must be a male and would revise the theory to be:

```
uncle(U, N) :- sibling(U, S), parent(S, N), male(U).
```

At this point, no more negative examples of `uncle` are misclassified. However, the theory is still in error because some positive examples of `uncle` are not correctly classified, i.e., situations in which the uncle is married to a sibling of a parent of person `N`. The system then looks for faults in the theory that would explain these errors. Finding that the concept for `uncle` is too specific, the system may then attempt to learn an additional clause for the `uncle` relation to cover those examples not covered by the existing clause. Assuming the system found a clause to cover the misclassified positive examples, it might produce the following correct theory:

```
uncle(U, N) :- sibling(U, S), parent(S, N), male(U).
uncle(U, N) :- male(U), married(U, M),
               sibling(M, S), parent(S, N).
```

Once the above theory had been learned, all of the positive and negative examples of the `uncle` concept would be correctly classified and the system would find no more faults in the theory.

1.4 Evaluation

As stated earlier, the theory revision task requires that the learner take an initial theory and examples as input and produce a minimally revised theory that correctly classifies the examples. Measuring the accuracy of a concept or theory is the most common metric used to evaluate machine learning systems. However, there are no

metrics currently being used to measure the degree of revision that these systems perform.

Along with a method for theory revision, this dissertation proposes a distance metric between theories that can be used to measure the degree of revision a system performs. The metric is defined in terms of the minimum number of edit operations required to transform one theory into another.

Computing the distance between the initial theory and the revised theory provides a measure of the degree of revision performed. Just as learning systems are compared to one another in terms of the accuracies of the theories they produce, this metric allows systems to be compared in terms of the amount of revision they perform on the initial theory.

If a correct theory for the problem is known ahead of time, computing the distance between the correct and revised theory provides a measure of how well the system can recover the correct theory from an initially incorrect one.

As an example of this distance metric, assume learners A and B are given the following theory and examples for the concept `uncle`:

```
uncle(U, N) :- sibling(U, S), parent(S, N).
```

Further, suppose that system A produced the following theory:

```
uncle(U, N) :- sibling(U, S), parent(S, N), male(U).
uncle(U, N) :- male(U), married(U, M),
               sibling(M, S), parent(S, N).
```

and that system B produced the theory:

```
uncle(U, N) :- sibling(U, S), parent(S, N),
               male(U), female(N).
uncle(U, N) :- sibling(U, S), parent(S, N),
               male(U), male(N).
uncle(U, N) :- male(U), married(U, M), sibling(M, S),
               parent(S, N), female(N).
uncle(U, N) :- male(U), married(U, M), sibling(M, S),
               parent(S, N), male(N).
```

The theories produced by both A and B are correct in the sense that they correctly classify all positive and negative examples of the `uncle` relation. However, when viewed in terms of distance, the theory produced by system A is syntactically much closer to the original theory than is the one produced by B and has therefore performed a more minimal revision of the initial theory. The distance metric is defined in Chapter 3 and quantifies this notion of degree of revision.

1.5 Contributions

The work in this dissertation provides several contributions to the field of machine learning research. As previously described, the problem being addressed is that of minimally revising an initial theory in order to explain a set of training data. A learning system, A3, is developed that performs this task.

A definition of the degree of revision is defined as the syntactic distance between two theories and is applied to evaluating theory revision systems. The degree of revision is defined in terms of the minimum number of edit operations required to transform one theory into another. This allows for a precise measurement of how much a theory has been revised and allows for comparison of different systems' abilities to perform minimal revisions. This distance metric may also be used by the learner in order to bias it towards finding minimal revisions that accurately explain the data.

This distance metric has also led to insights about the theory revision task. In particular, it is shown that the theory revision task is underconstrained if the additional goal of learning a particular correct theory is to be met. Without additional constraints, there exist many different accurate theories that are not syntactically close to one another. It is shown that providing examples of multiple concepts in the theory can provide some of these constraints.

The class of theories studied here are those expressible using function-free first-order Horn clauses. In addition, the clauses are allowed to be expressed using negations of other concepts in the theory. Previous work on theory revision has dealt with first-order theories but not with negation. Assumptions about whether the theory needs to be generalized or specialized with respect to misclassified examples are shown to be incorrect for theories containing negation. A3 correctly addresses these issues with negation and demonstrates a general approach to identifying types of errors in a theory by using a single mechanism to handle positive and negative examples as well as examples of different concepts.

1.6 Overview of the Dissertation

This dissertation describes the theory revision task, a solution to the problem in the form of a running system and an evaluation of the method. The following summarizes the contents of this dissertation.

- **Chapter 2: Background**

  Reviews work in the areas of first-order learning and theory revision. Some of this work forms the basis for the theory revision system proposed in this dissertation.

- **Chapter 3: Evaluating theory revision systems**

  Proposes a method for measuring the degree of revision performed by a theory revision system in terms of the syntactic distance between the revised theory and the initial theory. This distance is measured as the minimum number of literal-level edit operations required to transform one theory into another.

- **Chapter 4: System description**

  Describes the theory revision system A3 and discusses its various components and learning mechanisms in detail. The chapter concludes with a detailed example trace of the system in operation.

- **Chapter 5: Validation of the method**

  Contains experimental evaluation of the system's performance with respect to theory revision. In particular, the effect that different types of errors in an initial theory have on the predictive accuracy of the learned theories is measured. Also measured are the type and degree of modification the initial theories undergo. The chapter concludes with experiments demonstrating that presenting the system with examples of intermediate concepts in the theory allows it to learn theories that are much closer to the correct syntactic theory than if only examples of a single class were available.

- **Chapter 6: Comparison to other approaches**

  Describes other research addressing the theory revision task and other related issues. Comparisons between A3 and these other systems are made that includes discussion of the relative merits and drawbacks of the different approaches. Where possible, empirical studies are made comparing A3 to these related systems.

- **Chapter 7: Conclusions**

  Reviews the contributions of this dissertation and presents directions for future work.

Chapter 2
Background

This chapter reviews approaches to learning first-order concept descriptions and to revising first-order theories. Discussion begins with learning relations because this is the foundation and source upon which theory revision systems are built. Theory revision is a more elaborate version of the first-order learning task and involves identifying errors in the initial theory and then applying inductive learning methods to correct those errors. The two basic approaches to first-order learning are reviewed: searching for concept descriptions from general to specific and searching from specific to general. The chapter concludes with a review of different approaches to theory revision.

2.1 First-Order Concept Learning

As described in the first chapter, supervised concept learning systems accept examples of concepts to be learned and produce concept descriptions that can be used to classify unseen examples. One of the important characteristics of such systems is the representation language used to describe both examples and concepts. The work described here is concerned with first-order concepts because of their greater representation power. Concepts about temporal and spatial relationships require first-order representations as do logic programming concepts such as membership, list and sort (Plotkin, 1970; Winston, 1975; Vere, 1975; Quinlan, 1990; Clancey, 1992).

2.1.1 Definitions and Representation

Relations can be naturally represented using first-order logic. While not all relational learning systems explicitly use this representation (Michalski, 1983), it does provide a good framework for describing the relational learning task. The following are some useful definitions:

Definition: (concept)

A *concept* is a finite set of Horn clauses each of whose head uses the same predicate symbol.

Definition: (Horn clause)

A *Horn clause* is H :- $B_1, \ldots, B_n$. where H (called the *head*) and $B_1, \ldots, B_n$ (called the *body*) are literals and $n \geq 0$. When $n = 0$ a Horn clause is called a *fact.*

Definition: (literal)

A *literal* is $\mathtt{p}(t_1, \ldots, t_k)$ where p is a k-place *predicate symbol* and $t_1, \ldots, t_k$ are terms and $k \geq 0$.

Definition: (term, variable, function, constant)

A *term* is a *variable* or $f(t_1, \ldots, t_j)$ where f is a j-place *function symbol*, $t_1, \ldots, t_j$ are terms and $j \geq 0$. A 0-place function symbol is called a *constant.* Variables begin with an upper-case letter and functions with lower-case.

Definition: (ground literal, ground term, ground clause)

A literal, term, or clause that contains no variables is considered *ground.*

Definition: (example)

An *example* e of concept C is a ground literal that is labeled as either *positive* or *negative.*

Definition: (class membership)

An example e is a *member* of class C with respect to background knowledge K iff $(C \cup K)$ implies e.

A description for the concept "parent" might consist of the two clauses:

```
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
```

An example of this concept might be `parent(lee, jim)`. Note that the literal `brother(mark, jim)` is neither a positive nor negative example of the concept because they do not share the same predicate symbol.

Classifying an example e is then a matter of determining if the example is logically implied by the concept C, i.e., $C \vdash e$. In practice however there is typically another set of background clauses needed to support the concept definition. For

example, in the parent concept defined above, the two clauses alone are not sufficient because the "father" and "mother" relationship between the terms in the examples are also needed.

2.1.2 First-Order Learning Task

The first-order learning task is described as follows.

Given: a set of positive examples P and a set of negative examples N and background knowledge K (a set of Horn clauses)

Find: a concept C such that $\forall p \in P : (C \cup K) \vdash p$ and $\forall n \in N : (C \cup K) \nvdash n$.

There are two main approaches in relational concept learning for finding a definition of the concept being learned. The search for the concept definition can be either from general to specific or from specific to general. The rest of this section describes these two approaches to learning first-order concepts.

2.1.3 General to Specific Search

FOIL (Quinlan, 1990) is a relational learning system that searches from general to specific to find each clause comprising the target concept. FOIL learns function-free Horn clause concepts using a separate-and-conquer approach similar to Aq (Michalski, 1980), CN2 (Clark & Niblett, 1989), and GREEDY (Pagallo & Haussler, 1990). The search for concept descriptions is guided by an information-based heuristic.

Input to FOIL consists of ground literals representing the positive and negative examples of the concept being learned. Background knowledge for the task is also represented as ground literals. FOIL then learns a concept description in terms of the predicates in the example and background literals.

FOIL finds clauses one at a time that cover some portion of the input examples until of all the positive examples have been covered (Table 2.1). Each clause is required to cover some portion of the positive examples and none of the negative examples. FOIL's main loop performs a search for concept descriptions from specific to general because adding clauses to the concept being formed allows new positive examples to be classified while preventing any negatives from being incorrectly classified. However, the process by which each clause is formed is a general to specific search.

Clauses are formed in FOIL's inner loop by repeatedly specializing an initial clause to cover as many positive examples as possible until no negative examples

Table 2.1. FOIL's main loop.

```
FOIL(Pos, Neg)
  let Concept = ∅
  until Pos = ∅ do
    begin
      let C = FindClause(Pos, Neg)
      let Concept = Concept ∪ {C}
      let Pos = p ∈ Pos not covered by C
    end
  return Concept
```

Table 2.2. FOIL's clause finding algorithm.

```
FindClause(Pos, Neg)
  let Clause = p(X1, ..., Xk) :- true
  let i = 1
  while Neg ≠ ∅ do
    begin
      let Li = BestLiteral(Clause, Pos, Neg)
      let Clause = p(X1, ..., Xk) :- L1, ..., Li
      let Neg = extensions of Neg covered by Clause
      let Pos = extensions of Pos covered by Clause
      let i = i + 1
    end
  return Clause
```

Table 2.3. FOIL's best literal finding algorithm.

```
BestLiteral(Clause, Pos, Neg)
  let T_i^+ = |Pos|
  let T_i^- = |Neg|
  for each L in LiteralSpace(Clause) do
    begin
      let Pos' = extensions of Pos covered by L
      let Neg' = extensions of Neg covered by L
      let T_{i+1}^+ = |Pos'|
      let T_{i+1}^- = |Neg'|
      let T_i^{++} = number of Pos with extensions in Pos'
      let Gain(L) = T_i^{++} × (log2(T_{i+1}^+/(T_{i+1}^+ + T_{i+1}^-)) − log2(T_i^+/(T_i^+ + T_i^-)))
    end
  return L with highest Gain(L)
```

are covered by the clause (Table 2.2). The initial clause is the most general one that covers all positive examples, namely `p(`$X_1, \ldots, X_n$`) :- true` where each X_i is a distinct variable and `p` is the name of the concept being learned. This initial clause is too general because it is guaranteed to cover all the positive and negative examples. FOIL specializes the initial clause by adding literals to its body until no negative examples are covered by the clause. The selection of which literal to add is guided by an information-based heuristic.

When building a single clause, FOIL decides which literal to add to the body of the clause based upon how well the new clause partitions the positive and negative examples into two distinct classes (Table 2.3). An ideal literal would be one that causes the clause to cover all of the positive examples and none of the negatives. In practice though, a single literal will rarely have this property. Rather, each added literal will continue to cover some of the positive and negative examples. The heuristic for choosing the best literal trades off covering as many positives and covering as few negatives as possible.

As each clause is being built, FOIL maintains a list of positive and negative examples that the clause under construction covers. Each example is represented as a tuple that encodes the original example along with the bindings for any variables introduced by new literals in the clause. For example, the positive example `uncle(walt, jim)` would be represented as the tuple ⟨`walt`, `jim`, `carol`⟩ with respect to the clause `uncle(X, Y) :- brother(X, Z)` if the literal `brother(walt, carol)` were true.

To see how FOIL maintains these tuples, suppose the goal is to learn the concept "uncle" with background knowledge about immediate family relationships (e.g., brother and parent). The initial clause would be:

```
uncle(X1, X2) :- true.
```

which covers all of the positive and negative examples because there are no restrictions on the variable bindings. Adding a new literal to the clause results in:

```
uncle(X1, X2) :- brother(X1, X3).
```

which introduces a new variable `X3` to the clause. The only examples (both positive and negative) that will be covered by this clause are those where the person represented by the first variable (`X1`) are found to have a brother. FOIL extends the tuples representing the positive and negative examples for each possible variable binding of `X3` that makes the clause true. For example, the tuple ⟨`walt`, `jim`⟩ would be extended to ⟨`walt`, `jim`, `carol`⟩ and ⟨`walt`, `jim`, `lee`⟩ if the background knowledge specified that Walt was the brother of both Carol and Lee.

Maintaining the extended tuples with respect to the clause under construction allows FOIL to cache important partial proof information. Evaluating the addition of a new literal then only requires comparing the extended tuples to the new literal rather than recomputing from scratch whether the new clause covers the original examples. FOIL also uses information about the size of the extended tuple sets in its heuristic for choosing the best literal.

The space of literals that FOIL considers adding to a clause is computed in the `LiteralSpace` function mentioned in Table 2.3. The arguments to this function are the clause under construction and the tuples specifying the background knowledge. FOIL generates literals in the four forms $\mathtt{p}(V_1, \ldots, V_r)$, $\mathtt{not}(\mathtt{p}(V_1, \ldots, V_r))$, $X_i = X_j$, and $X_i \neq X_j$, where X_i and X_j are variables already existing in the clause, `p` is a predicate symbol found in the background tuples (this includes the concept being learned so recursive definitions are possible), and each V_i is either a new or existing variable.

FOIL prunes literals from this space when the syntactic form of the literal guarantees that it will not be useful (e.g., have $Gain = 0$). Literals with all new variables are discarded because they provide no discrimination and will have $Gain = 0$. Certain literals forming recursive definitions are discarded if they lead to infinite recursion. Finally, a partial order over literals with the same predicate symbol is defined by the occurrence of new variables. Computing the gain for literals in this partial order may lead to pruning when the *Gain* function guarantees no further improvement is possible.

FOIL uses an information-based heuristic to select the best literal as follows. Suppose FOIL has built the clause `p(X1, ..., Xk) :-` $L_1, \ldots, L_{i-1}$ and is trying to evaluate the usefulness of adding the new literal L_i. Before L_i is added there are T_i^+ positive and T_i^- negative tuples covered by the clause. The amount of information required to determine if a single tuple is either positive or negative is given by:

$$I(T_i) = -\log_2(T_i^+/(T_i^+ + T_i^-)).$$

Adding L_i to the clause gives rise to T_{i+1}^+ and T_{i+1}^- which denote the number of positive and negative tuples covered by the new clause. Similarly, the information now required to determine if a single tuple is either positive or negative is given by:

$$I(T_{i+1}) = -\log_2(T_{i+1}^+/(T_{i+1}^+ + T_{i+1}^-)).$$

The amount of information *gained* for classifying a single tuple by adding literal L_i is $I(T_i) - I(T_{i+1})$. If T_i^{++} is defined as the number of positive T_i tuples that are covered by the new clause, the total information gained by adding literal L_i is given by:

$$Gain(L_i) = T_i^{++} \times (I(T_i) - I(T_{i+1})).$$

One of the major problems with using this definition of *Gain* as the evaluation function is that there are cases where a correct literal to add has no information gain and will not be selected. This is often referred to as the "plateau problem" with hill-climbing search (Nilsson, 1980). For example, suppose FOIL were trying to learn the following definition for greater-than using only the successor function:

```
greater(X,Y) :- succ(X,Y).
greater(X,Y) :- succ(X,Z), greater(Z,Y).
```

Suppose that the first clause has already been learned and FOIL is starting to learn the second one. The positive tuples will include all of the greater-than relationships where the numbers are more than one apart (e.g., `greater(3, 0)`). When FOIL attempts to add the literal `succ(X, Z)`, it finds that $T_i^+ = T_{i+1}^+$ and $T_i^- = T_{i+1}^-$ because in every positive and negative tuple, the first argument is a number and every number has exactly one successor. This means that $Gain($`succ(X, Z)`$) = 0$ and it will not be chosen. The *Gain* will also be zero if FOIL tries to add the `greater(Z, Y)` literal. This problem can arise in general when the concept definition requires a literal that introduces a new variable. In this case, *Gain* is not an appropriate heuristic for selecting which literal to add. Solutions to this problem include changing the evaluation function (Kijsirikul, Numao & Shimura, 1991) and modifying the basic algorithm to add literals introducing new variables when no literal has sufficient gain (Quinlan, 1991; Richards & Mooney, 1992).

2.1.4 Specific to General Search

An alternative to FOIL's approach for finding relational concept descriptions is to search the space of clauses from specific to general. A number of relational learning systems use this approach (Vere, 1975; Muggleton & Buntine, 1988; Muggleton & Feng, 1990). A common feature of these methods is their use of least-general-generalizations (lgg). The idea being that one should not form concept descriptions any more general than is needed to explain the data. Plotkin (1970, 1971) defines the notion of an lgg for clauses and gives an algorithm for computing it.

Generalization for literals must be defined before discussing generalization of clauses. Literal L_1 is said to be more general than L_2 if $L_1\sigma = L_2$ for some substitution σ. For example, the literal `human(X)` is more general than `human(joe)`. The generalization of two literals L_1 and L_2 is also a literal L_g where $L_g\sigma_1 = L_1$ and $L_g\sigma_2 = L_2$. A least generalization of two literals L_1 and L_2 is a literal L_g such that all other generalizations of L_1 and L_2 are more general than L_g. For example, the literal `p(X, Y)` is a generalization of the literals `p(a, a)` and `p(b,b)` but is not the least general because `p(X, X)` is also a generalization but is more specific. There is always a unique lgg of two literals.

Literals themselves are insufficient for representing the class of relational concepts. Relational learning systems that search for Horn clauses from specific to general typically form generalizations between Horn clauses. Plotkin defines the lgg for clauses as follows. A clause C is represented as a set of literals $C = \{L_1, \ldots, L_n\}$ which is interpreted as the disjunction of literals L_j. A Horn clause is a clause with only one non-negated literal. Clause C_1 is defined to be more general than clause C_2 if there exists a substitution σ such that $C_1\sigma \subseteq C_2$ in which case C_1 is said to *subsume* C_2.

This definition of generality of clauses is different from the previous definition of generality for concepts which was based on implication. For example, consider the following two concepts:

$C_1 =$ { `c(X) :- real(X).` }
$C_2 =$ { `c(X) :- rational(X).` }

Concept C_1 describes the set of real numbers and C_2 the set of rational numbers. Under this interpretation concept C_1 is more general than C_2 because the set of rational numbers is a subset of the reals. However, under Plotkin's definition of generality, it is not true that C_1 is more general than C_2 because there does not exist a substitution σ such that $C_1\sigma \subseteq C_2$ where $C_1 =$ { `c(X)` $\vee$ `not(real(X))` } and $C_2 =$ { `c(X)` $\vee$ `not(rational(X))` }. Plotkin argues for using subsumption rather than implication to define generality because subsumption is more manageable. It is not known whether implication between clauses is even decidable (Plotkin, 1970; Buntine, 1988).

Table 2.4. Algorithm for computing lgg of two clauses.

```
lgg(C1, C2)
  /* first form lgg between all pairs of literals */
  let C = {K | L1 ∈ C1, L2 ∈ C2 and K = lgg(L1, L2)}
  /* simplify the generalization C */
  let E = C
  while E can be simplified do
    begin
      find L in C and substitution σ such that Eσ ⊆ E \ {L}
      let E = Eσ
    end
  return E
```

Plotkin's algorithm for computing the lgg of two clauses is given in Table 2.4. Both C_1 and C_2 are assumed to be of finite length. The algorithm operates in two phases. First, all pairs of literals from each clause are generalized to form a raw generalization C. Two literals can be generalized if they have the same predicate symbol and sign. So, `p(X, f(Y))` and `p(a, b)` can be generalized but `not(q(Z, a(b)))` and `q(W, Y)` cannot. The raw generalization C can have as many as $\mid C_1 \mid \times \mid C_2 \mid$ literals if both C_1 and C_2 contain only literals with the same predicate. The second phase of the algorithm simplifies the raw generalization down to the final lgg by removing all literals that are redundant under subsumption.

Systems for learning from examples that use Plotkin's method for computing generalizations typically work as follows (Vere, 1975; Muggleton & Feng, 1990). The initial clauses representing the concept are constructed from the examples themselves. If only a single positive example e_1 had been seen, the concept would have the form e_1 :- $L_1, \ldots, L_n$. The body of this clause would be comprised of background information that might be relevant to classifying the example. In the extreme case, these literals may include all the ground literals implied by the background knowledge.

The clauses forming a concept describe the conditions under which an example may be classified as a member of the concept. Without any *a priori* knowledge of what background information is relevant or needed to determine the class of an example the initial clause takes on the most specific form: e_1 :- *everything-we-know*. In other words, the initial hypothesis explaining e_1 states that e_1 is implied by the sum total of the background knowledge.

When a second example, e_2, is encountered, the system will add a second clause to the concept description: e_2 :- $L_1, \ldots L_n$. The bodies of these two clauses are identical and include all ground literals implied by the background knowledge. This concept description is not at all general because it can only classify the two examples e_1 and e_2. However, the least generalization of these two clauses provides a more

general clause that goes beyond covering just the two examples. Further, this generalization is the least general clause that explains the two examples and therefore is the least possible induction from the data. The guiding principle of this approach to concept learning is to make conservative generalizations that generalize beyond the data as little as possible. These generalizations are analogous to elements in the S set of version spaces (Mitchell, 1978).

There are a number of problems associated with computing the lgg of two clauses. Let M be the set of ground literals implied by the background knowledge, then the lgg of n examples could be as large as $\mid M \mid^n$. Without some means of weeding out irrelevant background knowledge, a learning system that relied on computing lggs would be greatly hampered rather than helped by providing more knowledge. Buntine (1988) presents a simple problem of learning list membership that produces a clause with 2,000 literals. Similarly, Muggleton and Feng (1990) show that the lgg of six examples of quick-sort having 49 background literals for append, 15 literals for partition and 16 examples of quick-sort will have $15^6+49^6+16^6+1 = 13,869,455,043$ literals. Plotkin (1970) also showed that a reduced lgg can be infinite in length.

One approach to solving the problem of intractably large lggs is the use of ij-determinate literals in GOLEM (Muggleton & Feng, 1990). This approach is a domain independent method for weeding out potentially irrelevant background knowledge. A determinate literal is one for which there is only ever a single binding for new variables not previously mentioned in the clause. The i and j parameters limit the number of other variables on which a new variable in the literal can depend. The basic idea is not to include literals that introduce new variables to the clause that are far away from variables in the head of the clause or that have variables that produce too many bindings.

Muggleton and Feng show that the size of the lgg of n examples is bounded by the parameters i and j and is independent of the number of examples n. However, the i and j parameters must be kept small because the bound is still large in the worst case. Muggleton and Feng show how their system can learn many list processing functions including a definition for quick-sort.

Quinlan (1991) shows how to add ij-determinate literals to FOIL which allows it to learn a definition for quick-sort it otherwise could not find. The effect of adding ij-determinate literals in FOIL is to move out of local minima where no single literal has positive gain by introducing literals with new variables that can provide gain to subsequent literals. It is interesting to note that FOIL uses determinate literals to expand its search space whereas GOLEM uses them to narrow it.

Another problem with basing a relational learning algorithm on lggs is learning clauses with negated literals. If the two clauses to be generalized contain no negated literals, Plotkin's lgg method will not introduce any such literals. To introduce negated literals, they need to be in the initial clauses formed from the background

knowledge. So the clause for a single example e_1 must contain negated literals. If the closed-world assumption is used, this extends the clause for e_1 to be e_1 :- $T_1 \wedge T_2 ... \wedge$ `not`$(F_1) \wedge$ `not`(F_2)... where the T_i are background literals known to be true and the F_j are literals that are not implied by the background knowledge. Clearly the number of ground literals that are *not* implied by the background knowledge will be intractably large. For example, if the background knowledge included the successor function, the initial clause would contain ground literals such as `succ(1, 2)`, `succ(3, 4)` as well as the literals `not(succ(1, 1))` and `not(succ(1, 3))`. This will cause a further increase in the size of lggs.

2.2 First-Order Theory Revision

A concept was earlier defined to be a set of Horn clauses that can be used to classify examples as either members or non-members of that concept. The bodies of these clauses are made from literals found in the background knowledge. By representing the background knowledge as a set of Horn clauses, the background knowledge forms a set of concepts. This leads to the following definition:

Definition: (theory)

A *theory* is a set of concepts.

Concepts within a theory are defined in terms of other concepts within the theory. For example, given a theory for family relationships, the concept `uncle` could be defined in terms of concepts for `parent`, `sibling`, and `male`.

Broadly stated, the theory revision task is to take as input an initial, possibly incorrect or incomplete theory and a set of examples of concepts in a theory and produce a new revised theory that covers the input examples. There have been three different basic approaches to the theory revision task, which are described in the rest of this section.

2.2.1 Theory-Guided Learning

Explanation-based learning (EBL) systems (Mitchell, Keller & Kedar-Cabelli, 1986) and some of the systems that combine inductive and explanation-based learning (Pazzani & Kibler, 1992; Tangkitvanich, Numao & Shimura, 1992) can be viewed as performing theory-guided learning. Such systems take advantage of the input theory when forming the final learned theory. Early EBL systems often assume the initial theory is correct but later extensions that also perform induction such as FOCL (Pazzani & Kibler, 1992) assume the theory is neither correct nor complete.

The goal of theory-guided learning is to produce a learned theory that can be used to classify unseen examples correctly. The initial input theory is only used to guide the learning process and is discarded once the final concept is produced. The form of the initial and learned theories is not assumed to be relevant and, in fact, the learned theories often look nothing like the original theories.

Mitchell et al.'s (Mitchell, Keller & Kedar-Cabelli, 1986) EBG system takes as input a domain theory and a single example and produces an operational concept description that generalizes the example based on the initial theory. The resulting learned concept is then expressed in terms of the lowest level predicates or facts in the theory and entirely eliminates any intermediate-level concepts. For example, the following is a theory for determining whether it is safe to stack one item on another:

```
safe_to_stack(X,Y) :- not(fragile(X)).
safe_to_stack(X,Y) :- lighter(X,Y).
lighter(X,Y) :-
      weight(X,W1),
      weight(Y,W2),
      less(W1,W2).
weight(X,W) :-
      volume(X,V),
      density(X,D),
      W is V * D.
weight(X,5) :- isa(X,endtable).
```

If the EBG system were presented with an example of an object that was safe to stack on an end table because its weight was less than five, it would learn the following concept:

```
safe_to_stack(X,Y) :-
      volume(X,V),
      density(X,D),
      W is V * D.
      less(W,5),
      isa(Y,endtable).
```

The learned concept is a generalization of the input example in so far as other examples that can be proved in the same way as this example will also be covered by the learned concept.

Although this technique and others based on it may have the property that the learned concept is correct or accurate, the theories produced have very little in common, at least syntactically, with the original input theory.

2.2.2 Theory Retranslation

Given the additional goal that the output theory should in some sense be close or related to the input theory, some learning systems attempt to perform the theory revision task by first translating the theory into some intermediate from, perform learning on the intermediate representation, and then re-translating the intermediate form back into the representation of the input theory. Examples of this approach are RTLS (Ginsberg, 1988; Ginsberg, 1990), KBANN (Towell, Shavlik & Noordewier, 1990; Towell, 1991), KR-FOCL (Pazzani & Brunk, 1991) and $\mathcal{RLA}$ (Tangkitvanich, Numao & Shimura, 1992).

RTLS and KBANN were designed to only handle propositional domain theories whereas KR-FOCL and $\mathcal{RLA}$ operate on first-order theories. RTLS, KR-FOCL and $\mathcal{RLA}$ all translate the initial theory into an operational form using methods from explanation-based learning. The operationalization process allows these systems to locate faults in the theory but often translating the repairs to the operational concepts back into corresponding repairs of the original theory is problematic.

KBANN takes a very different approach and translates the initial theory into a neural network representation with output units corresponding to the concepts being learned, input units corresponding to facts about the examples and hidden units corresponding to intermediate concepts. The weights on the connections are determined by the structure of the input theory. KBANN then trains the network on the example set and re-translates the resulting network back into propositional clauses based on the new weights of the connections.

While these methods may be useful in that well known learning algorithms can be applied to the intermediate representation, they may have difficulty in translating the intermediate form back into the original representation. Other difficulties arise when the expressiveness of the different representations do not match. For example, it is difficult to see how KBANN could be extended to first-order theories. Similarly, the systems that use operational concepts as their intermediate representation have difficulty with theories containing negation and recursion.

2.2.3 Theory Revision

Systems that perform theory revision have two goals: the learned theory should correctly classify the training examples and should retain as much of the form of the input theory as possible. Learning systems that make only minimally required changes to the initial theory have many desirable properties.

Presumably, the initial theory was created by a human expert in order to solve a particular task. The initial theory then is in a form comprehensible to the author

and others who may use it just as higher level programming languages make programs easier to write, understand, and maintain than if they were written using machine level instructions. Theories or programs often contain useful abstractions and concepts that should be maintained by the theory revision process.

Systems that perform theory revision include EITHER (Ourston & Mooney, 1990) which uses propositional domains, and AUDREY (Wogulis, 1991), FORTE (Richards & Mooney, 1991; Richards, 1992), AUDREY II (Wogulis & Pazzani, 1993), and A3 (Wogulis, 1993) which use first-order theories.

The first-order theory revision task is therefore defined to be:

Given: an initial theory of concepts expressed as first-order Horn clauses and positive and negative examples of concepts in the theory.

Find: a *minimally revised* version of the initial theory that is accurate on both seen and unseen examples of concepts in the theory.

Evaluating the accuracy of learned concepts has been a primary focus of research on concept learning systems. However, no work on theory revision to date has defined what constitutes a minimally revised theory nor how the degree of revision can be measured. The next chapter gives a precise definition of the distance between two theories and proposes its use as a metric for evaluating theory revision systems. Chapter 6 discusses theory revision systems such as FORTE and EITHER in more detail and compares these systems with A3.

2.3 Discussion

First-order theory revision extends the first-order learning task to repairing a set of interrelated concepts. Approaches to solving the first-order learning task fall into one of two different categories. First-order concept descriptions are typically built through either specialization or generalization. FOIL uses a hill-climbing algorithm to find literals to add to a clause that will cover positive examples and exclude negatives. This technique has some desirable properties in that it is computationally efficient, can easily add negated literals, and usually produces simple concept descriptions. Some drawbacks to this approach are that it can produce overly general concepts and that it may fail to learn certain concepts because of problems with hill-climbing.

The other approach to first-order learning is based on the notion of least-general generalization which performs the minimal amount of generalization between two positive examples. Some of the advantages to this approach are that it does not tend to overgeneralize, and it can find conjunctions of literals that hill-climbing methods can not find. Drawbacks to this approach are that it is computationally inefficient,

has problems introducing negated literals, and it may produce unnecessarily complex descriptions.

Subsequent chapters introduce the theory revision system A3 that builds upon the FOIL technique for learning first-order concept descriptions. The reasons for choosing this over the least generalization approach are its computational efficiency, that it can learn concepts containing negated literals, and that it learns small simple concept descriptions which may be more likely to be close to the initial theory.

To repair faulty theories, A3 takes the theory revision approach and makes repairs to the initial theory rather than to a retranslated form of the original theory. Such revisions are more likely to be minimal modifications than if the theory were translated into an intermediate form and then back again.

Chapter 3
Evaluating Theory Revision Systems

One of the goals of theory revision is to produce a minimal modification of the initial theory in order to classify the training examples. However, to date, no theory revision system has been evaluated with respect to the degree of revision performed nor has any system been designed explicitly to satisfy this goal.

This chapter proposes a distance metric between Horn clause theories that may be used to evaluate how well theory revision systems meet the goal of learning minimally revised theories. The measure is defined as the number of edit operations (i.e., additions, deletions, or replacements of literals) that are needed to transform one theory into another.

The notion of edit-distance has been studied by others (Tai, 1979; Shasha & Zhang, 1990) and applied to problems in areas such as pattern matching, parsing, and comparing secondary structures of RNA. Without such an evaluation, a system that ignores the initial theory and runs a purely inductive learner on the training data might also qualify as a theory revision system.

This chapter describes in detail why the minimal modification constraint is useful, and defines a measure of the distance between two theories. This measure is used to define the appropriateness of a revised theory with respect to the initial theory. The results reported in subsequent chapters use this metric to evaluate the performance of theory revision systems and the metric is also used in the theory revision system, A3, to choose among potential revisions to make.

3.1 Theory Revision

A theory revision system is given an initial domain theory that is incorrect and/or incomplete, and a set of positive and negative training examples and produces a revised theory. This chapter proposes that an appropriately revised theory is one

that is closest to the initial theory and one that makes the fewest errors on unseen examples drawn from the same distribution as the training examples.

Using distance as one factor to measure the quality of a revised theory favors theories that preserve as much of the initial theory as possible. There are several reasons why this is desirable:

- One can argue for minimal changes to a theory based on arguments of parsimony. Without any criteria other than accuracy on which a theory may be evaluated one should make only minimal modifications needed to repair a theory in order to explain new data.
- Such revisions are more likely to be understood by experts who presumably understand the initial theory, because they represent minor variants of the experts' theories. An expert may be unwilling to accept a drastic change to a theory, especially if adding or removing a few training examples results in yet another drastically different theory.
- The abstractions from the initial theory may be retained in the revised theory. Many learning systems (e.g., FOCL (Pazzani & Kibler, 1992) and A-EBL (Cohen, 1992a)) that use a domain theory to guide learning express the learned concept in terms of the operational predicates. This results in a "flat" theory, expressed solely in terms of "observables" without intermediate conclusions. In contrast, a theory revision system should use the abstract predicates from the initial theory, provided they are useful in making classifications.
- Abstractions from the initial theory that are retained in the revised theory can support the generation of meaningful natural language explanations of the system's translations (Swartout, 1981). For example, an explanation derived from clauses learned by FOCL to determine whether a student is required to pay back a loan might be expressed in English as *"John is not required to make a payment because John is enrolled at UCI for 15 units and 15 is greater than or equal to 12."* In contrast, the clauses learned by A3 would support an explanation that involves intermediate conclusions: *"John is not required to make a payment because John is eligible for a student deferment because John is a full time student at UCI."*

Preferring minimally revised theories based on the distance metric favors theories that make small incremental changes rather than major re-writes of the initial theory. While this seems desirable in general, there are situations, perhaps corresponding to paradigm shifts (Kuhn, 1962), in which an entire theory must be discarded and replaced by a drastically different theory. This issue is not considered in detail, but note that it might be addressed by introducing a dimension, such as elegance or simplicity, that may be combined with accuracy and distance to evaluate revised theories.

One can use the distance metric to compare different revisions of the same initial theory to determine which revision is closest to the initial theory. When used together with accuracy, distance provides a better overall measure of the quality of the revised theories. Clearly, a theory that performs better on both accuracy and distance is to be preferred. If theory A is more accurate than theory B but theory B is closer to the initial theory, the issue becomes murkier. Because these measures are orthogonal, preferring one theory over another depends on the user's trade-off between the two goals.

3.2 Theory Distance Metric

The goal, therefore, is to define a metric analogous to accuracy that can be used to measure the quality of revisions made by theory revision systems. Whereas the accuracy of a revised theory measures its semantic correctness, a measure of the syntactic closeness between the two theories is defined as the distance between the initial and revised theories. The distance between two theories is defined as the minimum number of revisions required to transform one theory into another. Naturally, this measure is commutative.

3.2.1 The Distance Between two Theories

As defined previously, a domain theory is a set of Horn clauses which is partitioned into concepts. A concept $K_{pred}(T)$ for theory T is defined as the set of clauses in the theory T with the same predicate symbol *pred* and arity for the head of the clause: $K_{pred}(T) = \{C \in T \mid C \equiv pred(\cdots) \leftarrow L_1, \cdots, L_n\}$. For example, the clauses `append([],[],[])` and `sort([],[])` belong to different concepts. The distance between two theories is defined to be the sum of the distances between corresponding concepts from each theory.

$$dist(T_1, T_2) = \sum_{pred \in T_1 \cup T_2} dist(K_{pred}(T_1), K_{pred}(T_2))$$

This definition of distance does not compare clauses from different concepts so there is no defined distance between the clauses for **append** and **sort** given above.

Because a concept is a set of clauses, there may be many ways in which one concept could be transformed into another. When measuring the distance between concept A and concept B, there are two main cases to consider. A clause C_A from A could be the source of many (including zero) clauses in B: $C1_B, ...Cn_B$, or similarly many clauses in A ($C1_A ... Cm_A$) could be the source of a single clause C_B in B. The

distance between two concepts is defined to be the minimum sum of clause distances for all mappings between clauses of each concept.

$$dist(A,B) = \min_{map \in mappings(A,B)} \left[\sum_{\langle C_A, C_B \rangle \in map} dist(C_A, C_B) \right]$$

Because there is no way to know which clauses in one theory should correspond to which clauses in another, the distance between two concepts is defined to be the minimum sum of distances for all possible mappings between clauses from each concept. Clauses may be mapped to the empty clause which corresponds to clause deletion.

The set of mappings between clause sets A and B can be computed as follows. First, form the Cartesian product of clauses from A and B. For example if $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$, one element of $A \times B$ would be $\langle a_1, b_2 \rangle$. Next, compute the power set of $A \times B$. A single mapping is formed from one element of this power set, e.g., $map = \{\langle a_1, b_1 \rangle, \langle a_2, b_1 \rangle\}$. For any $a_i \in A$ or $b_j \in B$ not appearing in map, add the elements $\langle a_i, \emptyset \rangle$ and $\langle \emptyset, b_j \rangle$ to map where $\emptyset$ represents the empty clause. Therefore, map would be transformed into $map = \{\langle a_1, b_1 \rangle, \langle a_2, b_1 \rangle, \langle \emptyset, b_2 \rangle\}$. This mapping is interpreted to mean B can be revised into A by transforming clause b_1 into clauses a_1 and a_2 and by deleting clause b_2. In general, the number of mappings between clause sets A and B is $2^{|A| \cdot |B|}$.

As an illustration of the mapping between clause sets, consider theories A and B each with the following clauses for the concept p:

A_1: `p :- t, a, m, e.`
A_2: `p :- x.`

B_1: `p :- t, i, m, e.`
B_2: `p :- t, o, m, e.`

The array in Table 3.1 presents the distances between the clauses from each theory including the empty clause. Finding the minimum mapping between clause sets is therefore a matter of finding a subset of array elements whose sum is minimum and such that the subset contains *at least* one element from each row and each column in the array. The underlined array elements in Table 3.1 form the minimum mapping giving a total distance of four. This mapping can be interpreted to mean that theory A can be transformed into theory B by deleting clause A_2 and transforming clause A_1 into clauses B_1 and B_2.

The problem of finding the subset of array elements with the minimum sum such that the subset contains at least one element from each row and each column is closely related to the *assignment problem* (Lawler, 1976) which has a polynomial

Table 3.1. Minimum mapping between clauses from two theories.

	`p :- t,i,m,e`	`p :- t,o,m,e`	∅
`p :- t,a,m,e`	**1**	**1**	5
`p :- x`	4	4	**2**
∅	5	5	**0**

time complexity solution. The assignment problem differs in that the solution must contain *exactly* one element from each row and each column. The complexity of the clause-mapping problem is unknown but there is reason to believe that it may be polynomial as well. As implemented in A3, the distance metric is exponential and somewhat costly to compute but not prohibitive on the domains described in this dissertation.

For present purposes, the definition of the edit distance between two clauses will be at the literal level: i.e., how many additions, deletions or replacements of literals are needed to transform one clause into another.[1] However, differences in the order of literals in a clause and renaming of logical variables are ignored. The distance between two clauses, $dist(C_1, C_2)$, is defined to be the minimum number of additions, deletions, or replacements of literals needed to transform clause C_1 into clause C_2 modulo variable names and literal ordering in the bodies of the clauses.[2] For example, the distance between the following two clauses is one:

C_1: `grandfather(X,Y) :- sister(X,Y), father(X,Z).`
C_2: `grandfather(A,B) :- father(A,C), parent(C,B).`

Without regard to the variable names or goal ordering, clause C_1 can be transformed into C_2 by replacing `sister(X,Y)` with `parent(Z,Y)`.

3.2.2 Relation Between Distance and Accuracy

It is important to note that the definitions of accuracy and distance are orthogonal to one another and thus complementary. That is to say that two theories may have the same accuracy, even make the same classification of every example, and yet

[1] Another possible definition could be more detailed and include the number of changes to terms within a literal. This complication is ignored because the theories used by theory revision systems so far do not include complex literals such as `pred(f(A),[A|B])`. Note that these might be handled within this framework by breaking a complex literal into several simpler ones (Norvig, 1992) `pred(V1,V2) & V1 = f(A) & V2 = [A|B]`.

[2] The *dist* function is known a *pseudo-metric* because $dist(x, y) = 0 \not\Rightarrow x = y$ due to the possibility that literals may be reordered and variables may be consistently renamed.

Table 3.2. Three theories for positive, even integers.

Theory 1:

```
poseven(X) :- greater(X,0), even(X).
```

Theory 2:

```
poseven(X) :- modulo(X,2,Y), equal(Y,0),
              not(greater(Y,X)).
```

Theory 3:

```
poseven(X) :- less(X,0), even(X).
```

there may be a large syntactic difference between the two theories. An example of this is the difference between theories 1 and 2 in Table 3.2, both of which represent a correct theory for positive, even integers.[3]

Similarly, two theories may be close to one another syntactically but have entirely different semantic meaning. An example of this is theories 1 and 3 in Table 3.2. Both theories are quite close syntactically but have very different meaning and hence accuracy.

3.2.3 Uses of the Distance Metric

The primary use of the distance metric will be to compare revisions of the same initial theory by theory revision systems to determine which revision is closest to the initial theory. This will allow one to evaluate the claim that theory revision system A produces better revisions than theory revision system B.

However, if the correct theory is known (e.g., if the incorrect theory were formed for evaluation purposes by introducing errors into a correct theory), there are two additional uses of the distance metric.

First, the distance between the initial theory and the correct one provides a measure of the syntactic corruptness of the initial theory. This should provide a more useful measure of the difficulty of a theory revision problem than just the accuracy of the initial theory. Presumably, the more corrupt a theory is, the harder it is to revise.

Second, the distance between the revised theory and the correct one can be used in conjunction with accuracy to measure how well a learning system is able to replicate theories created by humans. Currently, knowledge engineers perform the theory revision task manually. A prototype expert system is often built, and then

[3] *modulo*(A, B, M) is defined such that M is the remainder of A divided by B.

it is refined to perform better on a set of test cases (e.g., (Rabinowitz, Flamholtz, Wolin & Euchner, 1991)). The initial theory and test cases can serve as input to a theory revision system and the automatically derived theory can be compared to the theory produced manually. The distance between the automatically and manually revised theories measures the system's ability to find theories similar to ones built by experts.

3.2.4 Some Drawbacks

Theory revision systems require evaluation beyond simply measuring the accuracy of the theories produced. A measure of the type and quality of revisions made is also needed. Although the distance metric proposed here is useful toward that end, it is not a panacea. Because distance is a syntactic measure, it does not handle some theories one would like to consider equivalent. For example, the learner may have clauses containing literals with equivalent meaning such as `less(X,Y)` and `geq(Y,X)`. Comparing two theories at the syntactic level will not consider these two literals equivalent. This problem could be addressed by providing additional equality axioms to determine the equivalence between two literals. However, this is not a major problem with the theories described in this dissertation because the background knowledge tends to be limited to make the induction process more efficient.

3.3 Discussion

Theory revision systems have the goal of improving the accuracy of the initial theory as well as some constraints on the type and degree of revisions that are desirable. It is argued that revised theories should be as close to the original as possible. Unfortunately no theory revision systems to date have been evaluated in terms of the degree of revision made to the initial theory.

This chapter proposed a measure for the distance between two theories. This measure corresponds to the minimum number of edit operations at the literal level required to transform one theory into another. This can be used in conjunction with accuracy to give a better overall evaluation of the quality of revised theories when comparing one theory revision system to another. The distance metric can also be a useful tool when the correct theory is known. Comparisons of the revised and correct theories give a measure of a system's ability to learn theories in a form experts would write.

In addition to evaluating the performance of theory revision systems, the distance metric can also be used by such systems in order to evaluate the quality of potential revisions. The next chapter presents the theory revision system A3 that

uses both accuracy and the distance metric to guide its search through the space of revised theories. Incorporating the distance metric into the theory revision process can help bias the system toward making minimal revisions of the initial theory.

Chapter 4
A3: A First-Order Theory Revision System

Previous chapters have discussed the first-order theory revision task and described various approaches to solving the problem. One of the explicit goals of theory revision is to perform minimal modifications to the input theory in order to classify the training examples. However, no existing theory revision systems explicitly attempt to meet this goal. Also, none of these systems have been evaluated in their ability to achieve this goal.

This chapter presents A3, a first-order theory revision system that attempts to produce correct, minimally modified revisions of an initial theory. The class of theories the system can revise includes function-free first-order theories containing negated literals.

The basic approach is to locate and repair errors in the theory until all of the input examples are correctly classified. A uniform method for processing positive and negative examples of multiple concepts in the theory is developed that gives A3 a simple and elegant structure not found in other theory revision systems.

4.1 The Theory Revision Task

A3 solves the first-order theory revision task as described in Section 2.2.3. The systems accepts as input:

- An initial theory expressed as a set of function-free Horn clauses. Literals in the body of a clause may be negated where negation is defined as negation-by-failure using the closed-world assumption.
- Background knowledge identified as a subset of the initial theory. The learner may assume the background knowledge is correct and complete and requires no repairs.

Table 4.1. An incorrect domain theory for concepts odd and even.

```
odd(X) :- not(even(X)).
even(2).
```

- A set of positive and negative examples for concepts in the theory which may include examples of multiple concepts.

As output, A3 produces a revised theory that attempts to meet the following requirements:

- The revised theory should correctly classify the input examples.
- The revised theory should be a minimal revision of the initial input theory.
- Unseen examples of concepts in the theory should be correctly classified.

That the revised theory should correctly classify the input examples as well as unseen examples is the standard learning from examples task. Along with these requirements, the revised theory should be a minimal revision of the input theory. The degree of revision is defined in terms of the distance metric proposed in the previous chapter. An optimal solution would be a revised theory T_r that correctly classifies the input examples and $\forall T_j : dist(T_r, T_{input}) \leq dist(T_j, T_{input})$ that correctly classify the input examples.

Although A3 does not guarantee it will find an optimal solution, it does use the criteria of correctness and minimal revision to guide its search for a solution.

4.2 Negation in First-Order Theories

Before discussing the details of the A3 system, this section points out a problem with negation in first-order theories that other theory revision systems do not handle correctly.

Table 4.1 presents an incorrect theory for the concepts odd and even. The definition of odd is overly general because it is defined in terms of the negation of even which is overly specific.

If the input examples were for the concept odd, most theory revision systems would assume that the theory is overly general and requires either deleting some clauses or adding literals to existing clauses. However, as demonstrated by this example, the definition of odd can be considered correct and it is the concept even

Table 4.2. Top-level Algorithm for A3.

```
A3(theory, examples)
  {
    while theory accuracy over examples is increasing do
      assumption = find_best_assumption(theory, examples)
      theory = modify_theory(assumption, theory, examples)
    return theory
  }
```

which is overly specific. Thus to repair this theory, the concept for `even` must be generalized.

Even though the overall theory may be too general with respect to the input examples, this does not imply that some concept within the theory is overly general. Thus any theory revision system that uses the classification of an input example as the only means of deciding to apply generalization or specialization operators will be incorrect when confronted with a theory containing negated literals.

4.3 The A3 Theory Revision System

A3 is a system for revising first-order theories containing negation. The basic approach is first to identify the type of error and its location within the theory and then to correct the error based on this information.

Table 4.2 shows A3's top-level algorithm. The inputs are an initial, possibly incorrect theory and a set of positive and negative examples of concepts in the theory. The examples are not represented as two distinct sets but rather as a single set of literals that the theory should prove true. Thus the positive example `odd(3)` would be represented as `odd(3)` and the negative example `odd(4)` would be represented as `not(odd(4))`. An example literal is referred to as *positive* if it is non-negated and *negative* if the literal is negated. Note that A3 accepts examples of any concept in the theory so { `odd(3)`, `even(4)` } is a valid input example set. A3 returns a modified theory that should cover all of the input examples and which will presumably do well at covering unseen examples of concepts in the theory. The search is a form of hill-climbing that always attempts to improve the coverage of the input examples. The system stops when it cannot find a theory that increases the coverage of the input examples.

The search for a revised theory has two main phases. First, a set of candidate assumptions is produced that would help explain the uncovered examples. A single

Table 4.3: An incorrect theory to identify an element in the symmetric difference of two lists.

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).

member(X,Z) :- cons(Y,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

best assumption is chosen from this set. Next, the best assumption is used as a guide to revise the theory in such a way that the assumption must no longer be made to cover some portion of the uncovered examples. This process is repeated until all examples are correctly classified.

4.4 Locating Errors in a Theory

If a concept in a theory fails to cover an example, the concept is too specific when the example is positive and is too general when the example is negative. Because a theory is comprised of a set of interrelated concepts, it is possible that one concept is overly general because another one is too specific (and vice versa). For example, Table 4.3 presents an incorrect theory for the concept `diff` that finds an element in the symmetric difference of two lists. Thus `diff(1, [1,2,4], [2,3])` is a positive example because the item 1 is a member of the first list but not the second whereas `diff(2, [1,2,4], [2,3])` is a negative example because the item 2 is a member of both lists. This theory is taken from (Shapiro, 1983) page 74.[1] The concept `diff` is overly specific because the definition of `member` is too general and is used within a negation for the definition of `diff`.

Because it is possible to generalize a concept in a theory by specializing another concept (and vice versa), the type of revision to be made (generalization or specialization) depends on the type of error and where it occurs in the theory. A3 uses a mechanism for finding assumptions to identify a faulty concept and whether that concept is overly general or specific.

[1] This theory has been re-written to remove function symbols such as `[|]` but remains semantically equivalent. This can be done with a simple preprocessor (Richards, 1992; De Raedt, 1991).

Table 4.4. Algorithm for finding the best assumption.

```
find_best_assumption(theory, examples)
  {
    all_assumptions = ∅
    for each uncovered example E do
      all_assumptions = all_assumptions ∪ find_assumptions(theory, E)
    best = elements from all_assumptions that cover most examples
    best = set of deepest assumptions from best
    return random element from best
  }
```

An assumption is defined in A3 as a literal that is assumed to be true but which is not implied by the theory. For example, in the above theory for `diff`, the positive example `diff(4, [1,2,4], [2,3])` could be proved by making any of the assumptions:

```
not(cons(Y, Xs, [1,2,4]))
not(member(4, [1,2,4]))
not(member(4, [2,3]))
diff(4, [1,2,4], [2,3])
```

none of which are provable by the theory.

The goal of the assumption finding phase is to find the single best assumption defined as the assumption that can be used to prove the most number of incorrectly classified examples. Table 4.4 outlines the algorithm for finding the best assumption. A3 only finds a single assumption although it could be extended to find a conjunction of assumed literals. This would presumably allow A3 to repair a larger class of theories but experiments indicate that a single assumption is adequate.

It is important to note that this does not imply that A3 cannot repair theories with multiple errors. As long as the errors in a theory do not interact with one another, A3 should still be able to repair the errors by finding different assumptions that highlight those errors. By chosing a single best assumption as the basis for revising a theory, A3 is biased towards first repairing those errors responsible for the largest number of misclassified examples. Even if multiple errors do interact within a theory, there is usually a repair that will yield some improvement in accuracy.

Table 4.5. Algorithm for finding assumptions for a single example.

```
find_assumptions(theory, example)
  {
    assumptions = ∅
    for each goal literal G tried in the proof of example do
      if G can be solved
        then
          if provable_using_assumption(example, not(G), theory)
            then
              add not(G) to assumptions
        else
          if provable_using_assumption(example, G, theory)
            then
              add G to assumptions
    return assumptions
  }
```

Table 4.6. Algorithm for proving an example using an assumption.

```
provable_using_assumption(example, assumption, theory)
  {
    attempt to prove example using theory
    but before attempting to prove a goal G do
      G' = G without negation
      A' = assumption without negation
      if G' unifies with A'
        then
          if G and assumption are both negated or non-negated
            then goal G succeeds
            else goal G fails and no alternatives are tried
        else
          alternatives for goal G are tried
  }
```

4.4.1 Finding Assumptions for a Single Example

The first step in the algorithm for finding the best assumption is to find all assumptions for each uncovered example (see Table 4.4). If an example can be correctly classified, no assumptions are required. If the example cannot be correctly classified, a set of assumptions must be found. The algorithm for finding a set of assumptions for a single examples is given in Table 4.5. Note that every example has at least one assumption that can be made to prove the example, namely, the example itself.

A3 finds the set of possible assumptions for an example in the following way. First, a proof of the example is attempted. If the proof fails, A3 searches for assumptions. During the attempted proof of the example, every goal literal that succeeds or fails is a candidate for an assumption. If a goal literal `pred(...)` succeeds, A3 tries the assumption `not(pred(...))` and if the goal fails, it tries the assumption `pred(...)`. Conversely, if the goal `not(pred(...))` succeeds (i.e., `pred(...)` is shown to be false), A3 tries the assumption `pred(...)` and if the goal fails (i.e., `pred(...)` is found to be true), it tries the assumption `not(pred(...))`.

Assumptions are literals that must hold true regardless of whether or not they can be proved by the theory. They are intended to point out where a theory is in error and the type of error involved (overly general or overly specific). Therefore all succeeding and failing goals in the proof of an example are candidates for being an assumption.

Assumptions can be made for goals corresponding to literals that are in the background knowledge but not for goals that occur in the bodies of clauses in the background knowledge. For example, consider the following theory:

```
poseven(X) :- even(X), greater(X,0).
greater(X,Y) :- X > Y.
```

If `greater` is part of the background knowledge, A3 can assume the literal `greater(-4, 0)` but not `-4 > 0`. Assumptions are allowed for predicates in the background knowledge because they may indicate an error in a clause using a literal from the background. The assumption `-4 > 0` will not be made because the `greater` predicate is assumed to be correct and there is no need to make assumptions within it.

An assumption for an example will be retained if the example can be proved using the assumption. Table 4.6 gives the algorithm for proving an example using an assumption. Note that, during the proof of an example, an assumption always overrides the theory regardless of what the theory says about the assumption.

To see how A3 finds assumptions for an example, assume the system is given the theory in Table 4.1 where the concept `odd` is overly general and will incorrectly

classify `odd(4)` as a positive example. Because `odd(4)` is a negative example, the goal `not(odd(4))` should be true. In the proof for `not(odd(4))`, A3 first checks the goal `not(even(4))`, which requires trying to prove the goal `even(4)`, which fails and causes the proof of `not(odd(4))` to fail. The system therefore proposes the assumption `even(4)` which would allow the example `not(odd(4))` to be provable using the assumption. Going back to the proof for `not(odd(4))`, the goal `not(even(4))` succeeds because `even(4)` failed. This leads A3 to propose the assumption `even(4)` which was already tried. Finally, the system proposes the assumption `not(odd(4))`, because the goal for `not(odd(4))` fails. So for the given example and theory, A3 finds two assumptions that could be made in order to prove the example: `even(4)` and `not(odd(4))`.

It is important to note that not all proposed assumptions will lead to the example being provable. For example, consider the following theory with only one clause:

```
uncle(U,N) :- brother(U,X), parent(X,N).
```

The positive example `uncle(walt, jim)` cannot be classified correctly and A3 would try to find assumptions that would make it provable. One of the assumptions attempted would be `brother(walt, X)`. This assumption would not be retained because the original example `uncle(walt, jim)` still can not be proved due to the `parent` predicate failing.

4.4.2 Finding the Best Assumption

Assumptions are used to identify the location (which concept) and type of error (overly general or specific) found in the theory. A3 finds the set of all single literal assumptions that could be made to prove each incorrectly classified example. The number of different examples proposing the same assumption is recorded along with the assumption itself. The assumption that can be used to prove the most number of uncovered examples is selected as the best assumption to use as a guide for revising the theory.

When computing the example count for an assumption, two assumptions are considered equivalent if they use the same predicate name, are either both negated, or non-negated literals and are used in the same locations within the theory. This factors out variable bindings from the specific examples and indicates which clauses are incorrect and which literal within the clause is incorrect.

It is possible that more than one assumption will have the maximum count. For example, four assumptions were found in the previous section for the positive example `diff(4, [1,2,4], [2,3])`. If this were the only uncovered example, the four assumptions would each have a count of one. Of the candidate assumptions with

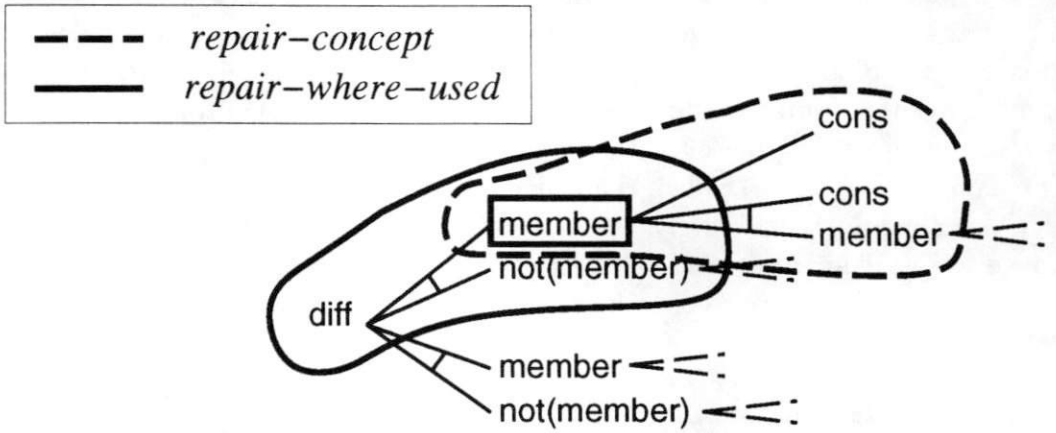

Figure 4.1. Call graph for `diff` showing repair locations.

the same maximum count, A3 chooses those deeper in the call graph. Therefore, in the above example, `not(cons(Y, Xs, [1,2,4]))` would be chosen. If there is more than one assumption at the same depth, A3 randomly selects a best assumption.

4.5 Repairing Errors in a Theory

Once an assumption has been found, A3 attempts to revise the theory so that the examples requiring the assumption may be proved without it. There are two basic choices to be made about how the theory should be repaired.

- The error may be due to some clause or clauses that used the assumption in the body of the clause. In this case, the clauses are assumed to be incorrect and A3 attempts to repair each of them.
- The error may be due to the concept corresponding to the assumption being incorrect. A3 attempts to repair the concept representing the assumption by either generalizing or specializing the concept depending on the context in which the assumption was used.

Figure 4.1 shows the call graph for the symmetric difference theory first presented in Table 4.3. In order to illustrate how A3 repairs a theory based on an assumption, assume that A3 has made the assumption corresponding to the `member` literal in the first clause for `diff`. The portions of the theory that correspond to repairing the clause that uses the assumption, and repairing the concept representing the assumption are shown as circled in Figure 4.1. The solid circle represents the single clause that is repaired on the basis of using the assumption. The dashed circle

Table 4.7. Algorithm for revising theories.

```
modify_theory(assumption, theory, examples)
  {
    T1 = repair_where_used(assumption, theory, examples)
    if assumption is negated
      then
        T2 = specialize_concept(assumption, theory, examples)
      else
        T2 = generalize_concept(assumption, theory, examples)
    return the better of T1 and T2 (T2 if tied)
  }
```

includes both clauses for the `member` concept which is the concept corresponding to the assumed literal. In summary, an assumption indicates that either the assumed literal is being used incorrectly in the theory, or the concept corresponding to the literal is in error.

Table 4.7 gives the algorithm for revising a theory based on an assumption. A negated assumption indicates that its corresponding concept is overly general and must be specialized. Conversely, if the assumed literal is not negated, its corresponding concept is overly specific and must be generalized.

If two revisions have the same accuracy, the one that is the closest distance to the initial theory is preferred. If the two theories are tied on both accuracy and distance measures, A3 prefers repairing the concept corresponding to the assumption.

4.5.1 Using Assumptions to Repair Clauses

When trying to repair clauses that used the assumption, A3 attempts to repair each clause according to how it used the assumption (Table 4.8). A clause uses an assumption if a literal in the body of the clause unifies with the assumption in the successful proof of an example.

Clauses may be repaired through generalization or specialization. If the sign (negated or non-negated) of the assumption is the same as the sign of the goal using the assumption, the clause is too specific and must be generalized. If the signs differ, the clause is overly general and must be specialized.

In the example given above for the positive example `diff(4, [1,2,4], [2,3])`, A3 found the negated assumption `not(cons(Y, Xs, [1,2,4]))`. It is used in both

Table 4.8. Algorithm for repairing clauses that use an assumption.

```
repair_where_used(assumption, theory, examples)
  {
    T' = theory
    for each clause in theory using assumption do
      assumed_goal = goal in clause that used assumption
      if assumption and assumed_goal are both negated or both non-negated
        then
          T' = generalize_clause(clause, assumption, theory, examples)
        else
          T' = specialize_clause(clause, assumption, theory, examples)
    return T'
  }
```

clauses for `member` as a non-negated goal which indicates that the clauses must need to be specialized.

Specializing Clauses

Table 4.9 gives the algorithm for specializing clauses based on an assumption. A3 attempts to specialize a clause by negating the assumed goal, adding a conjunction of new literals to the clause using induction (Quinlan, 1990), and by deleting the clause altogether. The best revision returned is the one with the greatest accuracy on the training examples. In case of a tie, A3 prefers the revision whose distance to the initial theory is smallest. If none of the revisions are more accurate than the theory input to **specialize_clause**, no modification is suggested by this operator.

Table 4.9. Algorithm for specializing clauses.

```
specialize_clause(clause, assumption, theory, examples)
  {
    assumed_goal = goal in clause that used assumption
    return the better revision of:
       • negate assumed_goal in clause
       • specialize clause with literals found by induction
       • delete clause from theory
  }
```

Table 4.10. Algorithm for generalizing clauses.

generalize_clause(*clause*, *assumption*, *theory*, *examples*)
{
assumed_goal = goal in *clause* that used *assumption*
return the better revision of:
- delete *assumed_goal* from *clause*
- replace *assumed_goal* in *clause* with literals found by induction
- inductively learn new clause for *clause* predicate

}

In the example for `diff`, A3 attempts to specialize both clauses for `member` based on the assumption `not(cons(Y, Xs, [1,2,4]))`. The second (and correct) clause cannot be modified to cover the positive example. Similarly, neither deleting the first clause for `member` nor negating the goal for this clause causes the example to be covered. Adding new literals to the clause finds that the literal `equal(X, Y)`[2] correctly specializes the first clause for `member` which correctly repairs the theory for `diff` so that it works on all examples.

The `specialize_clause` operator gives rise to the following three candidate revisions to the first clause for `member` based on the assumption `not(cons(Y, Xs, [1,2,4]))`:

Negate the assumed literal:

```
member(X,Z) :- not(cons(Y,Xs,Z)).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

Specialize the clause:

```
member(X,Z) :- cons(Y,Xs,Z), equal(X,Y).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

Delete the clause:

```
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

[2]For this problem, A3 might also find literals that are semantically equivalent to `equal(X, Y)` in the context of the clause such as `cons(X, Xs, Z)`.

Generalizing Clauses

Table 4.10 gives the algorithm for generalizing clauses based on an assumption. Similar to specializing a clause, the generalization phase trys to modify the clause in three different ways based on the assumption. A clause may be generalized by deleting the goal using the assumption, replacing the goal using the assumption with literals found by induction, or by learning a new clause. As with the specialize clause operator, the best revision returned is the one with the greatest accuracy on the training examples. In case of a tie, A3 prefers the revision whose distance to the initial theory is smallest. If none of revisions is more accurate than the theory input to **generalize_clause**, no modification is suggested by this operator.

As an example of generalizing clauses, consider the following simple theory for the concept `grandparent` consisting of only the (incorrect) clause:

```
grandparent(G,C) :- mother(G,P), parent(P,C).
```

This clause only covers cases where the grandparent is the grandmother. Given examples of the grandfather relationship, A3 would make the assumption `mother(G,P)` for the `grandparent` clause. In attempting to generalize this clause, A3 might produce the following candidate revisions:

Delete assumed goal:

```
grandparent(G,C) :- parent(P,C).
```

Replace assumed goal:

`grandparent(G,C) :-` *parent(G,P),* `parent(P,C).`

Inductively learn new clause:

```
grandparent(G,C) :- mother(G,P), parent(P,C).
```
grandparent(G,C) :- father(G,P), parent(P,C).

Deleting the assumed goal overgeneralizes the concept to the point where anyone is the grandparent of anyone else. Replacing the assumed goal will correctly repair the concept provided the correct literal is used as the replacement. Also, in this case, learning a new clause correctly repairs the concept provided the new clause covers the portion missing from the first (i.e., all grandfather relationships). If there are ties between revisions on both accuracy and distance, A3 choses them in the order listed in Table 4.10. In this case, the revision to the existing clause is preferred to learning a second clause.

Table 4.11. Algorithm for specializing concepts.

```
specialize_concept(assumption, theory, examples)
  {
    C = best_clause_to_specialize(assumption, theory, examples)
    T_D = theory with C deleted from it
    T_S = theory with C specialized by adding literals using induction
    if acc(T_D) ≥ acc(T_S) and acc(T_D) ≥ acc(theory)
       then return T_D
       elseif acc(T_S) > acc(T_D) and acc(T_S) > acc(theory)
          then return T_S
    return no repair found
  }
```

4.5.2 Using Assumptions to Repair Concepts

Along with modifying the clauses in the theory that use the assumption, A3 also attempts to repair the concept for the assumption itself. If the assumption is negated, the corresponding concept must be overly general; and if the assumption is non-negated, it must be too specific. A3 does not attempt to repair a concept if the assumption corresponds to a literal in the background knowledge.

Specializing Concepts

Table 4.11 gives the algorithm for specializing concepts. A3 selects the best clause to specialize and then returns the best revision of deleting the clause and specializing the clause by conjoining new clauses using induction. In the case of a tie, A3 prefers to delete the clause selected for specialization. Note that A3 might choose to delete the best clause even if it does not improve the accuracy of the theory. The reason for this can be seen in the example discussed below.

The algorithm for choosing the best clause to specialize is shown in Table 4.12. The clause is selected on the basis of the incorrectly classified examples that were provable using the assumption. Each clause corresponding to the assumption is removed from the theory in turn and is scored by computing the difference between the average number of proofs for each of the positive examples and the average number of proofs for each of the negative examples. The clause with the highest score is returned as the best one to specialize.

Because A3 specializes a concept based on choosing a single clause to specialize, there may be theories containing overly general concepts for which no single clause

Table 4.12. Algorithm for choosing best clause to specialize.

```
best_clause_to_specialize(assumption, theory, examples)
  {
    P = set of positive examples using assumption
    N = set of negative examples using assumption
    for each clause C in theory corresponding to assumption do
        T' = theory without clause C
        V_C = average number of proofs using T' for each e ∈ P
              − average number of proofs using T' for each e ∈ N
    return clause C with largest V_C
  }
```

may be specialized that will improve the accuracy of the revised theory. An example of such a theory is the following one defining the **parent** relation:

```
parent(X,Y).
parent(X,Y) :- female(Z).
parent(X,Y) :- father(X,Y).
```

Assuming **female** and **father** are background predicates, this theory is overly general because two of the three clauses will match any negative example of the **parent** predicate. The final clause is correct. If the selection of which clause to specialize were based on the accuracy of the theory with each clause removed, all three clauses would look equally good to specialize. Because two of the three clauses are overly general, there is no single clause that can be deleted and show an improvement in accuracy.

The idea behind the method in Table 4.12 for selecting which clause to specialize is that either of the first two clauses should be preferred over the third one because they do not participate in a meaningful way in the classification of examples. The third clause does participate in a meaningful way because it accounts for some of the positive examples but none of the negatives.

An alternative method of scoring the clauses could have been based on the accuracy of the theories with each clause removed. However, this method of scoring clauses is undesirable because it does not distinguish between the three clauses for **parent** in the above example.

Using the method described in Table 4.12, A3 will choose either of the overly general clauses to specialize and not the correct clause which should be left in the theory. To see why, assume there are m positive examples of **parent** due to the person being a mother, f positives corresponding to the father, n negative examples that are

neither fathers nor mothers, and z females in the background knowledge. Deleting the first clause for **parent** leads to a score of:

$$V_1 = \frac{mz + f(1+z)}{m+f} - \frac{zn}{n} = \frac{f}{m+f}$$

because without the first clause, there are mz ways to prove the mother positives and $f(1+z)$ ways to prove the father positives out of a total of $m+f$ positives. Each negative example can be proved by the second clause giving an average of z proofs per negative example. Similarly, deleting the second clause will also give a score of:

$$V_2 = V_1 = \frac{m+2f}{m+f} - \frac{n}{n} = \frac{f}{m+f}$$

Finally, deleting the third clause for **parent** results in a score of:

$$V_3 = \frac{(m+f)(1+z)}{m+f} - \frac{n(1+z)}{n} = 0$$

because without the third clause, there are fewer ways to prove a positive example where the parent is the father. If there are any positive examples where the parent is the father, $f \neq 0$ and $V_1 = V_2 > V_3$. Consequently, either of the first two clauses will be chosen for specialization over the correct third clause.

A3 will therefore delete either of the first two clauses for **parent** even though it does not improve the accuracy of the theory. Once this repair is made, A3 could then delete the other clause for **parent** or specialize it in order to repair the theory correctly.

Table 4.13. Algorithm for generalizing concepts.

```
generalize_concept(assumption, theory, examples)
  {
    predicate = predicate name corresponding to assumption
    T' = inductively learn a new clause for predicate
    return T'
  }
```

Generalizing Concepts

Table 4.13 gives the algorithm for generalizing a concept and consists of simply learning a new clause for the assumption predicate using induction. The induction operator is discussed in the next section. Note that another possibility for the generalize concept operator could have been to try deleting literals from each of the clauses in the concept. However, A3's assumptions are intended to identify such literals and the `repair_where_used` operators should perform this task.

The operators for generalizing and specializing concepts are in some sense simpler than those for clauses because the assumptions that guide the revision process give little information as to what is wrong with the concept other than being overly general or specific. In this case, more precise information about which clause or clauses in the concept need to be repaired is missing. In contrast, the assumptions give more precise information about how to revise the theory where they are used, namely which clause and which literal in the clause need to use the assumption.

Alternate approaches to the operators for generalizing and specializing concepts could include choosing some subset of the clauses belonging to a concept and either generalizing or specializing them all at once. However, experiments indicate that the simple approach taken by A3 works well but further investigation into alternate approaches would be worthwhile.

4.6 Clause Induction

The induction operator in A3 for learning new clauses and specializing clauses is a variant of FOIL's `FindClause` (see Table 2.2) operator. A clause is built through a hill-climbing process of adding new literals to the clause under construction. The best literal to add is the one that gives the best coverage of the top-level input examples. Therefore, in the example for the concept `diff`, specializing a clause for `member` would determine the best literal to add by computing the coverage of examples for `diff`.

Table 4.14. A3's algorithm for clause induction.

```
induce_clause(clause, theory, examples)
  {
    T = add clause to theory
    while accuracy of T on examples is improving do
      literal = best_literal(clause, T, examples)
      add literal to clause in T
    return T
  }
```

Table 4.14 shows the top-level algorithm for inducing new clauses or specializing existing clauses. The main differences between A3's clause induction and FOIL's are that A3 uses only the input examples and does not keep track of tuples, the sets of positive and negative examples covered by the clause under construction are not reduced by A3 when a new literal is added, and A3 uses a simpler evaluation function when deciding which new literal to add to a clause, namely, choosing the literal that gives the most improvement in accuracy.

Because A3 learns clauses within the context of a theory, one cannot assume that adding literals to a clause will specialize other concepts using that clause. However, FOIL does learn clauses in isolation from other concepts and so can assume that when a literal is added to a clause that all examples (tuples) that are not covered by the clause need not be checked when adding further literals. Because of this, FOIL's algorithm for learning a single clause can be made more efficient by trimming the list of examples to check as they are no longer covered by the clause under construction.

A3's algorithm for learning new clauses (Table 4.14) cannot exclude uncovered examples once the clause being built no longer covers them because specializing one clause in a theory may cause another to be generalized. Also, A3 cannot use the same termination criterion as does FOIL and must instead stop when no new literal can improve the accuracy of the theory. The following example demonstrates the differences between A3 and FOIL.

Consider the following incorrect theory for odd and even numbers:

```
odd(X) :- not(even(X)).
even(X).
```

The definition of **even** is overly general and states that everything is even. Consequently **odd** is overly specific and states that nothing is odd. Now assume that to repair this theory A3 decides to specialize the clause **even(X)** and is only

given the positive examples `odd(1)`, `odd(3)` and the negative examples `not(odd(2))`, `not(odd(4))`. Only `not(odd(2))` and `not(odd(4))` are correctly classified.

If A3 were to weed out those examples not provable by the theory, none of the above examples would be available to compute the evaluation function for new literals. The two negative examples are correctly classified because they are not provable and the two positives are incorrectly classified because they are also not provable. Consequently, A3 must use *all* the available examples when computing its evaluation function because one cannot assume, as one can in FOIL, that adding literals to a clause strictly specializes a theory.

Another consequent of needing to compute the evaluation function over all the input examples is that A3 cannot use FOIL's termination criterion that the set of negative examples to exclude is empty. Rather, A3 terminates when no new literal can be added to the clause that will improve the overall accuracy of the learned theory.

A3 behaves as FOIL when no initial theory is present because it attempts to learn a set of clauses to cover the input examples. In this case, A3's method of retaining all examples for evaluation and stopping when no new literal improves the evaluation function is less efficient but still equivalent to the control structure used by FOIL.

4.6.1 Evaluation Function

Whereas FOIL uses an information-based heuristic for evaluating which new literal to add to a clause, A3 uses the simpler measure of accuracy of the theory on the input training set. There are several reasons for doing this. A3's primary control structure must at times choose between different revisions to a theory and does so on the basis of overall accuracy on the input examples. It is therefore reasonable that all of A3's operators use the same measure. Mingers (1989) showed that classification accuracy for decision trees is not very sensitive to the choice of evaluation function. Even though A3 uses a first-order Horn clause representation, the selection of literals to add to a clause is quite similar to the selection of an attribute when building a decision tree.

It has not been shown that FOIL's use of tuple counts in the evaluation function measures any meaningful aspect of the concept to be learned. In fact, this aspect of FOIL's evaluation function biases the selection of literals towards those that introduce new variables. The introduction of literals with new variables is very problematic for approaches similar to FOIL as was mentioned earlier in Section 2.1.3. Solutions to this must directly address the problem rather than bury it in the evaluation function. Other attempts at solving this problem include CHAM (Kijsirikul, Numao

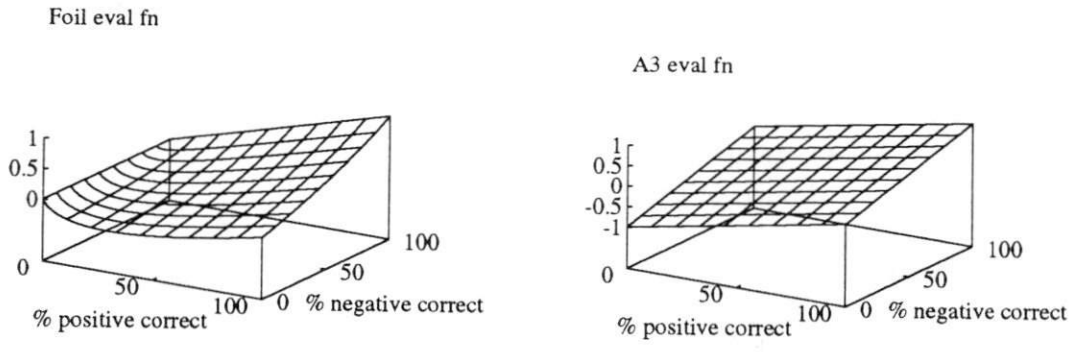

Figure 4.2. Comparison of FOIL's and A3's evaluation function.

& Shimura, 1991), FOIL with determinate literals (Quinlan, 1991), and FORTE's relational pathfinding component (Richards & Mooney, 1992).

A3 and FOIL's evaluation functions are actually quite similar in behavior. The two system's evaluation functions are given as follows:

$$Eval_{FOIL} = T_i^{++}\left(-log_2(\frac{T_i^+}{T_i^+ + T_i^-}) + log_2(\frac{T_{i+1}^+}{T_{i+1}^+ + T_{i+1}^-})\right)$$

$$Eval_{A3} = \frac{P_{correct} + N_{correct}}{P_{total} + N_{total}}$$

Assuming that no new variables are introduced into a new literal then $T_i^{++} = T_{i+1}^+$. Figure 4.2 shows the surfaces of both evaluation functions as a function of the proportion of positive and negative examples that are correct assuming equal proportions of positive and negative input examples.[3] The graphs have been normalized to have a maximum value of one. One can readily see that the two surfaces are almost indistinguishable in the region that FOIL's gain is greater than zero.

4.7 An Example Trace

This section presents a simple example of A3's operation on a theory containing multiple errors. The correct theory found in Table 4.15 is used to determine whether

[3]The difference between the two surfaces is greatest when the initial numbers of positive and negative examples are equal. As the ratio of initial positives to negatives approaches zero or infinity, the distance between the two surfaces approaches zero.

Table 4.15. A correct theory for determining voter eligibility.

```
vote(X) :- uscitizen(X), not(felon(X)).
uscitizen(X) :- born(X,Y), usa(Y).
uscitizen(X) :- naturalized(X).
felon(X) :- convicted(X,Y), felony(Y).

/* background clauses */
felony(murder).
misdemeanor(indecent-exposure).
usa(usa).
england(england).
```

a person is eligible to vote in the United States. The theory states that a person is eligible to vote if they are a U.S. citizen and have not been convicted of a felony.[4] The theory contains two intermediate concepts: `uscitizen` and `felon`. A person is considered a U.S. citizen if they were either born in the U.S. or if they have become a naturalized citizen. A felon is a person who has been convicted of a felony.

The input examples and background information given to A3 are found in Table 4.16. The background information represents literals that are present in the input theory but not shown in Table 4.15. For example, person `p1` is a male, a felon, and was born in the U.S. which means the ground clauses `male(p1)`, `born(p1, usa)`, and `convicted(p1, murder)` can be found in the theory. The examples given to A3 are of the form `vote(p5)` or `not(vote(p1))`.

Table 4.17 presents an incorrect version of the theory found in Table 4.15. This theory has four different types of errors: an extra clause, a missing clause, an extra literal, and a missing literal. The extra clause is for the `vote` predicate. The `uscitizen` predicate is missing a clause and one of its clauses is missing a literal. The clause for `felon` contains an extra literal.

The theory in Table 4.17 correctly classifies all of the eight positive examples and none of the ten negative examples. The misclassifications are due to the incorrect clause `vote(X)` which matches all examples and consequently gets all of the negatives wrong. Failing to correctly classify all of the training examples A3 finds assumptions to locate an error in the theory. A3 finds only one assumption for each of the negative examples:

[4]This theory has been intentionally simplified. For example, it omits the requirement that the person is over eighteen years in age.

Table 4.16. Examples and background information for the voter eligibility theory.

Examples		Background Information			
name	can vote?	born	naturalized	crime	male
p1	no	usa	no	murder	yes
p2	yes	usa	no	indecent-exposure	yes
p3	yes	usa	no	none	yes
p4	no	england	yes	murder	yes
p5	yes	england	yes	indecent-exposure	yes
p6	yes	england	yes	none	yes
p7	no	england	no	murder	yes
p8	no	england	no	indecent-exposure	yes
p9	no	england	no	none	yes
p10	no	usa	no	murder	no
p11	yes	usa	no	indecent-exposure	no
p12	yes	usa	no	none	no
p13	no	england	yes	murder	no
p14	yes	england	yes	indecent-exposure	no
p15	yes	england	yes	none	no
p16	no	england	no	murder	no
p17	no	england	no	indecent-exposure	no
p18	no	england	no	none	no

Table 4.17. An incorrect theory for determining voter eligibility.

```
vote(X) :- uscitizen(X), not(felon(X)).
vote(X).
uscitizen(X) :- born(X,Y).
felon(X) :- convicted(X,Y), felony(Y), male(X).

/* background clauses */
felony(murder).
misdemeanor(indecent-exposure).
usa(usa).
england(england).
```

# Examples Using Assumption	Assumed Literal
10	`not(vote p7))`

The assumption `not(vote(p7))`[5] is used by all of the ten negative examples. A3 attempts to repair the theory by repairing the clause using the assumption and by repairing the concept corresponding to the assumption. Because the assumption `not(vote(p7))` is at the top-level, there are no clauses that use it. Consequently, A3 only tries to specialize the concept `vote`.

To specialize `vote`, A3 first chooses the clause for `vote` that is used in the most number of incorrect classifications of examples. In this case, the clause `vote(X)` is chosen. A3 considers deleting the clause and it considers specializing the clause by adding new literals. The better of these two revisions is retained. The results of these two revisions are given below along with the total number of examples correctly classified and the distance between the revised theory and the initial theory.

- *Delete Clause*: 11 correct, distance to initial is 1.
- *Add Literals*: 11 correct, distance to initial is 1.

  ```
  vote(X) :- not(convicted(X,Y)).
  ```

In the case of a tie on both accuracy and distance, A3 always prefers the delete clause operator which in this case leads to the following theory:

```
vote(X) :- uscitizen(X), not(felon(X)).
uscitizen(X) :- born(X,Y).
felon(X) :- convicted(X,Y), felony(Y), male(X).
```

This revised theory still correctly classifies all the positive examples, but misclassifies seven of the ten negative examples. A3 finds the following assumptions for the seven incorrect negatives:

# Examples Using Assumption	Assumed Literal
7	`not(born(p13,Y))`
7	`felon(p13)`
7	`not(uscitizen(p13))`
7	`not(vote(p13))`
3	`male(p13)`
2	`convicted(p8,Y)`
2	`felony(indecent-exposure)`

[5]The assumptions are written with specific bindings for some of the arguments. A3 ignores these variable bindings when considering whether or not two assumptions are equivalent.

There are two deepest assumptions covering the same number of examples: `not(born(p13, Y))` and `felon(p13)`. A3 chooses among these randomly so assume `not(born(p13, Y))` is chosen.[6] To repair the theory, A3 considers specializing the clause using the assumption as well as specializing the concept corresponding to the assumption.

The clause for `uscitizen` uses the assumption `not(born(p13, Y))` and is specialized by deleting the clause, negating the goal corresponding to the assumption, and by adding new literals to the clause. A3 generates the following candidate revisions:

- *Delete Clause*: 10 correct, distance to initial is 3.
- *Negate Goal*: 10 correct, distance to initial is 2.

  ```
  uscitizen(X) :- not(born(X,Y)).
  ```

- *Add Literals*: 13 correct, distance to initial is 2.

  ```
  uscitizen(X) :- born(X, Y), usa(Y).
  ```

Because the concept for the assumption `not(born(p13,Y))` is in the background knowledge, A3 does not attempt to repair the concept. Therefore A3 chooses to add a new literal the to clause for `uscitizen` resulting in the following theory:

```
vote(X) :- uscitizen(X), not(felon(X)).
uscitizen(X) :- born(X,Y), usa(Y).
felon(X) :- convicted(X,Y), felony(Y), male(X).
```

This theory now incorrectly classifies four of the eight positive examples but only one of the negative examples. To locate further errors in the theory, A3 finds the following assumptions based on the incorrectly classified examples:

# Examples Using Assumption	Assumed Literal
4	`usa(england)`
4	`born(p15, Y)`
4	`uscitizen(p15)`
4	`vote(p15)`
1	`not(usa(usa))`
1	`not(born(p10, Y))`
1	`felon(p10)`
1	`male(p10)`
1	`not(uscitizen(p10))`
1	`not(vote(p10))`

[6]Note that both assumptions correspond to different errors that occur in the theory.

There are two deepest assumptions covering the same number of examples: `usa(england)` and `born(p13,Y)`. A3 chooses among these randomly so assume that `usa(england)` is chosen. To repair the theory, A3 attempts to both generalize the clause using the assumption as well as generalize the concept corresponding to the assumption. However, because the assumption corresponds to background knowledge, A3 only attempts to generalize the clause using the assumption.

The clause `uscitizen(X) :- born(X,Y), usa(Y)` is generalized by deleting the goal using the assumption, replacing the goal with new literals, and by learning a new clause for the predicate `uscitizen`.

- *Delete Goal*: 11 correct, distance to initial is 1.

  ```
  uscitizen(X) :- born(X,Y).
  ```

- *Replace Goal*: 13 correct, distance to initial is 2.

  ```
  uscitizen(X) :- born(X,Y), naturalized(X).
  ```

- *Learn new clause*: 16 correct, distance to initial is 3.

  ```
  uscitizen(X) :- naturalized(X).
  ```

The best revision found by A3 is to add the new clause for `uscitizen` resulting in the following theory:

```
vote(X) :- uscitizen(X), not(felon(X)).
uscitizen(X) :- born(X,Y), usa(Y).
uscitizen(X) :- naturalized(X).
felon(X) :- convicted(X,Y), felony(Y), male(X).
```

The theory is now almost completely repaired except for the clause for `felon` which causes two of the negative examples to be incorrectly classified. A3 find the following assumptions for these two examples:

# Examples Using Assumption	Assumed Literal
2	`male(p13)`
2	`not (uscitizen(p13))`
2	`felon(p13)`
2	`not(vote(p13))`
1	`not(naturalized(p13))`
1	`not(usa(usa))`
1	`not(born(p10, Y))`

The deepest assumption covering the most number of examples is `male(p13)`. A3 attempts to repair the theory by generalizing the clause using the assumption and

by generalizing the concept corresponding to the assumption. Because the assumption `male(p13)` is in the background knowledge, A3 does not attempt to generalize the concept. The clause `felon(X) :- convicted(X, Y), felony(Y), male(X)` uses the assumption and is generalized by deleting the goal corresponding to the assumption, replacing the goal with new literals, and by learning a new clause for the predicate `felon`:

- *Delete Goal*: 18 correct, distance to initial 4.

  ```
  felon(X) :- convicted(X,Y), felony(Y).
  ```

- *Replace Goal*: 18 correct, distance to initial 4.

  ```
  felon(X) :- convicted(X,Y), felony(Y).
  ```

- *Learn new clause*: 16 correct, distance to initial 5.

  ```
  felon(X) :- not(convicted(X, Y)).
  ```

Note that both the delete goal and replace goal operator make the same revision which classifies all eighteen examples correctly. The replace goal operator did not find any literals to add in place of the goal `male(X)` that were able to improve the accuracy of the theory. A3 therefore chooses to delete the goal `male(X)` from the clause for `felon` producing the final and correct revision of the initial theory:

```
vote(X) :- uscitizen(X), not(felon(X)).
uscitizen(X) :- born(X,Y), usa(Y).
uscitizen(X) :- naturalized(X).
felon(X) :- convicted(X,Y), felony(Y).
```

4.8 Discussion

This chapter has presented the details of the A3 system for revising function-free first-order theories containing negation. There are several important aspects of the system that should be noted.

As stated earlier, one of the goals of the theory revision task is to learn correct theories that are minimal revisions of the incorrect input theory. One of the unique features of A3 is that it has a well defined notion of the degree of revision. A minimally revised theory is one whose edit distance from the initial theory is smallest over the set of all theories that correctly classify the training examples. A3 uses this metric in its evaluation function for deciding on which candidate revision to make. The system prefers more accurate theories but breaks ties using the distance metric. Although this does not guarantee that A3 will find an optimal solution the use of the distance metric for selecting revisions will bias it towards making minimal revisions of the input theory.

A3 derives much of its power through its assumption finding mechanism. These assumptions play a crucial role in both locating and repairing errors in a theory. Assumptions are generated on the basis of incorrectly classified examples. The system uniformly treats positive and negative examples as literals that should be provable by the theory. Thus the negative example of an even number `even(3)` can be represented as `not(even(3))`. Using negation-as-failure and the closed-world assumption a negative example such as `not(even(3))` should be provable by a correct theory. An incorrectly classified example is therefore one which has no proof. Assumptions in A3 are made in order to allow the proof of an example to complete. Consequently, an assumption could be used by more than one example, be it positive or negative, or by examples of different classes altogether.

The most commonly occurring assumption over the set of incorrectly classified examples serves as an indication of an error in the theory. By focusing on the most common errors first, A3 performs a hill-climbing search for revised theories. Because the same assumption can be used by both positive and negative examples as well as examples of different concepts in the theory, A3 is able to make repairs that have the largest impact over the entire set of examples.

Most learning systems decide to generalize when a positive example is uncovered or to specialize when a negative example is covered by the concept description being built. However, as shown earlier in this chapter, a theory containing negation does not have this property. A3 avoids this problem altogether through the use of assumptions. The form of the assumption and how it is used provide the basis for deciding which concepts in the theory need to be generalized and which need to be specialized. No longer does the system decide this on the basis of the positive or negative examples themselves; rather, the sign of the assumed literal (negated or non-negated) indicates the need for specialization or generalization. It is interesting to note that in the special case where there are no intermediate concepts in the theory, the assumptions will correspond directly to the input examples: incorrectly classified positives will indicate the need for generalization and incorrectly classified negatives will indicate the need for specialization.

Finally, A3 demonstrates how an existing method for learning first-order concepts can be used in the larger theory revision task. A3 has operators for generalizing and specializing clauses and one of the methods for specializing clauses is based on techniques used in FOIL. Once again, a hill-climbing search through the space of literals to add to a clause forms the basis for clause specialization. A3 generalizes the FOIL technique in order to specialize a clause within the context of a larger theory being repaired. In the special case where there is no larger context and only clauses for a single concept are being built, A3's method for clause specialization is equivalent to FOIL's.

Chapter 5
Validation of the Method

This chapter explores A3's ability to perform theory revision on a variety of domains. The learned theories are evaluated in terms of accuracy as well as the distances between the initial, learned, and correct theories.

The first section presents demonstrations of A3's ability to repair complex theories. A3 is shown to be able to repair recursive logic programs containing errors within the scope of a negation.

The second section presents the results of systematic evaluation of A3's abilities to repair theories containing varying degrees and types of errors. Two very different domains are used: the king-rook-king problem and a theory of common sense moral reasoning (Shultz, 1990). A correct theory for each domain is mutated by varying degrees with varying types of errors and given to A3 to revise. The revised theories are then evaluated in terms of both accuracy and distance. The results show that A3 does not handle all types of errors equally well but that its performance degrades gracefully with increased complexity.

The final set of experiments demonstrates A3's ability to learn from examples of multiple concepts in a theory. Because some errors in a theory can be corrected with repairs at different locations, further constraints on the learning task are required in order for the system to learn theories that are close to the correct one. Examples of intermediate-level concepts provide one form of such constraints. The experiments with intermediate concepts demonstrate how A3 uniformly handles examples of multiple concepts in order to learn theories that are closer to the correct one than if only top-level examples were available.

5.1 Demonstrations of Theory Revision

This section demonstrates A3's ability to repair incorrect theories containing negation. As mentioned earlier, other first-order theory revision systems, including KR-FOCL (Pazzani & Brunk, 1991), AUDREY (Wogulis, 1991), CLINT (De Raedt, 1991), FORTE (Richards & Mooney, 1991; Richards, 1992), and $\mathcal{RLA}$ (Tangkitvanich,

Numao & Shimura, 1992) are not fully able to revise theories containing errors within the scope of a negation. Most of these systems can add and delete clauses containing negated literals or even add and delete negated literals from clauses. However, the control structures of these systems are not able to locate or explicitly repair errors within a negation. The following sections demonstrate A3's ability to repair theories with negation on a number of domains.

5.1.1 A Simple Domain: Odd and Even Numbers

The first domain examined is the very simple one presented in Table 4.1 for determining whether a number is odd or even. The theory contains the following two clauses:

```
odd(X) :- not(even(X)).
even(2).
```

Once again, note that the concept **even** is too specific because it is only correct for the value two. Consequently, the concept **odd** is overly general and is incorrect for all even numbers except two. All training sets of examples for these concepts can only include four types of examples:

`odd(`*num*`)`
`not(odd(`*num*`))`
`even(`*num*`)`
`not(even(`*num*`))`

where **num** is an integer. All examples belonging to the class **odd(num)** are correct even though for the wrong reason, and all examples in the class **not(odd(num))** except for **not(odd(2))** are incorrectly classified. Similarly, all examples of the class **not(even(num))** are correct and all examples of the class **even(num)** except for **even(2)** are incorrect.

Given a training set for this problem, A3 will first try to find an error in the theory that can account for the most number of incorrectly classified examples. The incorrect examples fall into two classes: **not(odd(num))** and **even(num)**. Because there is only a single clause for the concept **even**, all of the incorrectly classified examples of **even** could be proved by making the assumption **even(num)**. Every incorrect example from the class **not(odd(num))** has two assumptions that would make it provable, namely **not(odd(num))** and **even(num)** of which the later would be chosen because it is deeper in the call graph than the first. Regardless of the proportion of examples between these different classes, A3 would always choose **even(num)** as the best assumption to use as a guide for revising the theory. Having correctly identified the error in the theory, A3 will repair the theory in the correct place given

sufficient background knowledge to express even numbers. For example, given predicates `mod2(X,Y)` and `zerop(X)`, A3 will repair the definition of `even` to be `even(X) :- mod2(X,Y),zerop(Y)` A3 will not repair the theory by modifying the clause for `odd` because that would not improve the coverage for the incorrect examples of `even`.

5.1.2 Logic Programs

Recursion and Negation: Symmetric Difference of Lists

A more complex theory that A3 is able to repair was presented in Table 4.3 for finding elements in the symmetric difference between two lists. Shapiro (1983) presented this theory to show how MIS could solve the problem with a single positive example and the use of an oracle. This theory was used in Chapter 4 to demonstrate the workings of different components of A3 which is able to correctly repair the theory given only the positive example `diff(4,[1,2,4],[2,3])`. In both A3 and MIS, the positive example is used to identify the error. However, A3 is also able to use the positive example to correct the error in the definition of `member` whereas the MIS system required answers from an oracle about the intended meaning of the `member` predicate.

The clauses from the incorrect theory are as follows:

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).

member(X,Z) :- cons(Y,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

Note that the concept `diff` is overly specific because the error occurs in the first clause for `member` (indicated by italics) making it overly general. In order to repair this theory, A3 must at least be given positive examples of the predicate `diff`. Positive examples are required to find the location of the error in the `member` concept because negative examples are correctly classified and do not require finding assumptions.

All positive examples of the `diff` predicate will be incorrectly classified and can be used by A3 to generate assumptions that locate the error in the theory. In fact, all positive examples of the form `diff(`n_1`, `l_1`, `l_2`)` will lead to the generation of the same best assumption `not(cons(Y, Xs, `l_1`))` used in both clauses for the `member` predicate. Because all negative examples are correctly classified (even if for the wrong reason), none will lead to the generation of assumptions. Consequently, for any training set of positive and negative examples of the predicate `diff`, A3 will correctly locate the error in the theory as indicated by the assumption `not(cons(Y, Xs, `l_1`))`.

Table 5.1. Positive and negative examples of symmetric difference.

```
diff(1, [2,4,1], [2,3])
diff(3, [1,2,4], [2,3])
diff(4, [1,2,4], [2,3])
diff(2, [3,2], [])
diff(2, [], [1,2])

not(diff(1, [1,2,4], [2,1,3]))
not(diff(2, [1,2,4], [2,3]))
not(diff(6, [1,2,4], [2,3]))
not(diff(7, [1,2,4], [2,3]))
not(diff(1, [], []))
```

The assumption essentially states that the first list argument, l_1, for the `diff` predicate can not be decomposed. Consequently, neither clause for `member` will work when membership is asked of l_1 but will work for the list l_2. This assumption then allows the example `diff(4,[1,2,4],[2,3])` to be provable.

In order to repair the theory, A3 must be given examples that can be used to specialize the concept for `member`. This is the role played by the two oracle queries presented by Shapiro: `member(4,[1,2,4])` and `member(4,[2,3])`. The answers to these two queries amount to giving MIS both a positive and negative example of the `member` predicate. A3 is able to use the positive example `diff(4,[1,2,4],[2,3])` to fill this role because correctly classifying the example using the first clause for `diff` requires both proving that the goal `member(4,[1,2,4])` is true and that `member(4,[2,3])` is false. Unless A3 is able to repair the concept for `member` such that these queries are true, the positive example `diff(4,[1,2,4],[2,3])` will not be correctly classified. Given the background clause `equal(X, X) :- true`, A3 will then correctly repair the first clause for `member` to be `member(X,Z) :- cons(Y,Xs,Z), equal(X,Y)` by using the algorithm for repairing a concept where an assumption is used (Table 4.8).

While A3 is able to repair the incorrect theory for `diff` given by Shapiro, it is also able to repair other types of errors that could occur in this theory. The following experiments demonstrate how A3 repairs errors in this theory consisting of missing literals and missing clauses given a small training set of positive and negative examples of the concept `diff`. Incorrect versions of the theory are generated by deleting each literal and each clause in turn. Because the two clauses for `diff` are symmetric, only one of them will be modified to demonstrate A3's behavior.

All of the experiments use the training set consisting of the five positive and five negative examples shown in Table 5.1. The correct theory and available background knowledge are given in Table 5.2.

Table 5.2. Correct theory and background knowledge for symmetric difference.

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).

member(X,Z) :- cons(X,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).

/* background knowledge */

null([]).
cons(X,Y,[X|Y]).

eq(X,X).
```

The first test case involves deleting the first clause for `diff` giving the following overly specific theory that correctly classifies two of the five positive examples and all five of the negative examples:

```
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(X,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

To cover the three incorrect positive examples, A3 finds that the single assumption `diff(2, [3, 2], [])` can be used for each one. Note that this assumption is reported with constants from a specific example. Even though the constants are different for each example, all three use equivalent assumptions. From here on, assumptions will be presented for a specific example even though different examples use an equivalent one. The assumption `diff(2, [3, 2], [])` indicates that the concept for `diff` is overly specific and A3 uses the operator **generalize_concept** to repair it. A3 generalizes concepts by inducing a new clause and finds the correct clause `diff(A, B, C) :- member(A, B), not(member(A, C))` to repair the theory.

The second test case involves deleting the first clause for the `member` concept resulting in the following overly specific theory that correctly classifies none of the five positive examples but all of the five negatives:

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

A3 finds that the assumption `member(4, [4])` will allow the proof of all the five incorrectly classified positive examples. Notice that this assumption corresponds

to the base case for a recursive definition of `member` which is the missing clause from the theory. The assumption indicates that the `member` concept must be generalized and A3 attempts to learn a new clause using induction. From the examples provided, A3 learns the correct clause for the base case: `member(A, B) :- cons(A, C, B).`

The final test case for deleting clauses results in the following theory that correctly classifies none of the five positive examples and three of negatives. Note that this theory is both overly general and overly specific:

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(X,Xs,Z).
```

A3 finds that the assumption `cons(2, X, [1,2,4])` allows all five of the incorrect positives and both of the incorrect negatives to be correctly classified. Notice that this single assumption is used by both positive and negative examples. The assumption indicates that either the concept `cons` or `member`, which uses `cons`, is overly specific and must be generalized. Because `cons` is part of the background knowledge, A3 only attempts to repair clauses for `member`. Using the **repair_where_used** operator, A3 is able to induce a new clause for `member` which is the correct missing clause: `member(A, B) :- cons(C, D, B), member(A, D).`

The next set of test cases involves deleting single literals from the clauses in the correct theory. Again, because the two clauses for `diff` are symmetric, only literals from one are deleted. The first test case deletes the first literal from the first clause for `diff` resulting in the following overly general theory that correctly classifies all five positive examples but only two of the five negatives:

```
diff(X,Ys,Zs) :- not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(X,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

A3 finds that the assumption `cons(7, X, [2,3])` will cover all three incorrect negative examples. Because `cons` is a part of the background knowledge, A3 only attempts to repair clauses using the assumption which in this case are the clauses for `member`. However, A3 is unable to repair the clauses for `member` to improve the accuracy of the theory. Consequently, A3 uses the next best assumption, `not(diff(1, [], []))` which is also used by all three negatives. That the assumption is negated means that the concept for `diff` must be specialized. A3 chooses the first clause as the best to specialize because it is responsible for misclassifying some of the negatives. The clause is then specialized by adding new literals using induction and A3 repairs the incorrect clause to be: `diff(X, Ys, Zs) :- not(member(X, Zs)), member(X, Ys).`

The next test case for deleting a literal produces the following overly general theory that correctly classifies all five positive examples but only three of the five negatives:

```
diff(X,Ys,Zs) :- member(X,Ys).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(X,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

A3 finds that the assumption `not(diff(2, [1,2,4], [2,3]))` covers both incorrect negatives. The assumption indicates that the concept for `diff` must be specialized and A3 uses the operator **specialize_concept** to select the first incorrect clause for `diff` to specialize. The clause is specialized by adding new literals and A3 learns the correct clause:

```
diff(X, Ys, Zs) :- member(X, Ys), not(member(X, Zs)).
```

The third test case deletes the `member(X, Ys)` literal from the second clause for `member` resulting in the following overly specific theory that correctly classifies two of the five positives and all five of the negatives:

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(X,Xs,Z).
member(X,Z) :- cons(Y,Ys,Z).
```

All three of the incorrect positive examples can be correctly classified using the assumption `not(cons(Y, Ys, [1,2,4]))`. A3 attempts to repair the theory by specializing the clauses that use the assumption. The incorrect clause for `member` is specialized by adding the correct literal to produce the correct repair: `member(X, Z) :- cons(Y, Ys, Z), member(X, Ys).`

The next test case involves deleting the literal for the base-case clause for `member` producing the following overly specific theory that correctly classifies none of the positive examples and all of the negatives:

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z).
member(X,Z) :- cons(Y,Ys,Z), member(X,Ys).
```

The incorrect positive examples can be correctly classified by making the assumption `not(member(4, [2,3]))` which indicates that either the `member` concept or clauses that use it must be specialized. The correct repair is to specialize the first clause for `member`. However, using the **specialize_concept** operator, A3 is unable to determine which clause for `member` is the best one to specialize and so does not

learn the correct repair to the theory. Instead, A3 repairs the theory by adding the clause `diff(A, B, C) :- not(eq(B, C))` which improves the accuracy of the initial theory but does not cover all of the training examples.

The final test case deletes the first literal in the second clause for `member` producing the following overly specific theory that correctly classifies none of the five positive examples and all of the negatives:

```
diff(X,Ys,Zs) :- member(X,Ys), not(member(X,Zs)).
diff(X,Ys,Zs) :- member(X,Zs), not(member(X,Ys)).
member(X,Z) :- cons(X,Xs,Z).
member(X,Z) :- member(X,Ys).
```

For the incorrectly classified positive examples, A3 only finds the assumption `diff(4, [1,2,4], [2,3])` indicating that the concept `diff` is overly specific. However, A3 is unable to generalize the concept `diff` to improve accuracy. Given the additional positive example `diff(1,[1],[])`, A3 would also find the assumption `not(member(1, []))`. This assumption indicates that the concept `member` is overly general. One of the operators used to repair `member` is to add new literals which will attempt to make the repair: `member(X, Z) :- member(X, Ys), cons(A, B, Ys)`. However, because of the way A3 does theorem proving, this clause will not work correctly due to adding the literal to the end of the clause.[1] Were A3 able to add the new literal before the `member(X, Ys)` literal, it would have found the correct repair to the theory.

These test cases show that A3 can repair a large variety of errors in recursive theories containing negation. Those errors that A3 could not repair point out some weaknesses in the system that will be discussed later.

A Complex Logic Program: N-Queens

To test A3 on a more complex logic programming domain, the system was provided with an incorrect theory for the five queens problem as shown in Table 5.3[2]. The theory contains the same error in the definition of `member` as was in the incorrect theory for `diff`. Note that in the theory for determining elements in the logical difference between two lists, the `member` predicate is used as a test whereas it is used to generate bindings within the `attacks` predicate from the five-queens problem.

[1] Both A3 and Prolog will go into infinite recursion when the item is not a member of the list. A3 uses a depth-bound theorem prover and chooses not to make a repair if the bound is exceeded during the proof of any example.

[2] Adapted from (Bratko, 1986).

Table 5.3. Incorrect domain theory for the five-queens problem.

```
solution([]).

solution(S) :-
   cons(A,Others,S),
   cons(X,[Y],A),
   solution(Others),
   member(Y,[1,2,3,4,5]),
   not(attacks(X,Y,Others)).

attacks(X,Y,Others) :-
   member([X1,Y1],Others),
   attacking(X,Y,X1,Y1).

attacking(X,Y,X1,Y1) :- eq(Y1,Y).
attacking(X,Y,X1,Y1) :- Y1 is (Y+X1-X).
attacking(X,Y,X1,Y1) :- Y1 is (Y-X1+X).

member(A,R) :- cons(B,L,R).
member(A,R) :- cons(B,L,R), member(A,L).
```

As with the list difference problem, all negative and no positive examples of `solution` are correctly classified by the initial theory. Consequently, positive examples are required in order to locate where the theory needs to be repaired. For any positive example of the form `solution(`l_1`)`, A3 finds that making the assumption `cons(X, Y,` l_1`)` will allow the example to be proved. Note that this is different than the assumption made for the symmetric list difference theory which was `not(cons(X, Y,` l_1`))`.

The assumption `cons(X, Y,` l_1`)` is used in two places in the second clause for `solution` and in both clauses for `member`. The deepest use of the assumption occurs in the `member` clauses. Because the assumption is not negated, A3 tries to generalize the clauses for `member`. The best alternative chosen by A3 is to replace the `cons` literal in the first clause for `member` with a new one found through induction: `cons(A,L,R)`. This repair will perform best on any input training examples because it completely and correctly repairs the theory.

5.2 Systematic Evaluation

This section attempts to characterize A3's ability to revise theories containing different types of errors. This is done by controlling the types of errors in the initial theory and measuring the system's resulting performance. These experiments provide empirical evidence that A3 is able to repair a wide variety of theory errors.

In order to perform these experiments, two domains were chosen, the king-rook-king chess end game and a theory that simulates moral reasoning (Shultz, 1990). The experiments involved systematically introducing different types and degrees of error and then measuring A3's performance after having been presented with differing numbers of training examples. The dependent performance variables include accuracy, the distance of the revised theory from the initial theory, and the distance between the revised theory and the correct one. The independent variables are the number of training examples, the type of theory errors introduced and the number of errors introduced.

5.2.1 Moral Reasoner: A Complex Domain

As a test of A3's ability to repair larger "real-world" domains, this section investigates Shultz's (Shultz, 1990; Darley & Shultz, 1990) MR (Moral Reasoner) which is a rule-based model that qualitatively simulates moral reasoning. The model was intended to simulate how an ordinary person, down to about age five, reasons about harm-doing.

The Domain

Shultz explains that the MR system has been implemented using production rules as well as LISP functions returning true or false according to the rules they encode. Translating these rules into a Horn clause representation is quite straight forward. The Horn clause rules for MR are listed in Appendix A. Figure 5.1 shows the relationship between the different concepts from the domain in the form of an AND-OR graph. Each node in the graph represents a concept and is connected to the concepts that comprise its definition. Nodes prefixed with the `not` symbol mean the negation is to be used. For example, the clause for `blameworthy` will be true if the person is `responsible`, the action was *not* `justified`, and the harm was greater than the benefit to the victim.

The MR domain is an especially interesting test bed for A3 because many of the concepts are expressed using the negation of others. In fact, in some cases, there are several layers of negation as seen in the clauses for `accident`. If the clauses for MR had to be written without using negation, the resulting theory would be much more cumbersome to write and more difficult to read. As this domain shows, some concepts are easier for people to express and understand in terms of the negation of other concepts and it therefore seems desirable for a theory revision system to handle such theories.

In order to evaluate the original MR model, Shultz (1990) tested the theory on two large sets of real-world cases of harm-doing. One set was taken from legal cases in English and American law and the other from anthropologists working with traditional cultures that had no codified legal system. The MR model agreed on the outcome of 84% of the legal cases and 97% of the traditional cultures cases. These experiments were intended to demonstrate the validity and generality of the model.

Because the theory did not agree 100% with all of the test cases, one might be tempted to use a theory revision system to repair the theory. However, this assumes the fault lies within the theory. The other alternative is that the test cases themselves may be at fault. First, the MR domain is a general theory of how people decide matters of responsibility. Trying to apply this general theory to specific cases may not always work. For example, in the legal cases, the final decision by which the theory is evaluated is whether the theory agrees with the judicial ruling. Clearly, no judicial ruling is guaranteed always to be right as evidenced by the appeals process where earlier decisions may be overturned. Second, when applying the MR theory to a specific legal case, information about the case must be whittled down and represented by only a small number of attributes. This will almost certainly lead to leaving out relevant information.

The MR domain is an interesting test bed for theory revision systems not so much because judicial rulings may be duplicated but rather as an example of

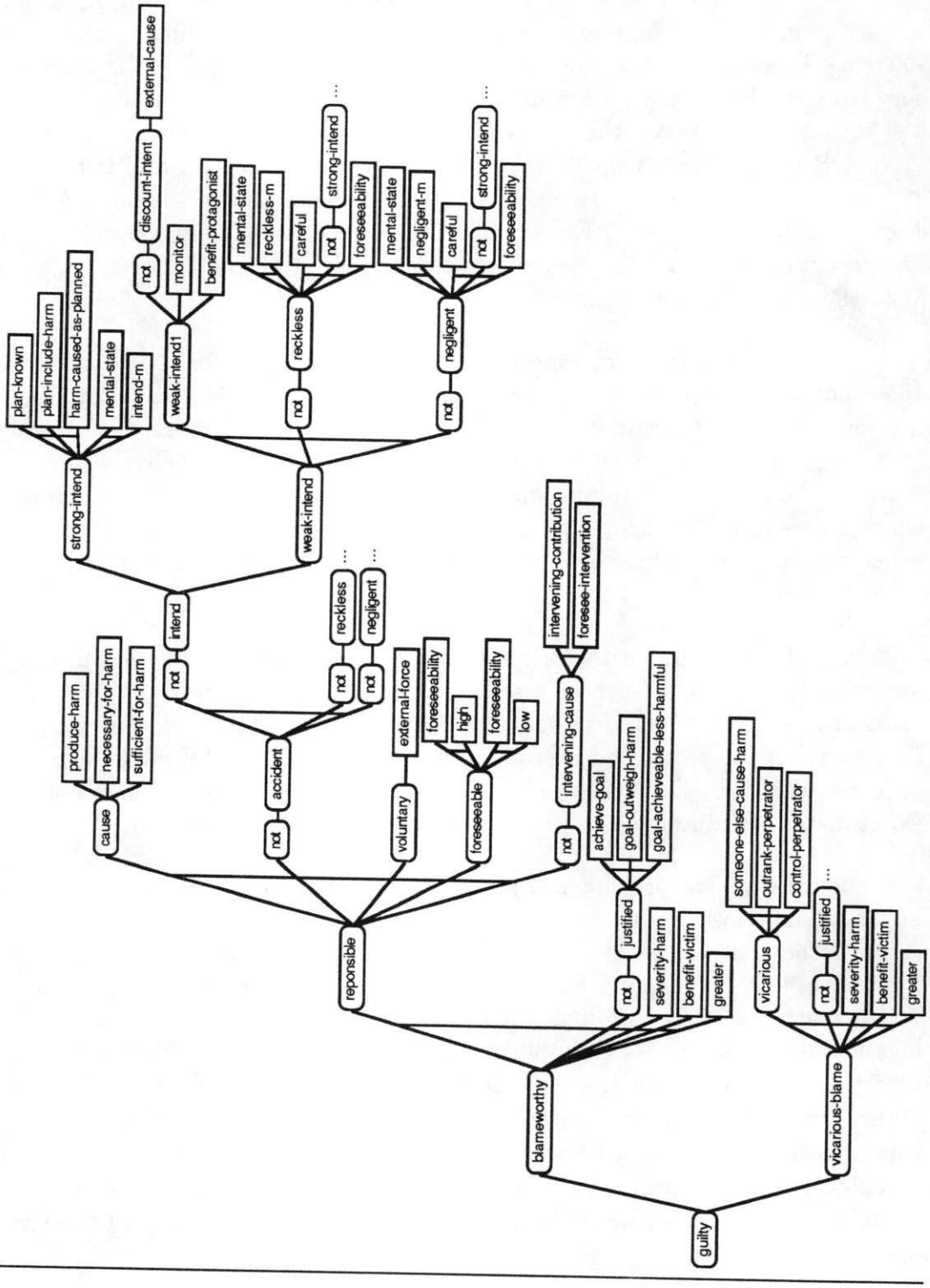

Figure 5.1. Relationship between concepts in the Moral Reasoner theory.

a somewhat complex theory that people use themselves and can understand. The experiments in this section with the MR domain test A3's ability to repair faulty versions of the theory.

Experimental Results

The experiments in this domain involved randomly mutating the correct theory, training it on a number of examples, and measuring the resulting accuracy and distances between the revised theories, the mutated theories, and the correct theory. The training and test examples were generated according to the correct theory.

The mutated theories were generated by applying different numbers of mutation operators to the correct theory. Four basic mutation operators were used: add a random literal to a clause, delete a random literal from a clause, add a random clause to the theory, and delete a random clause from the theory. These operators are discussed in more detail in Section 5.2.3 on page 75. The experiments were run at different levels of mutation in order to evaluate A3's ability to repair corrupt theories. Each trial involved mutating the correct theory by applying 1, 2, 4, 6 and 8 randomly chosen mutation operators and training the system on 100 randomly chosen examples from a possible set of 202. The results are averaged over ten trials for each level of mutation.

The set of 202 examples from which to choose was randomly generated to include 102 positive examples and 100 negative examples. Table 5.4 lists the background predicates used to represent each example along with the possible values for its arguments listed in braces. The set of negative examples was generated by selecting random values for each background predicate and testing the resulting example against the theory. Examples that were not provable by the theory to be `guilty` were retained as negative examples. Approximately 90% of examples generated this way are negative.

The set of positive examples is representative of the different classes occurring in the theory. Analysis of the theory shows that there are seventeen distinct classes representable in terms of the lowest background predicates. This was computed by expanding the definition for `guilty` through the intermediate concepts into a DNF form expressed in terms of the background predicates and a few intermediate-level predicates. A predicate was not expanded if its definition was expressed in terms of background predicates and there were multiple clauses defining the predicate. Also, the negation of a predicate whose definition was expressed in terms of background predicates was also not expanded. These seventeen classes are presented in Section A.2 in Appendix A. Six random positive examples were generated for each of the seventeen classes giving a total of 102 positive examples.

The experiments in this section consisted of varying the number of mutation operators applied to the correct theory and training A3 on the mutated theory with

Table 5.4: Background predicates along with possible argument values used to represent examples in the MR domain.

sufficient_for_harm(*case*, {yes, no})
produce_harm(*case*, {yes, no})
plan_known(*case*, {yes, no})
plan_include_harm(*case*, {yes, no})
someone_else_cause_harm(*case*, {yes, no})
outrank_perpetrator(*case*, {yes, no})
monitor(*case*, {yes, no})
harm_caused_as_planned(*case*, {yes, no})
goal_outweigh_harm(*case*, {yes, no})
goal_achievable_less_harmful(*case*, {yes, no})
foresee_intervention(*case*, {yes, no})
external_cause(*case*, {yes, no})
control_perpetrator(*case*, {yes, no})
benefit_protagonist(*case*, {yes, no})
careful(*case*, {yes, no})
benefit_victim(*case*, *number*)
severity_harm(*case*, *number*)
achieve_goal(*case*, {yes, no})
intervening_contribution(*case*, {yes, no})
foreseeability(*case*, {high, low, no})
external_force(*case*, {yes, no})
mental_state(*case*, {negligent, reckless, intend, n/a})
necessary_for_harm(*case*, {yes, no})

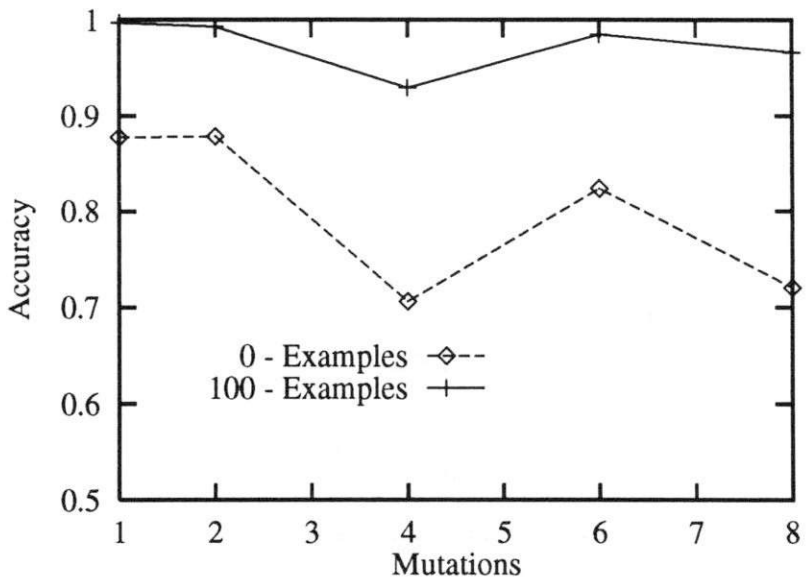

Figure 5.2: Accuracy of revised and initial mutated Moral Reasoner theories for A3.

100 randomly chosen examples using the remaining 102 for the test set. Figure 5.2 presents the accuracy of the revised theories compare to the accuracy of the initial mutated theory. Each point is the average of ten trials.

As one would expect, the accuracy of the mutated theories decreases as the number of mutations to the theory increases. At one mutation, the accuracy is approximately 88% and after eight mutations it is approximately 72%. The accuracies for the revised theories found by A3 after 100 training examples are all above 96% and for four of the five mutation levels the accuracy is above 98%. The shapes of the two accuracy curves in Figure 5.2 are similar which demonstrates that initial theories with lower accuracy are harder for A3 to repair than theories with higher initial accuracy. Overall, these results suggest that A3 is fairly robust with respect to repairing theories containing differing degrees of error.

It is interesting to note that without an initial theory A3 is unable to learn any clauses for this domain. The problem is in the clause induction algorithm used by A3. Because all the background predicates (see Table 5.4) require the introduction of a new variable to be useful, A3 is unable to find a literal to add that has any gain. FOIL would also have trouble with this domain unless it incorporated methods for handling this problem such as determinate literals (Quinlan, 1991) or relational pathfinding (Richards & Mooney, 1992).

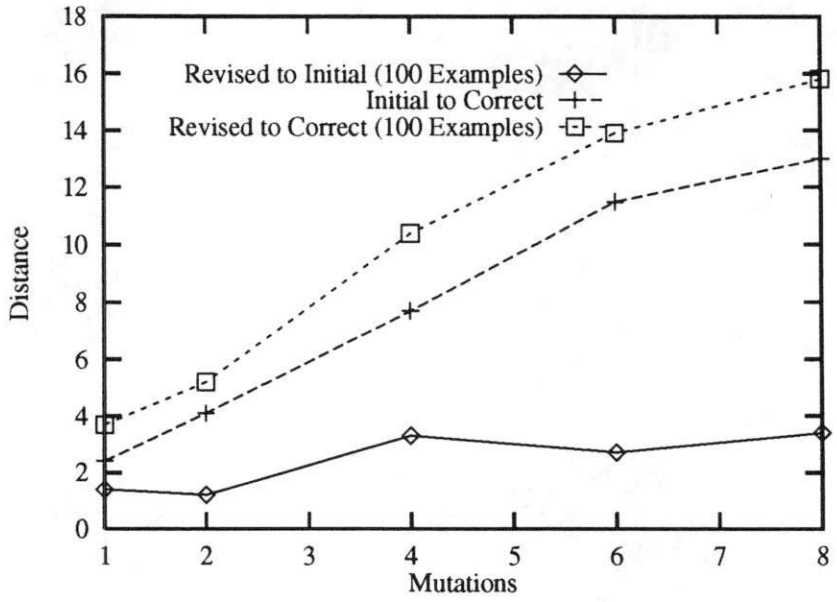

Figure 5.3: Distances between revised, initial mutated, and correct Moral Reasoner theories for A3.

Figure 5.3 presents the distance between the revised theory and the mutated input theory, the distance between the mutated theory and the correct theory, as well as the distance between the revised theory and the correct theory. One would expect the distance between the mutated theory and the correct theory to increase with the number of mutations. However, the distance between the revised and initial mutated theory does not increase nearly as much.

If A3 were fully repairing the errors introduced by the mutations, one would expect this curve to be higher. How does this reconcile with the high levels of accuracy achieved by A3? Clearly, A3 is able to repair the theories with respect to accuracy without many revisions. This could occur if the space of correct or even very accurate theories were large. A large space of accurate theories would also explain why the distance between the revised theory and the correct one is higher than between the initial mutated theory and the correct one. A3 finds repairs to the theory that are different from the mutations but still lead to theories with high accuracy. Another explanation for the small distances between revised and initial theories could be that many of the mutations have no effect on the accuracy of the theory and so A3 finds no need to repair them. However, Figure 5.2 provides evidence against this showing that the accuracy of the mutated theories decreases as the number of mutations increases.

5.2.2 The King-Rook-King Problem

The domain used to perform the mutation experiments in this section is one of the more common ones used for first-order learning systems: the king-rook-king problem (Muggleton, Bain, Hayes-Michie & Michie, 1989; Muggleton & Feng, 1990; Pazzani & Kibler, 1992; Richards, 1992). The goal of the problem is to identify which chess board positions containing a white king, white rook and black king are illegal. A board position is considered illegal if the black king is in check or if any two pieces occupy the same board position.

Table 5.5 gives a correct domain theory for this problem. The concept `illegal` is defined in terms of other concepts: `same_square`, `adjacent_squares`, `line_attack`, `equal`, `less`, and `adj`. The concepts for `equal`, `less`, and `adj` are specified as background knowledge to A3 and are therefore assumed to be correct. The arguments for the different predicates in the theory are interpreted as follows.

- `illegal(Wkr,Wkf,Wrr,Wrf,Bkr,Bkf)` : is true if the board position specified by its arguments is an illegal configuration. The first two arguments specify the position (rank and file) of the white king, the third and fourth arguments specify the position of the white rook and the fifth and sixth arguments specify the position of the black king.
- `same_square(R1,F1,R2,F2)` : is true when the two positions specified by its arguments are the same, i.e., when `R1 = R2` and `F1 = F2`.
- `adjacent_squares(R1,F1,R2,F2)` : is true when the two positions specified by its arguments are adjacent to one another. This predicate is defined to be false if the two positions are the same.
- `line_attack` : is true if the white rook is in the same row or column as the black king and the white king is not blocking the attack.
- `adj(X,Y)` : is true if the rank `X` (or file) is next to the rank (or file) of `Y`.
- `less(X,Y)` : is true if the number `X` is strictly less than the number `Y`.
- `equal(X,Y)` : is true if its two arguments are the same.

Note that the theory in Table 5.5 contains both negated and non-negated definitions of the background predicates. A3 does not need to explicitly represent concepts in this way but this theory was taken from (Richards, 1992) so that A3's performance could be directly compared to FORTE's.

5.2.3 Theory Mutation

The following four basic mutation operators are used in order to corrupt a theory: add a clause, delete a clause, add a literal to a clause, and delete a literal

Table 5.5. A correct theory for the king-rook-king problem.

```
illegal(Wkr,Wkf,Wrr,Wrf,Bkr,Bkf) :-
     same_square(Wkr,Wkf,Wrr,Wrf).
illegal(Wkr,Wkf,Wrr,Wrf,Bkr,Bkf) :-
     same_square(Wkr,Wkf,Bkr,Bkf).
illegal(Wkr,Wkf,Wrr,Wrf,Bkr,Bkf) :-
     same_square(Wrr,Wrf,Bkr,Bkf).
illegal(Wkr,Wkf,Wrr,Wrf,Bkr,Bkf) :-
     adjacent_squares(Wkr,Wkf,Bkr,Bkf).
illegal(Wkr,Wkf,Rank,Wrf,Rank,Bkf) :-
     line_attack(Wkr,Rank,Wkf,Wrf,Bkf).
illegal(Wkr,Wkf,Wrr,File,Bkr,File) :-
     line_attack(Wkf,File,Wkr,Wrr,Bkr).

same_square(A,B,A,B).

adjacent_squares(R1,F1,R2,F2) :- adj(R1,R2), adj(F1,F2).
adjacent_squares(R1,F,R2,F) :- adj(R1,R2).
adjacent_squares(R,F1,R,F2) :- adj(F1,F2).

line_attack(Wk, Others, A,B,C) :- not_equal(Wk,Others).
line_attack(Same,Same,Wk,Wr,Bk) :- less(Wk,Wr), less(Wk,Bk).
line_attack(Same,Same,Wk,Wr,Bk) :- less(Wr,Wk), less(Bk,Wk).

/* background knowledge */

equal(X,X).
not_equal(X,Y) :- not(equal(X,Y)).
adj(X,Y) :- 1 = X-Y.
adj(X,Y) :- 1 = Y-X.
not_adj(X,Y) :- not(adj(X,Y)).
less(X,Y) :- X < Y.
not_less(X,Y) :- X >= Y.
```

from a clause. One could certainly come up with other interesting operators such as replacing a literal with a different one, changing a variable in a clause, misspelling a predicate, etc. In the absence of any *a priori* classification of error types, these four base operators were chosen because all other possible operators can be expressed as sequences of these four. The degree and type of theory mutations are specified by varying which operator or operators to apply and how many times to apply it. In the experiments described in this section, corrupt theories are obtained by applying the mutation operators as follows.

Deleting a clause from a theory is quite straight forward. A theory is divided into two sets of clauses: those known to be correct, often referred to as "background knowledge," and all other clauses. Only non-background clauses may be deleted (or modified by any of the mutation operators). For example, the clause for the predicate `greater` is part of the background knowledge. The delete-clause operator works by randomly deleting a clause that is not part of the background knowledge.

Deleting a literal from a theory is also straight forward. The operator is applied by randomly chosing a clause (that is not part of the background) and then deleting a random literal from the body of the clause.

The operator for adding a literal is somewhat more complicated. Once again, only clauses that are not part of the background may be modified. As with the delete-literal operator, a random clause is chosen to modify. The predicate symbol of the literal to be added is randomly chosen from all the predicates appearing in the theory. A variablization for the predicate is created in accordance with the mode restrictions declared for the predicate. If the variable in position i is an input variable, it is randomly chosen from those variables already appearing in the clause. If the variable is an output variable, it is randomly chosen from the set of all variables appearing in the clause plus one new variable.

A random clause is added to a theory by selecting a random predicate for the head of the clause, adding new variables for each position in the head of the clause, and applying the add-literal operator multiple times. The number of literals added to create the body of the clause is dictated by characteristics of all the clauses in the theory. Statistics are gathered about the probability distribution of the number of literals in the body of a clause for all clauses in the theory. The randomly created clause will have a random number of literals according to this distribution.

5.2.4 Results with A3

The experiments described in this section involved mutating the correct king-rook-king theory (Table 5.5) to varying degrees using different combinations of the mutation operators and measuring A3's performance in terms of accuracy, distance

of revised theory from the mutated theory, and the distance of the revised theory from the correct theory. The independent variables used for these experiments were the number of training examples, the set of available mutation operators, and the number of mutations performed.

The training examples were selected from a random sample of 2000 board positions (about 34% were positive examples of `illegal`). Experiments were performed at 0, 75, and 200 training examples. Accuracy was measured on the remaining examples.

The theory was modified according to sets of available mutation operators. Five different sets of combinations of operators were used corresponding to each operator used alone (four), and all operators being available (one).

The degree of mutation was controlled by applying different numbers of mutation operators randomly selected from the available set. Experiments were performed for 1, 2, 4, and 8 mutations.

The results are averaged over twenty trails for every combination of these independent variables. Each trial differed in the set of randomly chosen training examples. Appendix D contains the average and standard deviation results of the experiments whereas the following sections present them graphically.

Accuracy of Learned Theories

Figures 5.4 – 5.6 show how the degree and type of mutation affects the accuracy of the resulting theory. The axes in these figures that correspond to the set of mutation operators are all sorted according to the accuracy of the learned theories at 200 examples with 8 mutations.

First of all, note that the accuracy of the mutated theories can vary greatly between the different operator sets. As one would expect, the accuracy decreases as the number of mutations increases regardless of the operator set being used. The add-literal operator appears to have the smallest effect and the delete-literal operator the greatest effect. One possible explanation for this has to do with the structure of the theory and characteristics of the domain. The theory has six top-level clauses for the concept `illegal`, each one having only a single literal in the body. These six clauses account for almost half of the thirteen clauses making up the theory (not including background clauses). Given that about 66% of the examples are negative, deleting a literal from one of these six clauses could cause most or all of the negative examples to be misclassified. On the other hand, adding a random literal to one of the six clauses would only affect the small portion of the positive examples covered by the clause. Also, it is possible that the added literal would have no effect on the semantics of the clause if it were always true or equivalent to another literal in the clause.

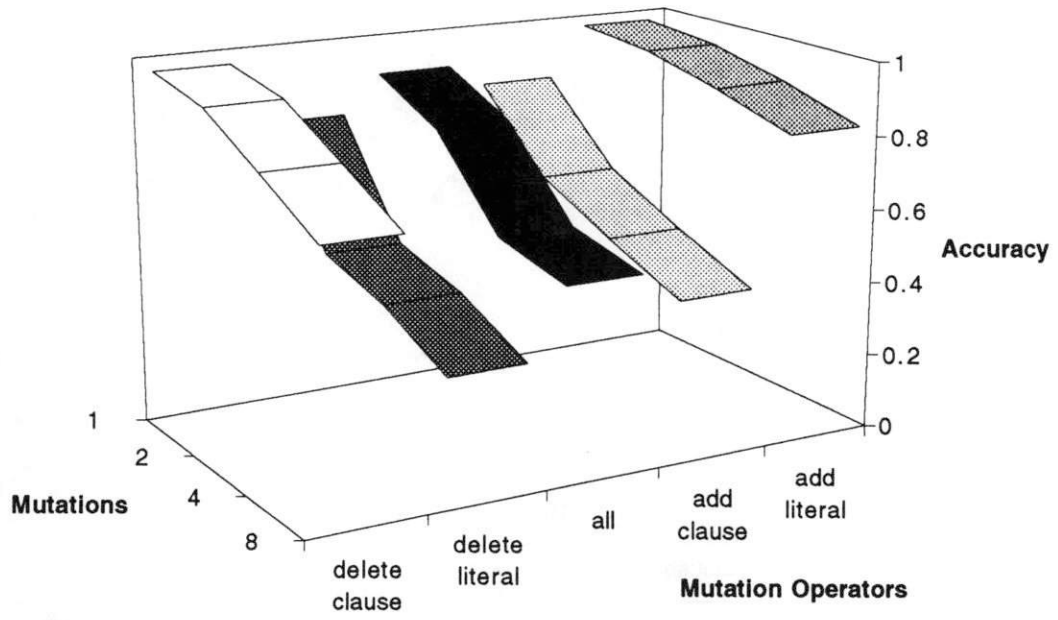

Figure 5.4. Initial accuracy of mutated theories for `illegal`.

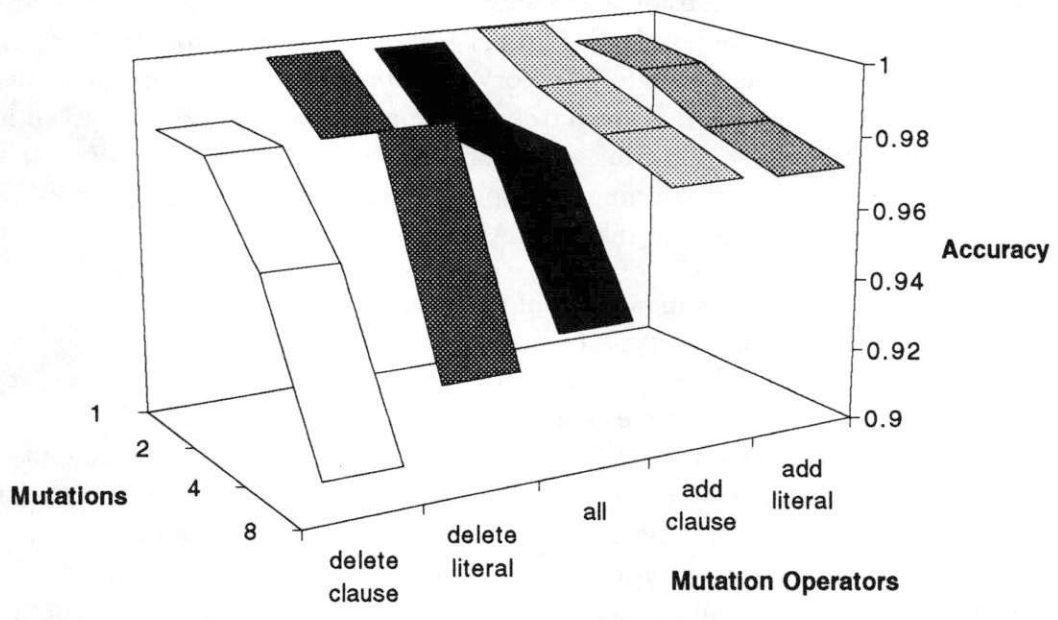

Figure 5.5. Accuracy of A3's revised theories for `illegal` after 75 examples.

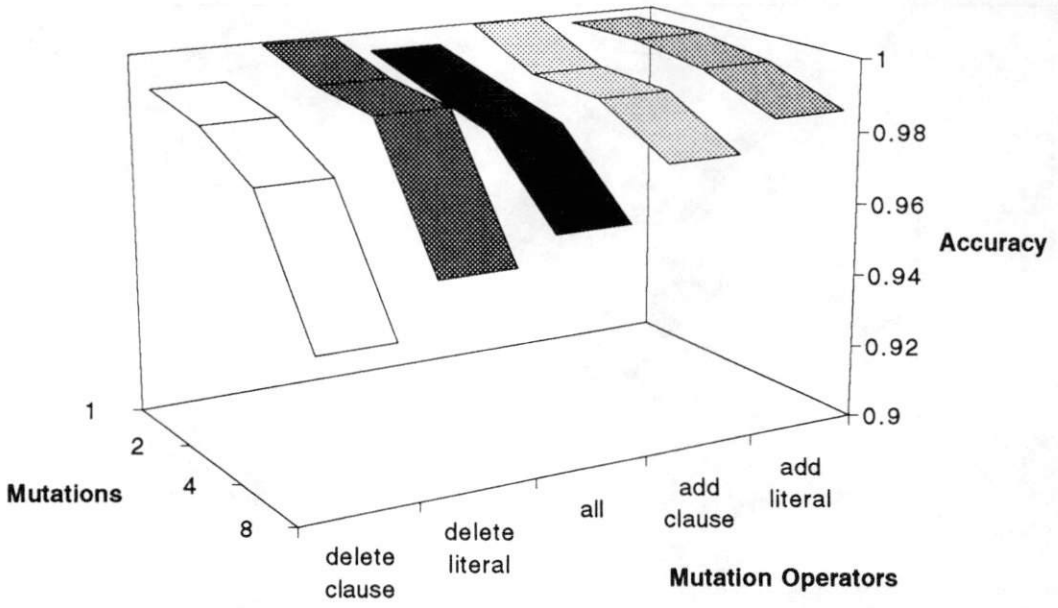

Figure 5.6. Accuracy of A3's revised theories for `illegal` after 200 examples.

Just as the accuracy of the mutated theories decreases as the number of mutations increases (Figure 5.4), the theories learned by A3 also decrease in accuracy as the number of mutations increases for both 75 (Figure 5.5) and 200 (Figure 5.6) examples. Note that the scale for accuracy in Figure 5.4 ranges from 0% to 100% whereas the scale for accuracy in both Figures 5.5 and 5.6 range from 90% to 100%. These results are quite reasonable for a theory revision system, the simple conclusion being that more corrupt theories are harder to repair. Similarly, at any given level of mutation, for any set of mutation operators, the accuracy of the revised theory increases with the number of training examples (the standard learning curve). This is easiest to see by looking at the tables in Appendix D.

Finally, the most interesting aspect of the results on accuracy is how the revised theories vary depending on the type of mutation applied to the correct theory. As stated earlier, the axes for the mutation type are sorted according to the accuracy of the learned theories at 200 examples and 8 mutations. Figures 5.5 and 5.6 show that after 200 examples at 8 mutations, A3's performance varies according to the mutation type with theories mutated by the delete-clause operator being the hardest to repair and those with the add-literal being easiest to repair. As one would expect, A3's performance with all mutation types present falls in the middle with the add-clause and add-literal mutations being easier to repair and the delete-clause and delete-literal mutations being harder to repair.

Note that A3's accuracy after 200 examples is not perfectly correlated with the initial accuracy of the mutated theories. Figure 5.4 shows that after eight mutations

the accuracy of theories produced with the delete-clause operator are much higher than those produced by the add-clause operator. However, after 200 examples, the accuracy of A3's revised theories are much greater for the add-clause operator than for the delete-clause operator. This result indicates that the difficulty of repairing a theory is not so much dependent on the accuracy of the initial theory as it is on the nature of the errors that occur in the theory.

Although there are performance differences depending on the type of mutation applied, those differences are not as extreme as they may appear in Figures 5.5 and 5.6 which are scaled in order to highlight the differences. After 200 examples with eight mutations, the worst performance is for the delete-clause operator which is 94.29% $\pm$2.21 and the best performance is for the add-literal operator which is 98.65% $\pm$1.13.

In order to compare these results with the purely inductive task of learning a definition for `illegal`, A3 was presented with 200 examples and an initial theory containing only the background predicates shown in Table 5.5. Averaged over 20 trials, A3 learned theories that were only 89.83% accurate. This level of accuracy is lower than any of the values shown in Figure 5.6. This result seems to indicate that even having an incorrect initial theory is better than having no initial theory at all. Clearly, A3 must be taking advantage of parts of the initial theory that are correct or at least partially correct in order to produce more accurate theories than it could if the initial theory were not present at all. Although one could likely construct an initial theory that would lead A3 to learn less accurate theories, the results presented in the section illustrate the system's robustness with respect to errors in the initial theory.

Distance of Learned Theories from Initial Theory

Along with measuring the accuracy of the revised theories, the distance metric was used to measure the distance between the revised theory and the initial mutated theory. These results can be seen in Figures 5.8 and 5.9. The graph for the zero examples case is not included because clearly no revision has taken place and all the distances are zero. Just as Figure 5.4 shows the accuracy of the mutated theories, Figure 5.7 shows the distance between the correct and mutated theories.

The curves in Figure 5.7 show that for all types of mutation operators, the distance between the correct theory and the mutated theory grows roughly linearly with the number of mutations. This is to be expected, especially for the add-literal and delete-literal operators which typically change the distance by one each time they are applied. Notice, however, that the curves for the add-clause and delete-clause operators are also roughly linear in the number of mutations but that their slope is steeper. This is due to the fact that adding or deleting a whole clause has a greater effect on the distance measure than does adding or deleting a single literal.

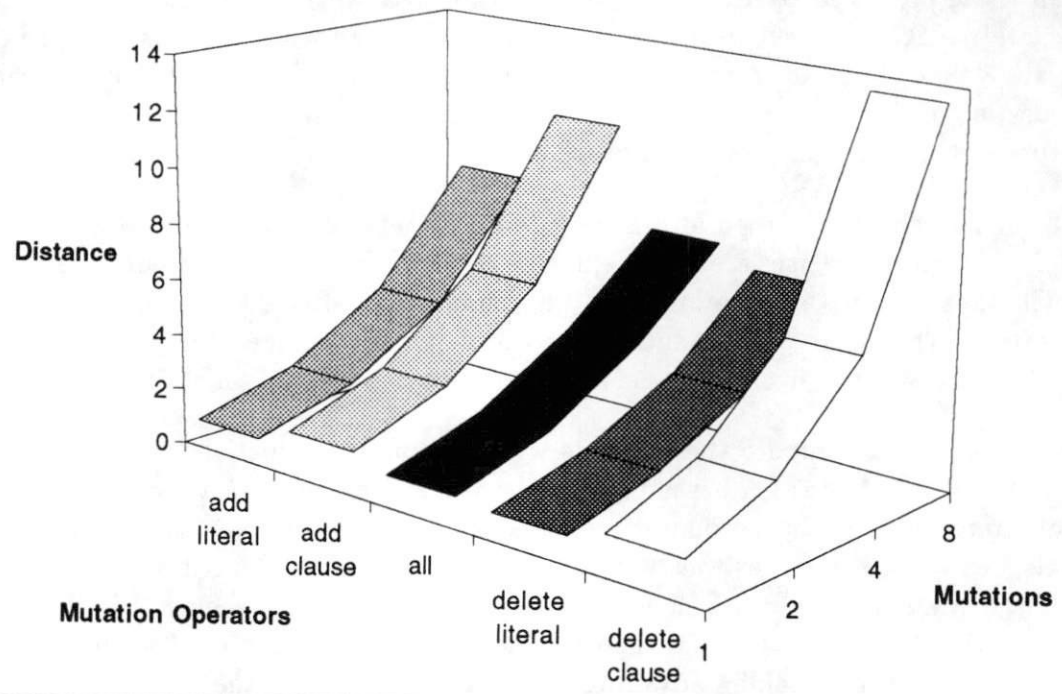

Figure 5.7. Initial distance between mutated and correct theories for `illegal`.

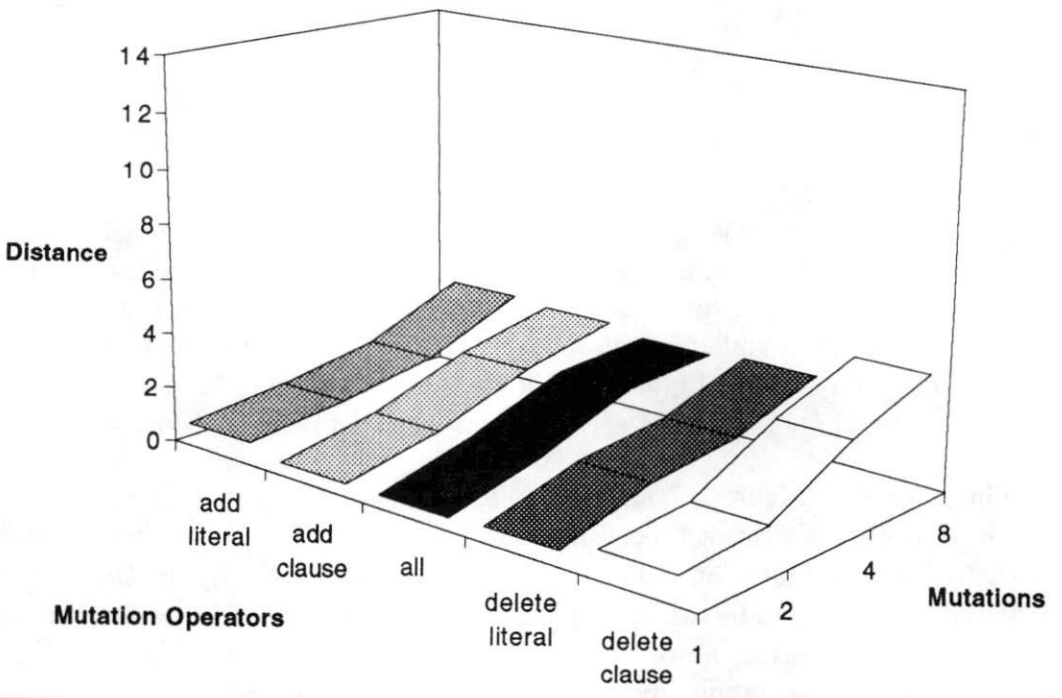

Figure 5.8: Distance between revised and mutated theories for A3 after 75 examples.

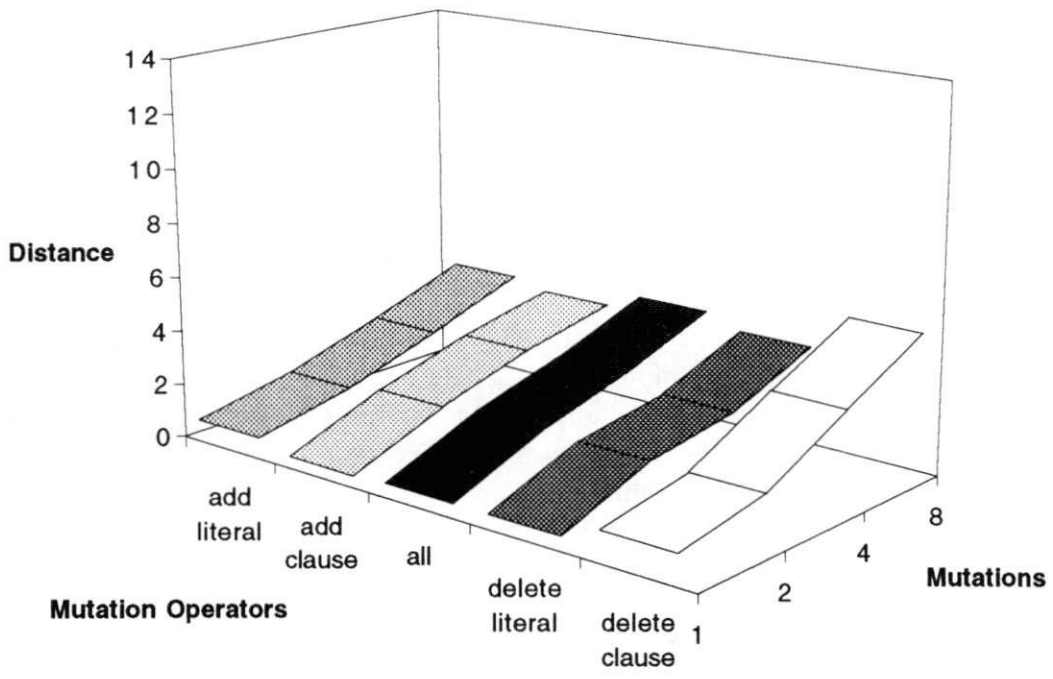

Figure 5.9: Distance between revised and mutated theories for A3 after 200 examples.

The results shown in Figures 5.8 and 5.9 provide a measure of the degree to which theories are being revised by A3. The distance between the mutated theory and the revised theory increases as the number of mutations increases for all classes of mutation operators. One would expect this in light of fact that the mutated theories being revised also increase in distance from the correct theory as the number of mutations increases.

It is also important to note that even though the distance between the revised and mutated theories increases with the number of mutations, this distance is always less than the distance between the mutated and correct theories. A theory revision system should minimally revise an input theory in order to classify the training examples. These experiments use one particular correct theory for the king-rook-king problem, namely the one given in Table 5.5, even though it is possible to come up with other equivalent theories. If somehow the system were to "undo" the mutations that formed the input theory, the distance between the revised and the input theories would be the same as the distance between the input and correct theories. Consequently, the distance between the revised theory and the input theory should not be greater than the distance between the original theory and the correct one. Any theory revision system that produced distances greater than those between the input and correct theories would be doing an unnecessary amount of revision.

Why then are the revised theory distances produced by A3 consistently less than the distance between the mutated and correct theories? Because A3's primary goal is to produce theories that correctly classify the training examples, no revisions will be made to a theory beyond what is needed to cover the examples. Many of the mutations to the theory may have little or no effect on the accuracy of the theory, especially if they are in the form of extra clauses that are never true. Also, the training set may not have examples representative of all the possible disjuncts occurring for the domain. If some of the representative examples of these rare disjuncts are not in the training set, those parts of the theory affecting these examples will not be revised. Of the six clauses for `illegal` shown in Figure 5.5, the first three each cover only about 4.5% of the positive examples whereas the last three each cover roughly 33% positive examples.[3] This could explain, in part, why the distances at 200 examples (Figure 5.9) are slightly greater than those at 75 examples (Figure 5.8) even though the curves in the two figures are similar in form.

Finally, note that while the curves in Figures 5.8 and 5.9 are quite close to one another independent of the mutation operator, two of the mutation operators behave somewhat differently. The distances for the add-clause operator are lower than the rest, especially at larger numbers of mutations. As described earlier, A3 has trouble repairing theories when the error is multiple, incorrect clauses for the same concept. These lower distances indicate that fewer revisions are being made presumably because the extra clauses are not being identified and deleted. The distances for the delete-clause operator are somewhat greater than for the rest of the mutation operators because as Figure 5.7 indicates, the initial theories are further from the original. A greater degree of revision is required to learn a clause that has been deleted than to identify and delete a single extra literal.

Distance of Learned Theories from Correct Theory

The final set of results presented in Figures 5.10–5.11 are for the distance between the revised theories and the correct one given in Table 5.5. Although it was not a stated goal that A3 should learn the "correct" theory, it is interesting to see how the revised theories do in fact compare to the correct one. The striking thing to notice when comparing Figure 5.7 to Figures 5.10 and 5.11 is that for a given type and number of mutations, the number of training examples has almost no effect on the distance between the revised and correct theories. These results alone might lead one to believe little or no revision was taking place. However, the results for accuracy and distance between the revised and input theories clearly show that A3 is doing a reasonable job of revising incorrect theories.

The conclusion to draw from the results in Figures 5.7, 5.10 and 5.11 is that the space of correct or equivalent theories must be large and that while A3 is able to

[3]The total is greater than 100% because more than one clause can cover an example.

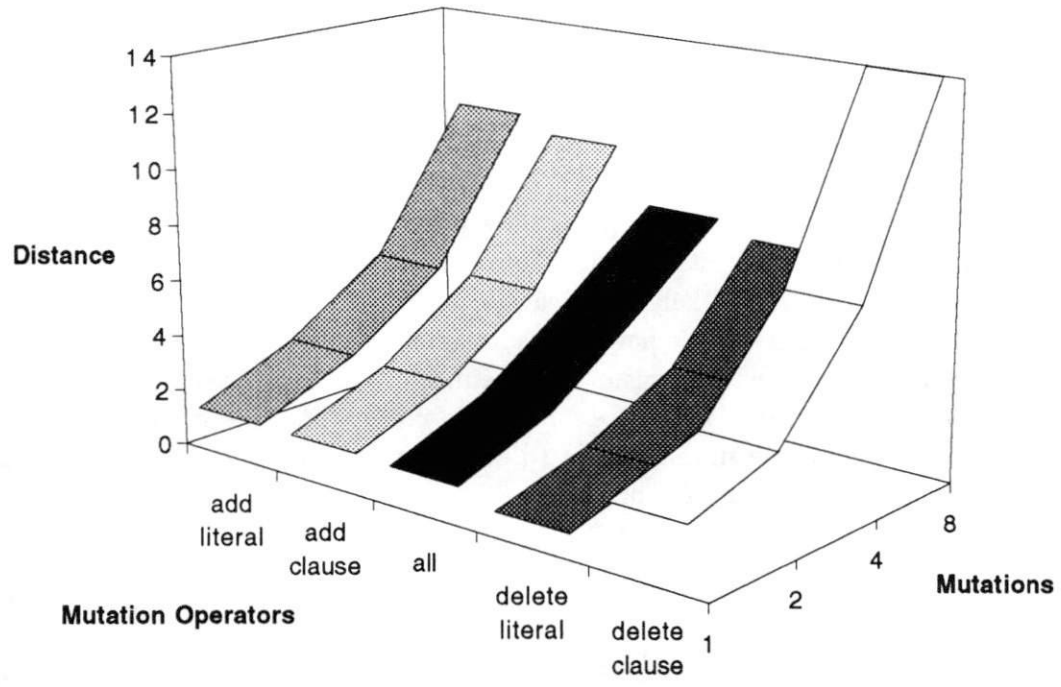

Figure 5.10: Distance between revised and correct theories for A3 after 75 examples.

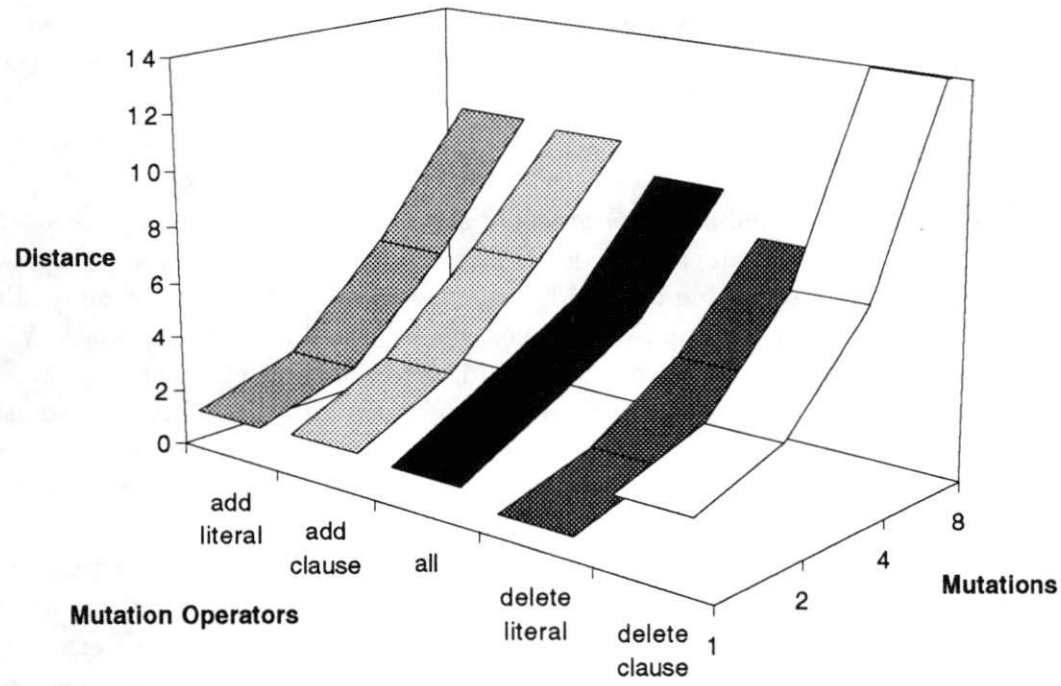

Figure 5.11: Distance between revised and correct theories for A3 after 200 examples.

repair incorrect theories it would have great difficulty in finding the "correct" theory defined as the initial theory that was mutated to form the input theory. This is clearly to be expected given that the space of semantically equivalent but syntactically different theories is so large. For example, what would one expect as the output of a first-order learner that was only given input examples of sorted and unsorted list pairs: bubble-sort?, merge-sort?, quick-sort? Even larger than the number of correct theories is the number of approximately correct theories. For example, in an introductory programming class, students are required to write programs that solve a particular problem. In all likelyhood, each student's solution will be different from all the others and not all will be perfectly correct. Without further information than just the input/output specifications of a procedure or the example/class specification of a concept to be learned, there are simply no constraints on the form a solution may take and it would be unreasonable to expect a theory revision system to come up with any one solution in particular.

5.2.5 Experiments with Intermediate Concepts

Whereas the experiments in the previous sections only presented A3 with examples of the top-level concept to be learned, the experiments in this section investigate A3's performance when given examples of intermediate concepts along with the examples of the top-level concept. It is expected that the additional information provided by intermediate-level examples will help A3 to choose more appropriate revisions that should improve both the accuracy of the revised theories and the distance between the revised and correct theories.

Examples of intermediate-level concepts should help in identifying the location of errors and in making repairs to the correct concepts in a theory because they provide further evidence to the learning algorithm about which concepts need to be repaired. By only giving examples of top-level concepts it is possible that candidate repairs in several different intermediate concepts could have the same effect on the accuracy of the top-level examples and would therefore be indistinguishable from one another. However, providing examples of intermediate concepts should help the decision among such alternatives and favor making the repair to the appropriate concept.

Pazzani and Brunk (1991) presented a domain theory for determining if a student needed to make payments on a student loan. The theory, shown in Figure 5.12, contains sixteen clauses up to four levels deep. Four different types of errors were introduced into the theory: one clause was missing a literal, a whole clause was missing, one clause had an extra literal, and one concept had an extra clause. Figure 5.13 shows the incorrect theory and Appendix B lists the clauses in the correct and incorrect student loan theories.

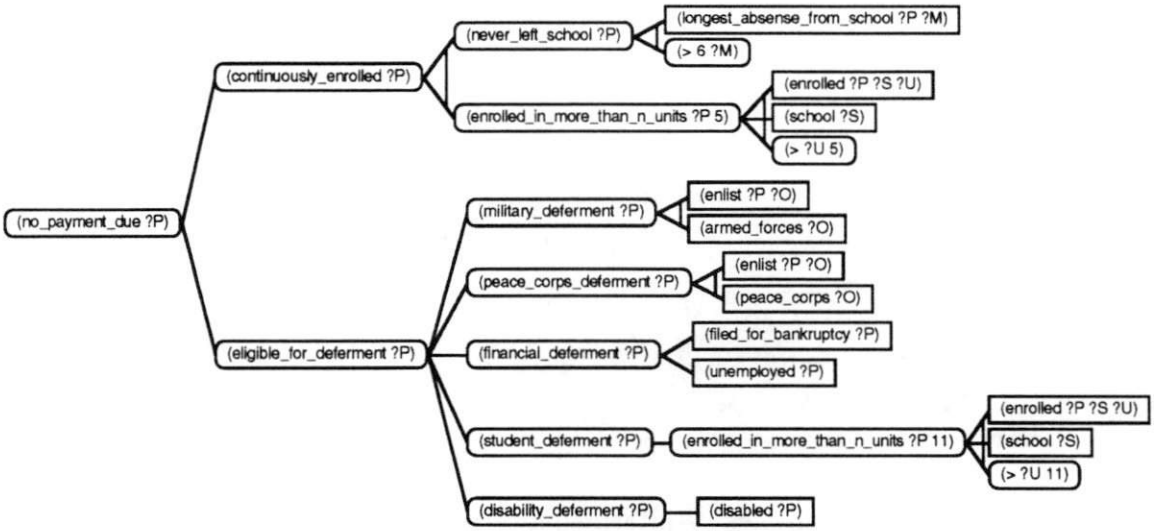

Figure 5.12. Correct theory for the student loan domain.

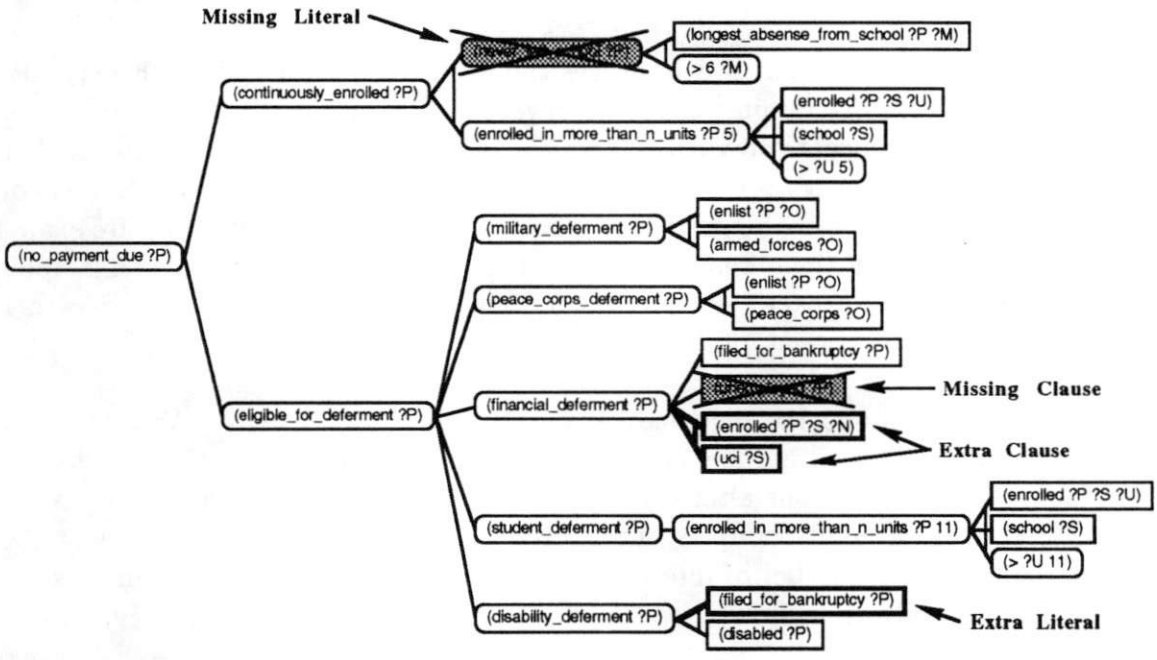

Figure 5.13. Incorrect theory for the student loan domain with four errors.

The following experiment was run using the incorrect theory shown in Figure 5.13. A3 was presented with examples of the top-level concept `no_payment_due` along with a varying number of randomly chosen examples of intermediate concepts from the theory corresponding to each training example. The following eight intermediate concepts were used:

```
continuously_enrolled
never_left_school
eligible_for_deferment
military_deferment
peace_corps_deferment
financial_deferment
student_deferment
disability_deferment
```

The number of examples of intermediate concepts for each top-level example varied among $i = 0$, 2, 6, and 8. The number of top-level examples presented to A3 ranged from 5 to 45 out of a possible 50. For each top-level training example, i randomly chosen intermediate examples were generated for that example. The revised theories were evaluated in terms of accuracy on the unseen top-level examples (Figure 5.14), the distance between the revised and initial theory (Figure 5.15), and the distance between the revised and correct theory (Figure 5.16). The results are averaged over 30 trials.

Figure 5.14 shows that A3 improves the accuracy of the revised theory with increasing numbers of training examples regardless of the number of examples of intermediate-level concepts provided. After 45 examples, for all levels of i (intermediate-level examples), A3 achieves nearly 100% accuracy. However, with fewer numbers of examples, the accuracy of the revised theories increases as more intermediate examples are provided with the difference diminishing as the number of training examples increases. One would expect a higher level of accuracy in this case because the learner is presented with additional information about each training example.

Figure 5.15 shows the distance between the initial and revised theories. A3 performs similarly for all numbers of intermediate-level examples with all reaching a distance of a little over four after training on 45 examples. Although the curves are indistinguishable after 20 examples, one can see that more revision is performed on the initial theory as the number of intermediate examples increases. A more interesting result can be seen in Figure 5.16, which shows the distance between the revised and correct theory.

As the number of top-level and intermediate-level examples increases, A3 learns theories that are increasingly closer to the correct theory. This indicates that the intermediate-level examples are in fact providing useful information to the learner

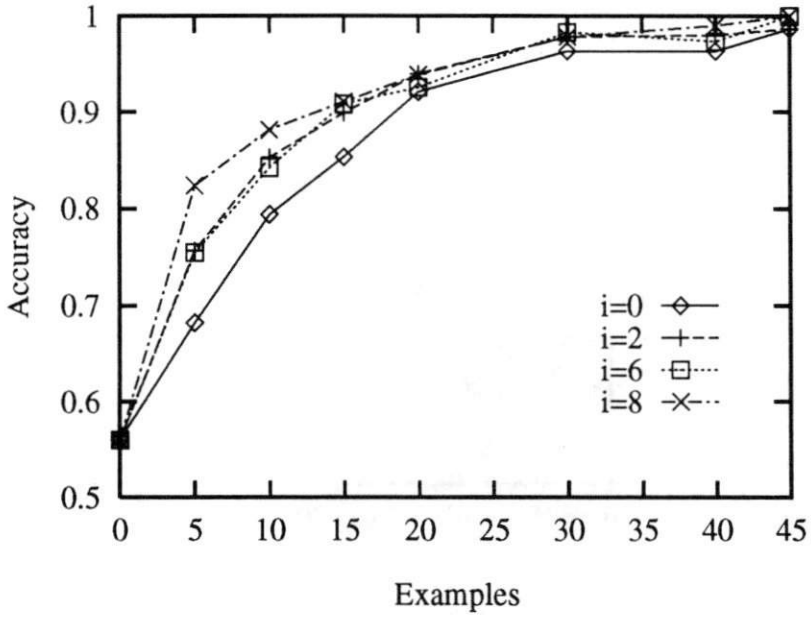

Figure 5.14. Accuracy of A3 when given examples of intermediate concepts.

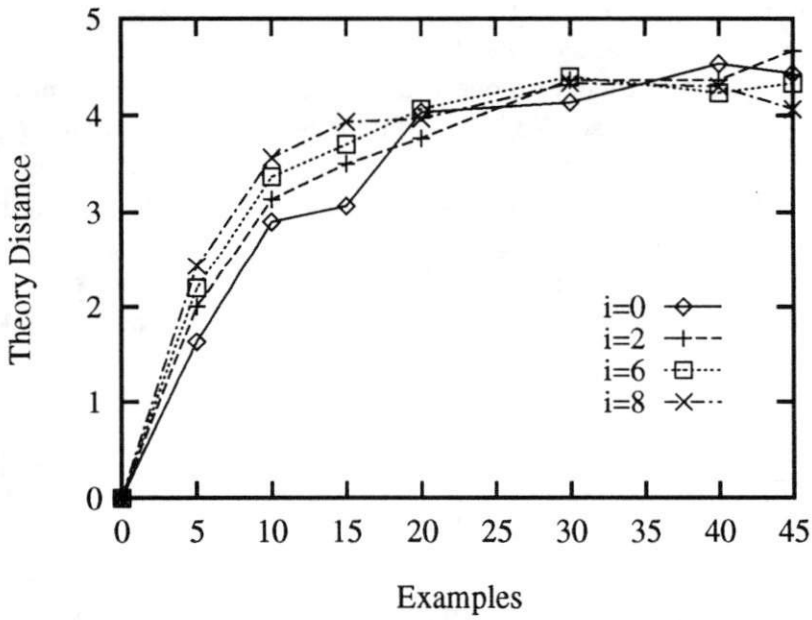

Figure 5.15: Distance between initial and revised theories for A3 when given examples of intermediate concepts for the student loan domain.

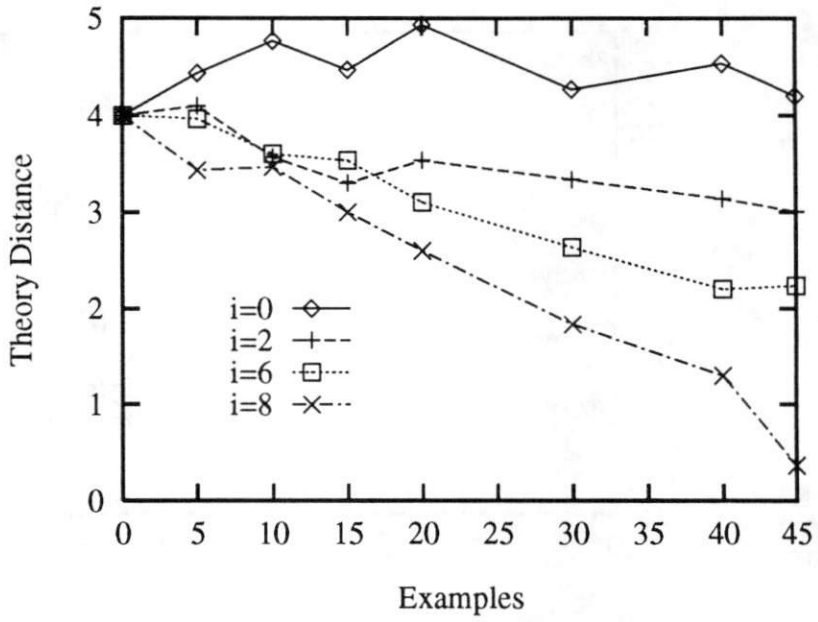

Figure 5.16: Distance between revised and correct theories for A3 when given examples of intermediate concepts for the student loan domain.

about where and how repairs to the theory must be made. When $i = 0$, only top-level examples are presented to A3. Regardless of the number of training examples, A3 revises the initial theory to one that is nearly always a distance of four from the correct theory. In this case, A3 has no guidance as to which changes to make. Figures 5.14 and 5.15 show that when $i = 0$, A3 is able to learn an accurate revision of the initial theory; it just is not able to come up with the correct one. However, when $i = 8$, the distance between the revised and correct theories is dramatically smaller and approaches zero as the number of training examples increases. In this case, A3 is able to make use of the extra information provided by the intermediate-level examples and can correctly locate and repair the errors in the initial theory.

These experiments with intermediate-level concepts demonstrate that A3 can use the additional information provided by intermediate-level examples in order to learn theories that are closer in distance to the correct theory than would be the case if only top-level examples were provided. These experiments also highlight the advantages of A3's simple mechanisms for locating and repairing errors in a theory that do not require designating a single specific concept to be learned. These mechanisms not only treat positive and negative examples in a uniform manner, but also make no special distinction between examples from multiple concepts.

5.3 Discussion

This chapter has demonstrated how A3 handles a variety of different problems relevant to the theory revision task. The system was shown to repair theories containing negation and recursion as well has theories containing multiple errors of different types. Finally, it was shown that providing examples of intermediate concepts can help A3 learn more accurate theories as well as theories that are closer to the correct theory.

The first section of this chapter provided simple demonstrations of how A3 is able to repair theories containing negation and recursion. An example from Shapiro for computing the symmetric difference between two lists was used. Whereas Shapiro's system required the use of an oracle as well as examples of a theory's intended behavior, in contrast, A3 required only the examples. Although A3 was able to repair most of the errors introduced into the `diff` theory, there were two cases that were problematic. One case involved learning the recursive clause for the `member` predicate that required adding a literal in the correct location in the body of the clause. Because A3 only adds literals to the end of a clause, there are cases where the correct literal may not show an improvement if, for instance, the literal bound new variables required by other literals in the clause. One possible solution to the problem would be to try adding literals at all locations in the body of the clause. Of course, a static analysis of the variables in the literal and the clause could reduce this search space by determining which locations are equivalent to which others.

The other problem A3 had with the `diff` theory was the case in which it identified an incorrect concept but could not determine which clause in the concept required specialization. A possible solution to this problem would be to attempt to specialize each clause in turn and select the best revision. However, A3 was designed to choose a single best clause to specialize in order to be more computationally efficient. Specializing clauses requires a search for new literals to add to the body of the clause and evaluating each literal is an expensive process because it involves attempting to prove each example. FOIL's technique is more efficient because partial proof information can be retained whereas A3 must reprove each example for each new literal added. Caching partial proof information in the theorem proving component of A3 is one possible solution but initial experiments with this approach showed that the overhead of caching was approximately equivalent to the savings obtained.

The second half of this chapter reported empirical studies demonstrating A3's performance on a variety of tasks. The first experiment involved mutating a complex theory of moral reasoning that contained multiple levels of negation. It was shown that A3 performed well at repairing these mutated theories even for relatively high numbers of errors. The results for the distance measures of the revised theories proved to be interesting. As the number of mutations to the correct theory rose, A3 was able to repair the theories in terms of accuracy; but, the distance of the revised theories

to the correct one rose as well indicating that the repairs to the theory were in some sense orthogonal to the errors introduced through mutation. The distances between the revised and initial theories provide some clues as to why this is the case. For all levels of mutation, A3 performed small numbers of revisions to the initial theory which led to highly accurate revised theories. In fact, the degree of revision was much less than the degree of mutation. This seems to indicate that for this domain the number of correct or highly accurate theories is very large and that such accurate theories can be found by taking a few revision steps away from the initial mutated theory.

The second set of empirical studies involved introducing different classes and degrees of errors into a correct theory for the king-rook-king problem and measuring A3's ability to repair those theories. The results in this domain showed that A3 was able to repair certain types of errors more easily than others although it was quite competent at handling them all. In fact, for all types and degrees of error, A3 learned more accurate theories than it would without an initial theory at all.

Comparing the initial accuracy of the mutated theories with the accuracy of the theories produced by A3 shows no correlation between the two. This indicates that the correctness of the initial theory has less to do with how well A3 will repair it than does the type of errors occurring in the theory. The experimental results show that A3 more readily repairs theories with extra literals or extra clauses than with missing literals or missing clauses. A possible explanation of this behavior is that the extra literals and clauses are usually identified with the assumption finding mechanism whereas missing literals and clauses must be learned using the inductive component. A3's implementation of the FOIL-like method is very simple and does not have some of the more advanced features such as Quinlan's determinate literals (Quinlan, 1991) or Richard's relational pathfinding mechanism (Richards & Mooney, 1992).

The final empirical study investigated the effect that examples of intermediate concepts had on A3's performance. Previous experiments had demonstrated that the theory revision task is underconstrained with respect to recovering the correct theory from an initially incorrect one. Part of the problem is due to having insufficient information as to where in a theory a repair should be made. In some cases, repairing different intermediate concepts may have an equivalent effect on the accuracy of the top-level examples but be incorrect with respect to the intermediate concepts. The experiments with the student loan domain demonstrate that A3 is able to use the information contained in examples of intermediate concepts effectively in order to recover the correct theory. As more examples of intermediate concepts were presented, the theories learned by A3 were shown to be progressively closer to the correct one. With or without the intermediate-level examples, A3 performed the same amount of revision on the initial theory. However, the intermediate examples were able to guide A3 towards repairs closer to the correct theory.

In summary, A3 has been shown to be a robust learner with respect to the type and degree of errors in the initial theory. Also demonstrated was A3's ability to repair theories containing negation and recursion and its ability to use examples of intermediate-level concepts in order to recover the correct theory from an initially incorrect one.

Chapter 6
Comparison to Other Approaches

This chapter presents other systems that perform first-order theory revision and compares them to A3. Where possible, empirical comparisons between these systems and A3 are made. Also described are systems that perform certain aspects of the first-order theory revision task but which fall short of performing the full task.

6.1 FORTE

FORTE (Richards & Mooney, 1991; Mooney & Richards, 1992; Richards, 1992) is a system that performs first-order theory revision using operators based on propositional theory revision (Ourston & Mooney, 1990) and inductive logic programming (Muggleton & Buntine, 1988). The system accepts as input an initial theory and positive and negative examples of concepts in the theory and revises the theory to cover the input examples while keeping the revised theory as semantically and syntactically similar to the input theory as possible.

Although both FORTE and A3 can revise first-order theories, FORTE does not allow the initial theory to contain negated literals nor can it add negated literals to clauses during the revision process. In this respect, A3 is able to repair and learn a larger class of theories than can FORTE.

6.1.1 System Operation

Table 6.1 outlines the top-level algorithm of FORTE. The search terminates when no operator application can improve the accuracy of the theory. A3's top-level control structure is similar to FORTE's in that both perform a hill-climbing search through the space of theory revisions. However, the systems differ on the number of alternatives tried at each point in the search space. FORTE attempts repairs at all revision points and prunes the list when it is unlikely or impossible for a revision point to lead to a better theory. A3, on the other hand, attempts revisions at a single best

Table 6.1. FORTE's main loop.

```
FORTE(examples, theory)
  let T = ∅
  let T_best = theory
  while T_best better than T do
    begin
      let T = T_best
      let T_best = ∅
      generate revision points for T based on examples
      sort revision points by potential gain
      for each revision point from best to worse do
        if revision potential better than T_best
          then generate revisions
               update T_best
    end
  return T_best
```

point and only tries other alternatives if that revision does not improve accuracy. In this respect, A3 performs less search than FORTE.

FORTE uses an extensive set of theory revision operators, each with a possibly large number of instantiations. FORTE controls the branching factor of the search by restricting operator applications to a set of revision points identified in the theory. There are two basic types of revision points: specialization and generalization. Revision points are identified during the process of trying to prove misclassified examples. In some respects, FORTE's revision points are similar to A3's assumptions. However, FORTE generates many more revision points for an example than A3 finds assumptions.

FORTE defines a *specialization revision point* to be a clause that participated in the proof of a negative example. For each clause in the theory, FORTE records how many negative examples used that clause in a proof of the example. This number represents the *potential* of the revision point which is the maximum increase in accuracy one could expect by specializing the theory at that point.

For example, one of FORTE's operators is to delete a clause from the theory. Without any constraints, this would lead to a large number of revised theories to evaluate. However, FORTE uses the specialization revision points as a guide for restricting the application of this operator by attempting to delete clauses with high potential.

FORTE defines a *generalization revision point* as a specific antecedent literal in a clause, as a clause, or as a predicate. These revision points correspond to the different generalization operators used by FORTE and indicate points of failure in proving positive examples. In trying to prove a misclassified positive example, FORTE notes all goals in the theory that failed and caused the proof process to backtrack. In addition to these failing literals, FORTE keeps track of the literals that bound variables during the proof up to the failure point. These literals form the *antecedent-based* revision points. Every clause containing an antecedent-based revision point becomes a *clause-based* revision point. Finally, a *predicate-based* revision point is created for each distinct predicate from the antecedent-based revision points.

Revision points identify likely places in the theory where errors occur. The following discusses the details of each of FORTE's revision operators and how they use revision points to guide learning.

Delete Rule

Specialization revision points identify clauses in the theory that may contribute to it being overly general. The more drastic method of specializing such a clause is to delete it from the theory. FORTE produces a new candidate theory for each specialization revision point that corresponds to deleting the identified clause. Presumably fewer candidate theories will be created than if all clauses in the theory were deleted one at a time because FORTE only attempts to delete clauses corresponding to specialization revision points.

Delete Antecedent

Clause-based generalization revision points are used to identify clauses in the theory that are overly specific and may have caused some positive examples to be unprovable. Each such revision point is annotated with the positive examples that identified the clause. FORTE produces a new candidate theory for each revision point by generalizing the clause to cover all the failing positives and to cover no negative examples. Clauses are generalized by deleting antecedents and, if necessary, several generalizations of the same clause may be built in order to cover all of the failing positive examples for that clause. FORTE first attempts to generalize the clause by repeatedly deleting antecedents from the clause that allow the most number of positives to be proved without covering any negatives. Once such a generalization of the clause is found, it is added to the new theory and the process is repeated in order to cover any positives not covered by the generalized clause. It is possible that this hill-climbing approach to deleting antecedents will become stuck on a plateau and not find a generalization of the clause. In this case, FORTE then tries an exhaustive search of deleting combinations of multiple literals from the clause. This is clearly

an expensive combinatoric search; but, FORTE can prune the space when it is known that deleting a certain literal will cause some negative examples to be provable.

Add Antecedent

Specialization revision points identify potentially overly general clauses. The delete clause operator is a drastic approach to correcting the problem and suffers from the possibility that some positive examples need the deleted clause. Rather than throw away an overly general clause, the add antecedent operator attempts to specialize the clause so that it is still useful for proving positive examples but prevents any negatives from being misclassified. The add antecedent operator is similar to that of FOIL. It begins with the overly general clause and adds literals until the clause excludes all negative examples. If any positives are left uncovered, the process repeats and builds new clauses to cover the remaining positives. FORTE uses a slightly different evaluation function than FOIL's; it only counts the number of positive and negative examples that are provable and not the number of ways in which each example is provable. As described in Table 2.3, FOIL's evaluation function is:

$$Gain_{FOIL}(L) = T_i^{++} \times (I(T_i) - I(T_{i+1}))$$

whereas FORTE uses:

$$Gain_{FORTE}(L) = T_{i+1}^{+} \times I(T_i) - I(T_{i+1}).$$

If no literal has positive gain and there are still provable negative examples, FORTE calls on its *relational pathfinding* (Richards & Mooney, 1992) component to find new literals to add. This method is intended to overcome the well known problem with FOIL that makes it difficult, if not impossible, to learn certain conjunctive definitions where each literal on its own has no information gain but the conjunction does. After the relational pathfinding component returns, FORTE may again try adding single literals to the clause if some negative examples are still covered.

When adding literals to a clause at an intermediate-level in a theory, FORTE uses the top-level positive and negative examples in order to generate positive and negative examples for the head of the clause being repaired. Once the examples have been reformulated, FORTE runs the basic FOIL algorithm on the clause using the new examples. However, reformulating the top-level examples can be problematic, especially if the clause being repaired generates new bindings for variables. For example, in the following theory the clause for `brother` is overly general and needs to be specialized:

```
uncle(U,N) :- brother(U,X), parent(X,N).
brother(A,B).
parent(sam,joe).
```

However, given the positive example `uncle(fred, joe)`, the generation of examples for the `brother` predicate is problematic because there is no information about how to bind the variable `X` in the clause for `uncle`. Similarly, generating negative examples for a clause is difficult because the clause may not be used in any proof of a positive or negative example and consequently generates no bindings for variables in the head of the clause.

The add-antecedent operator replaces a single clause in the theory with one or more clauses that are specializations of the original clause. The new theory is therefore specialized with respect to the input examples but may be more general than the original theory because new clauses may have been added.

Add Rule

Clause-based generalization revision points are used to identify clauses that may be overly specific and require generalization in order to allow some positive examples to be proved. The add-rule operator is really a combination of several other operator applications. The identified clause does not allow the proof of some number of positive examples and the goal is to modify the clause to cover the positives while excluding all negative examples. The operator first copies the clause then deletes all literals from the clause that do not allow any negatives to be covered. At this point, if any positives remain uncovered, FORTE will continue to delete literals from the clause until all positives are covered, even if that means some negatives will now be included. Finally, the add-antecedent operator is applied to the new clause in order to exclude all negative examples that may have been included by deleting literals. Note that the add-rule operator may add several clauses to the theory because the add-antecedent operator can generate multiple clauses.

Identification

Predicate-based revision points indicate that a theory is too specific and may require a new clause in order to prove a failing literal. Whereas clause-based generalization revision points indicate particular clauses that need to be generalized, predicate-based revision points indicate that an altogether new clause is needed. The idea behind the identification operator was presented in CIGOL (Muggleton & Buntine, 1988) and is based on inverse resolution. Suppose the following clauses are in a theory:

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

Notice that the two clauses for a share some structure. Identification will create a new clause and rewrite the theory to be:

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

The definition of a remains semantically the same but the definition of x has been generalized by adding the new clause `x :- c, d.` FORTE generates candidate theory revisions based on all places in the theory where identification creates a new clause for the failing predicate defined by the revision point.

Absorption

The absorption operator is the complement of the identification operator and its application is driven by antecedent-based revision points. Absorption is also based on inverse resolution (Muggleton & Buntine, 1988). Suppose the following two clauses are in a theory:

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

Notice that the clause for tt a shares structure with the clause for x. Absorption will rewrite the clause for a giving:

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

The definition of x remains semantically the same but the definition of a has been generalized because it may use definitions for x other than `x :- c, d.` FORTE attempts to apply the absorption operator to the literals in clauses identified by the revision points. There may be many ways to apply the absorption operator to a failing literal and FORTE will generate all such applications as candidate theory revisions.

Note that both the absorption and identification operators may compact the theory as measured by number of literals. Because FORTE uses an evaluation function that compares theories based on their improvement in accuracy and breaks ties according to theory size, it is possible that these operators may be applied even if no improvement in accuracy results. Once the theory has been compacted, it may then be possible for the other operators to improve upon the theory accuracy. Because

these operators change the syntactic form of the theory, they will also cause an increase in the distance measure between the revised and input theories.

6.1.2 Comparison to A3

Although many aspects of FORTE and A3 are similar, there are some important differences between the two systems. Broadly, these differences are in how errors are located, how they are repaired, and the types of errors that can be repaired.

Locating Errors

Both FORTE and A3 base their revision of a theory on identifying the location of errors in the theory. This is the role of FORTE's revision points and A3's assumptions. In general, FORTE will generate many more revision points for an incorrectly classified example than will A3.

A provable negative example will cause FORTE to generate a revision point for each clause used in all proofs of the example. If a negative example has only one proof, FORTE and A3 will generate the same number of revision points and assumptions. Note, however, that FORTE will attempt to revise the theory based on each revision point whereas A3 will choose a single best revision point to use. Consequently, A3 has a smaller space to search than FORTE when trying to revise a theory. The smaller search space could be detrimental to accuracy but experiments indicate that this is not the case.

If there are multiple proofs of a negative example, FORTE will generate more revision points than A3 and will consequently have an even larger search space. A clause used in the proof of a negative example may be useless to revise if there are other proofs that circumvent that clause. A3, on the other hand, only generates assumptions that are known to be potentially worthwhile. In this case, A3's search space is smaller than FORTE's; but, A3 has eliminated branches that are known *a priori* to be useless.

The other type of revision point generated by FORTE is the generalization revision point for unprovable positive examples. These revision points correspond to all failed literals in the proof of the example as well as "contributing points" which are literals that bind variables used by each failing literal. A3 finds a subset of these literals for its assumptions. A failing literal will only be considered for a revision point if the success of that literal would have lead to the correct classification of the positive example. FORTE will generate a revision point for a literal even if the successful matching of the literal would still not lead to the positive example being provable. Again, A3's search space of revision points is much smaller than FORTE's. Using

assumptions, A3 is able to prune the search space of revisions to avoid attempting repairs to a theory that could not lead to an improvement in accuracy.

Although FORTE performs more search because it attempts repairs at multiple revision points, it does have the advantage that it uses an admissible search heuristic (Nilsson, 1980) over the list of possible revision points. The revision points are ordered by their potential improvement in accuracy. Once the potential improvement of a revision point is less than the accuracy of the best revision found, the system can stop searching the list. This approach could not be used in A3 or other theory revision systems that handled negation because one can not determine an upper bound on the impact of different types of repairs on the classification of the training examples. Consequently, it is not possible to determine the potential improvement of a revision point.

Revision Operators

Both FORTE and A3 use sets of operators to revise theories that are built upon the basic four: add a literal, delete a literal, add a clause, and delete a clause. FORTE, however, uses two other operators that are not used by A3: identification and absorption.

The delete-clause operator is the most straight forward one to apply and is used by both systems to remove a clause that appears to be incorrect. The only difference between the systems with respect to this operator is the conditions under which the operator is applied.

The add-literal operator is used in both systems to specialize an overly general clause rather than delete it. Both systems add literals based on techniques derived from FOIL. However, FORTE uses an additional method, relational pathfinding, to improve upon the plateau problem caused by FOIL's hill-climbing method. A3 could easily be modified to take advantage of such improvements over the basic FOIL method but the emphasis of the research to date has been on the general problem of theory revision.

The main distinction between A3 and FORTE's add-literal operator is that A3 does not have to reformulate the top-level examples into examples of the intermediate-level clause being revised. A3 evaluates the addition of a single literal by measuring the accuracy of the revised theory with respect to the input examples. FORTE on the other hand must generate intermediate-level examples which can be problematic. Also, because FORTE only evaluates the candidate literals to add with respect to the clause being revised and the reformulated examples, it may select a literal that causes the accuracy of the theory taken as a whole to decrease. FORTE's add-literal operator is consequently more efficient because A3 must use theorem proving to

evaluate candidate literals. However, FORTE gains this efficiency at the expense of potentially producing incorrect repairs.

The add-clause operator is fairly different between the two systems. Whereas A3 adds a new clause to a concept found to be too specific, FORTE copies existing clauses found through the clause-based generalization points. FORTE attempts to generalize those clauses by applying an elaborate search mechanism combining both delete-literal and add-literal operators. A3 only modifies existing clauses if it can identify an error in the clause in which case it need not perform the extensive search that FORTE applies in its add-clause operator.

The application of the delete-literal operator is much different in A3 than in FORTE. The clause-based generalization points found by FORTE indicate that a clause must be generalized. FORTE then performs a search in the space of literal deletions from the clause until the failing example can be proved. If the resulting theory is too general, FORTE will attempt to specialize the clause using the add-literal operator. A3, one the other hand, only deletes literals it has identified as being incorrect and does not require the search that FORTE does. As with FORTE, the clause may be specialized if deleting the literal caused the clause to become overly general.

The identification and absorption operators are used to change the syntactic form of the theory rather than improve system accuracy. Consequently, these operators will cause the resulting theories to be further in distance from the original without a corresponding gain in accuracy. In part, these operators transform theories that FORTE is unable to revise into ones that it can repair.

Negation

FORTE repairs theories based on the revision points found for incorrectly classified examples. These revision points identify the type and location of potential errors. The error type indicates whether the theory is overly general or overly specific. The application of theory revision operators is based on this type information. However, if the theory to be repaired has errors within the scope of a negation, FORTE will not be able to identify the type or location of the actual error. For example, consider the following over-general theory for odd numbers:

```
odd(X) :- not(even(X)).
even(2).
```

The incorrectly classified negative example `not(odd(4))` leads FORTE to generate a specialization revision point corresponding to each clause that matched in the proof of the example. In this case, the only clause that matched was the clause for

Table 6.2: Average accuracy and distance measures for A3 and FORTE on mutated theories for `illegal` after 50 and 250 examples.

	A3			FORTE		
	Accuracy	Distance		Accuracy	Distance	
Examples		Initial	Correct		Initial	Correct
50	**95.09**	**2.75**	**5.90**	**96.06**	**3.10**	**5.50**
250	**98.89**	**3.83**	**6.22**	**99.85**	**4.33**	**6.50**

`odd`. Consequently, FORTE will only attempt to revise this clause to repair the theory even though this is not where the error occurs. Now, consider the overly specific theory for odd numbers:

```
odd(X) :- not(even(X)).
even(X).
```

The incorrectly classified positive example `odd(3)` leads FORTE to generate generalization revision points that correspond to failing literals in the theory. The only failing literal in this case is `not(even(X))`. This revision point leads to revisions of the clause for `odd` which, again, is not where the theory is wrong.

6.1.3 Empirical Comparison of A3 and FORTE

This section compares the performance of A3 and FORTE on a number of mutated theories for the king-rook-king problem. Each trial consisted of mutating the correct theory for `illegal` (Table 5.5) by introducing one of each type of error: an extra clause, a missing clause, an extra literal, and a missing literal. Both systems were given the same initial theory to revise and the same training and test sets of examples. Training sets of size 50 and 250 out of 2000 were used with the remaining examples used for the test set. Table 6.2 presents the accuracy and distance measures for both A3 and FORTE after seeing 50 and 250 examples averaged over 20 and 18 trials respectively.

After 50 examples, FORTE outperforms A3 in terms of accuracy and A3 outperforms FORTE in terms of the degree of revision performed on the initial theory. FORTE's theories are on average about 1% more accurate than A3's but A3's theories are approximately 11% closer to the initial theory than are FORTE's.

Although FORTE learns theories that are slightly closer to the correct theory than A3, this measure is less important to the theory revision task than is accuracy and degree of revision. Without additional information such as examples of intermediate-level concepts, the theory revision task in underconstrained and one

cannot expect a theory revision system to approximate a correct theory in terms of syntactic distance. At 50 examples, FORTE learns theories that are about 7% closer to the correct theory than does A3.

The results for 250 examples are similar to those for 50 examples. Both systems learn more accurate theories at 250 examples than at 50 and FORTE's theories are still an average of 1% more accurate than A3's. Both systems perform more revision of the initial theory at 250 examples as indicated by the average distance between the learned and initial theories. At 250 examples A3's theories are on average about 12% closer to the initial theory than FORTE's. A3 also shows a slight improvement of about 4% over FORTE for the distance between the learned and correct theories.

It is difficult to draw any strong conclusions about the overall relative performance of A3 and FORTE on the basis of these experiments. However, both systems are able to repair incorrect theories achieving nearly the same levels of accuracy with A3 learning theories that are slightly closer to the initial theory than does FORTE. That A3 learns more minimally revised theories should not be too surprising because it uses the distance metric in its evaluation function for choosing which revision to make. FORTE, on the other hand, uses accuracy and the size of the revised theory in its evaluation function.

The theories used in this section to compare A3 and FORTE did not include errors within the scope of a negation which is problematic for FORTE but is easily handled by A3. To get a better overall comparison of these systems would involve comparing them on more domains and with different types of errors. It would also be interesting to compare both systems' ability to revise different types of errors in isolation as was done for A3 in Section 5.2.

6.2 FOCL and KR-FOCL

6.2.1 Theory Guided Induction

FOCL (Pazzani & Kibler, 1992) is a first-order learning system that combines both empirical and explanation-based learning (EBL) (Mitchell, Keller & Kedar-Cabelli, 1986) to take advantage of existing background knowledge even when that knowledge is incomplete or incorrect. FOCL performs theory guided induction rather than theory revision, the difference being that while both types of learning rely on an initial theory, FOCL outputs a new concept definition rather a revision of the input theory. KR-FOCL (Pazzani & Brunk, 1991) is an extension of FOCL that uses information gathered during the learning process to suggest to the user possible revisions to the initial theory. KR-FOCL can be viewed as a theory revision system that requires some decisions to be made by the user.

FOCL

FOIL is a purely inductive learning system that does not take advantage of existing background knowledge. Explanation-based learning methods use background knowledge to constrain the search for operational concept descriptions. FOCL was designed to combine these two learning methods so that the inductive component could take advantage of background knowledge when present and so that the explanation-based component could use induction when the initial theory was incomplete or incorrect.

FOCL's basic control structure is identical to FOIL's but differs on how new literals are chosen to add to the clause under construction. FOCL divides predicates in the domain theory into two classes: operational and non-operational. Operational predicates are those that have no clause in the theory with which to prove them; they must be present in the examples. Non-operational predicates are those that do have corresponding clauses in the theory. As with EBL methods, FOCL learns concept descriptions in terms of operational predicates only. When learning a new clause, FOCL first computes the information gain of the target concept if the background knowledge supplies a definition for it. If the gain is positive, FOCL operationalizes the target concept to form a new clause. If the clause still covers some negative examples, FOCL continues to add new literals to the clause as would FOIL. The new literals are variablizations of both operational and non-operational predicates. If a non-operation predicate has the highest information gain, FOCL will operationalize the predicate adding the resulting literals to the body of the clause.

FOCL's operationalization process differs from the standard EBL process which works with only a single example at a time. FOCL uses the available set of examples to guide the operationalization process. For example, take the following very simple theory where `a` is the target concept, `b` is non-operational and `c`, `d`, `e`, and `f` are operational predicates:

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

Given the positive example $\{c \wedge e \wedge f\}$, standard EBL would form the concept `a :- e, f, c` corresponding to the predicates in the leaves of the proof tree for the example. FOCL, on the other hand, has a set of positive and negative examples that it uses to guide the operationalization process. As each predicate is operationalized, FOCL computes the information gain for each alternative clause and expands the one with the highest gain. For example, using the above theory and the positive example $\{d \wedge e \wedge f \wedge h\}$ and the negative example $\{c \wedge e \wedge f \wedge j\}$, FOCL will expand the second clause for `a` because the first does not cover the positive example but does cover the

negative and so has negative gain. The final concept found by FOCL in this case would be `a :- e, f, d.`

FOCL has an advantage over FOIL because it can use an initial theory to guide the learning process, making use of the correct portions of the theory and avoiding incorrect portions. FOCL can also learn new clauses to cover the incomplete portions of the theory. Taking advantage of the correct or useful portions of a theory and identifying and avoiding the incorrect portions is an important component of any theory revision system. However, FOCL learns concept descriptions in terms of operational predicates and loses the structure of the initial theory. In this respect, FOCL fails at the theory revision task. KR-FOCL was designed to overcome this deficiency and uses the results of FOCL to suggest repairs to the initial theory.

KR-FOCL

KR-FOCL (Pazzani & Brunk, 1991) was designed to identify and suggest repairs for the four different types of errors: a missing clause, and extra clause, a clause with extra literals and a clause with too few literals. The system uses information about how FOCL learned the concept description in order to propose revisions to the initial theory that correct one or more of these errors. During the learning process, KR-FOCL notes which clauses were operationalized and which predicates were induced. This information is used in conjunction with several heuristics to suggest where and how the theory may be repaired.

The first heuristic states that a clause learned exclusively by operationalization indicates that all of the clauses operationalized were correct and require no modification.

The second heuristic states that a clause learned by first operationalization and then by adding induced predicates indicates that one or more of the operationalized clauses are too general and require adding the induced literals. Because the induced predicates could be added to any of the operationalized clauses and still result in an equivalent definition, the system asks the user to which clause they should be added.

The third heuristic deals with clauses that were learned exclusively by induction. In this case, there are three classes of possible repairs. The new clause could subsume an existing clause in the theory that was never operationalized because it was too specific and had too low information gain. Replacing the existing clause with the induced one has the effect of deleting literals from the existing clause. If the new clause is subsumed by an existing clause in the theory, it may be that the existing clause is too general and admits too many negative examples. Replacing the existing clause with the induced one has the effect of adding literals to the existing clause. Finally, the induced clause could be an alternative clause for either the top-level

predicate or any other unoperationalized predicate in the theory. KR-FOCL applies these heuristics and queries the user if any of the suggested revisions should be made.

The final heuristic states that any clause in the original theory that was not operationalized is a candidate for deletion. Such a clause has no function in the theory; presuming the examples are thoroughly representative of the domain, it may be safe to delete the clause. Again, the user is asked to confirm the suggested revision.

6.2.2 Comparison of A3, FOCL and KR-FOCL

The main difference between A3, FOCL, and KR-FOCL is that A3 performs theory revision whereas the others perform theory guided induction. KR-FOCL comes closest, however, in that it suggests to the user what revisions might be useful. One could imagine modifying KR-FOCL to make the revisions without user intervention and retain those revisions that improve accuracy the most. Such a modification would make KR-FOCL much more similar to A3 and FORTE in that it would perform a hill-climbing search through the space of revised theories guided by the proposed repairs and the resulting accuracy. Because this modification has not been made to KR-FOCL and because it has not undergone extensive experimental tests, there are no empirical results by which to compare A3 and KR-FOCL and comparison is limited to a discussion of the differences between the two algorithms.

To a large extent, the revision operators in A3 and KR-FOCL are similar. Both systems add and delete clauses as well as add and delete literals from clauses. The main difference lies in the conditions under which these operators are applied. A3 first identifies the location and type of error it believes occurs in the theory. Based on this information, A3 applies the appropriate operators to repair the error.

In A3, clauses to be deleted are identified as those contributing to the incorrect classification of an example. In KR-FOCL, clauses are considered for deletion if they were never operationalized, in which case no positive examples required them and they possibly covered some negative examples. Whereas A3 directly identifies clauses to be deleted, KR-FOCL must make this decision after all other learning has taken place. Only at this point do unused clauses become candidates for deletion.

One heuristic by which KR-FOCL can add literals to a clause is when a learned concept required adding induced literals. The user decides on which clause to add them. Because the induced literals to be added are based only on the examples that make it through a single operationalization path, it may be harder to tell to which clause to add them. In fact, the literals themselves might have been chosen differently if the other contexts (i.e., operationalizations) in which they would be used in the revised theory had been available. By deciding ahead of time which clause needs the

addition of new literals, A3 is able to measure the effect of adding literals to that clause with respect to the full theory and the whole example set.

Similarly, using the clause subsumption heuristics, KR-FOCL might have added or deleted different literals from the clause had the operator been performed in the context of the full theory and not just within the context of a single operationalization path.

Another problem for KR-FOCL arises when different revisions, adding or deleting literals, are made to different operationalizations of the same clause. It is not at all clear how this information could be used to decide what repairs are needed for the original clause.

Finally, KR-FOCL operates in two phases: it first uses FOCL to learn an operational concept and second suggests revisions to the theory based on how the concept was learned. This could lead to a great deal more work for the learner if the theory contained a single deep error that required FOCL to add induced literals to all of its operationalized concepts. This extra work could be avoided perhaps if the error was repaired early on in the learning process.

6.2.3 Empirical Comparison of A3 and FOCL

This section compares A3 to FOCL to emphasize the difference between revising a theory and using a theory to guide induction. Pazzani and Brunk (1991) presented a domain theory for determining if a student needed to make payments on a student loan. The theory, as previously shown in Figure 5.12 (Section 5.2.5), contains sixteen clauses up to four levels deep. Four different types of errors were introduced into the theory: one clause was missing a literal, a whole clause was missing, one clause had an extra literal, and one concept had an extra clause. Figure 5.13 in Section 5.2.5 shows the incorrect theory.

KR-FOCL operated on this incorrect theory suggesting different revisions to correct the different errors. Empirical results comparing A3 and KR-FOCL are not given because KR-FOCL requires user input to decide what revisions to make. Although FOCL does not perform theory revision, it does use an initial theory as a guide for learning. FOCL learns first-order Horn clause theories for the top-level concept being learned and these theories are compared to those learned by A3 for the domain.

The experiment consisted of giving A3 and FOCL the initial incorrect theory and examples chosen from a pool of 50 students. For each different number of input examples, the revised theories were measured in terms of accuracy on unseen examples, distance from the original theory, and distance from the correct theory. The results are shown in Figures 6.1, 6.2, and 6.3 and are averaged over 30 trials.

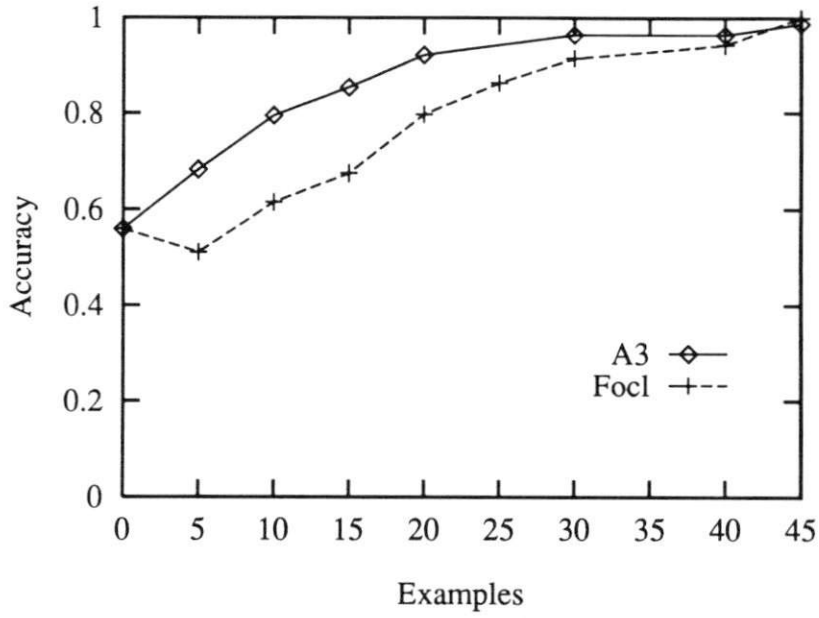

Figure 6.1. Accuracy for A3 and FOCL on student loan domain.

Figure 6.1 shows that A3 performs better overall on accuracy than FOCL and that the distances from both the original and correct theories for FOCL are much greater than for A3. The distance of the revised theory from the original as a function of the number examples (Figure 6.2) rises for A3 from 0 to around 20 examples after which it levels off. A "good" revised theory should be accurate and be close to the original theory. However, early on the theory is not very accurate but the distance to the original theory is small. After 45 examples its distance from the original has increased. In fact, according to the distance metric, the correct, 100% accurate theory given by Pazzani and Brunk (1991) should have a distance of four from the original theory, while the theory learned by A3 has an average distance of about four and a half after 45 examples.

As a comparison to a pure inductive learner, A3 was given 45 examples from this domain along with an empty initial theory. Averaged over 20 trials, A3 learned theories that were 78% accurate. This is much lower than either A3 or FOCL when given an initial incorrect theory. Part of the reason the accuracy with no initial theory is so low is that some of the correct clauses require adding literals with no information gain. Were A3 to incorporate methods such as determinate literals or relational pathfinding, the resulting accuracy would likely increase.

The results comparing the accuracy of FOCL and A3 indicate another benefit of the bias towards minimal revisions of theories. A3 does not delete any of the

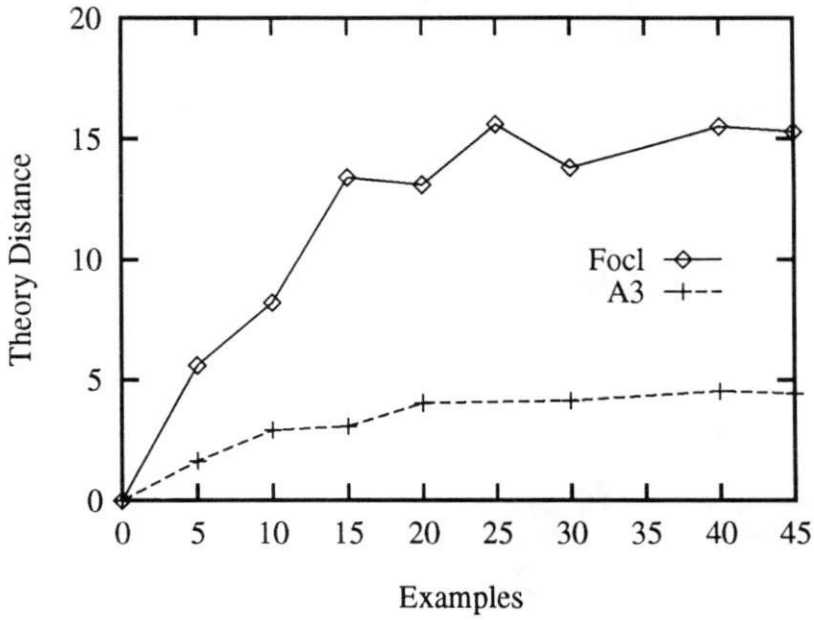

Figure 6.2: Distance between original and revised theory for A3 and FOCL on the student loan domain.

domain theory unless doing so improves accuracy on the training set. In contrast, FOCL learns clauses that discriminate positive from negative examples. When there are few training examples, it ignores those portions of the domain theory that are not needed to explain any of the positive examples in the training set.

Figure 6.3 shows the distance between the revised and correct theories. Notice that the original theory has a distance of four from the correct theory. As learning proceeds, the distance between the revised and correct theory actually increases. One would hope that this distance would, in fact, go to zero as accuracy approaches 100%. Unfortunately, for this problem, A3's distance from the correct theory is an average of 4.2 after 45 examples. In this case, the theory revision problem is under-constrained. For example, the correct theory should contain the following clauses:

```
continuously_enrolled(S) :-
        never_left_school(S),
        part_time_student(S).

part_time_student(S) :-
        enrolled(S,School,U),
        school(School), U > 5.
```

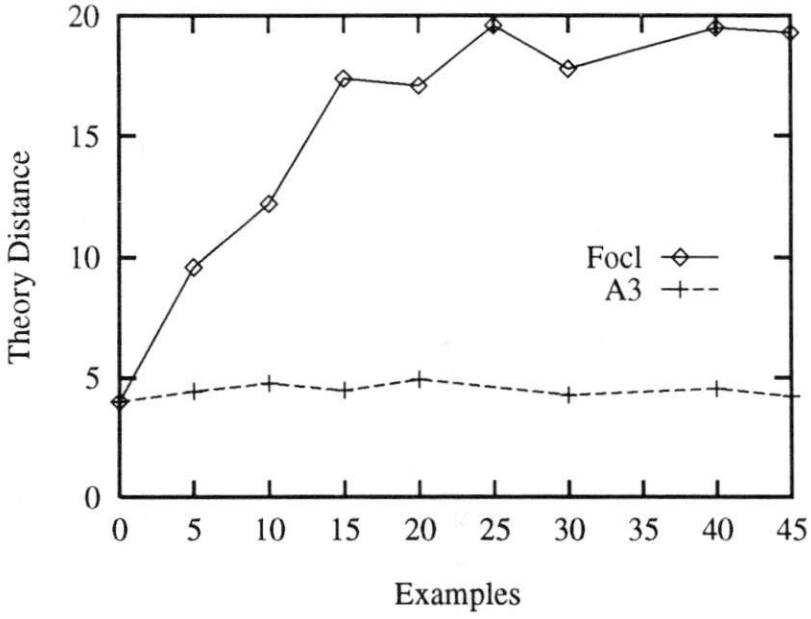

Figure 6.3: Distance between correct and revised theory for A3 and FOCL on the student loan domain.

While A3 typically learns:

```
continuously_enrolled(S) :-
        part_time_student(S).

part_time_student(S) :-
        enrolled(S,School,U),
        school(School), U > 5,
        never_left_school(S).
```

Although these two definitions are equivalent with respect to accuracy, the latter definition does not reflect a proper definition of part-time student that an expert would be willing to accept. It is exactly this type of difference that the distance metric is intended to capture. The problem arises in this domain because there are two places where one could add a literal to achieve the same accuracy. While both are equally distant from the original theory, one makes the revised theory closer to the known correct theory, while the other moves it further away. This problem would not occur if the syntactic differences between the theories resulted in differences in accuracy. This could be accomplished in A3 by providing examples for intermediate concepts (e.g., examples of part-time students), or if the predicate `part_time_student` were used elsewhere in the theory as demonstrated in Section 5.2.5.

6.3 MIS

Shapiro's MIS system (Shapiro, 1983) can inductively learn and correct logic programs given an initial program, examples of how it should behave, and an oracle to answer queries. In this respect, it performs theory revision. However, the system was designed in part as a program debugging aid and requires extensive use of an oracle, or user, to supply answers to queries about the intended meaning of the target program.

MIS was designed to be a general purpose solution to program debugging. It uses an abstract computational model that covers a class of common functional programming languages including Prolog. The algorithms are presented as Prolog code and the programs being debugged are represented as Prolog programs. This provides a direct mapping between a domain theory and what Shapiro calls a program.

Shapiro considers MIS to be an incremental learning system because it processes a single example at a time correcting the theory based on the example. However, the more common interpretation of incremental learning is that the learning system does no or little re-processing of past examples. Although MIS processes a single example at a time, the system is non-incremental in the sense that all of the past examples must be available in order to evaluate the correctness of proposed theory revisions.

6.3.1 Identifying Errors in a Theory

As with A3, FORTE, and other theory revision systems, MIS operates in two phases: error detection and error correction. In the error detection phase, MIS identifies three classes of errors: a program can terminate producing incorrect output, it can terminate failing to produce any output, or it can fail to terminate. The first case corresponds to over-generality where an example is implied by the theory even though it should not be. The second case corresponds to a theory being overly specific, in which case the theory fails to prove an example. The final case of non-termination of programs is one which A3 does not address.

Incorrect Output: Overly General Theories

A theory produces an incorrect output if it is able to prove a negative example, or, in other words, if it is overly general. It must therefore be the case that some subconcept used in the proof of the example produces an incorrect output. The goal of the bug identification phase is to find a goal that incorrectly proved to be true. MIS does this by essentially querying the user about every goal proved during the proof of the example. A bug, or error, is found when the user identifies a goal that

should not have been provable. In the worst case, this algorithm may query the user about every goal solved in the proof of the example which is $O(n)$ where n is the number of goals solved in the proof of the example.

Shapiro presents an improvement on this algorithm that only requires at worst $O(b \log n)$ queries where b is the branching factor of the proof tree and n is again the number of nodes in the proof tree. The basic idea is a divide-and-conquer approach that orders the queries according to the structure of the proof tree. By asking queries that isolate the fault to one half of the tree or another, the number of queries required becomes proportional to the number of branches at a node times the depth of the tree or $O(b \log n)$. The important thing to note about this algorithm is that it searches for a single error in the theory at a time based on a single example even if there may be multiple errors. The user is required to identify incorrect goals in the theory and must therefore have complete knowledge of the intended meaning of every concept in the theory.

Finite Failure: Overly Specific Theories

The other type of error that MIS detects is finite-failure in which a positive example is not provable by the theory. In this case the theory is overly specific. A theory that can not correctly classify a positive example must have some subconcept within the theory that is incorrect. MIS identifies failing goals by once again querying the user about every attempted goal in the proof process. However, in this case, MIS is trying to identify a goal that should be provable but is not; so, instead of querying the user after the goal was proved, it first queries the user for a solution to the goal. If the user supplied instantiation of the goal is not provable by the theory, an error has been found. If the goal is provable, MIS continues attempting the proof of the misclassified example. The worst case number of queries performed by this algorithm will be $O(bd)$ where d is the maximum depth of a proof for any example and b is the maximum number of solutions for any subgoal in the proof of any example.

Error Detection in A3 and MIS

Both A3 and MIS are able to diagnose theories containing multiple errors even though they both look for a single error at a time. MIS entirely relies on information from the oracle to determine whether something is an error or not. The algorithms for locating errors only serve to direct the queries and attempt to reduce the number of queries needed. A3, on the other hand, does not rely on an oracle but rather replaces it with a proof process using assumptions.

A3's proposed assumptions correspond to oracle queries in MIS; however, the answer to the query is decided based on whether or not the example can be proved

using the assumption. This is a more strict requirement for identifying errors because it is possible that multiple errors interact in which case A3 might not identify an assumption that truly corresponds to an error. Because MIS has complete information from the oracle, it need not worry about how errors interact; it simply returns the first error it finds in a theory.

MIS handles negation in much the same way as A3. The algorithms for detecting over-generality and speciality are interchanged when passing through a negated literal. Thus MIS has no difficulty with correctly identifying errors within the scope of a negation. However, the system still relies on an oracle to furnish the intended meaning of all concepts in the theory.

6.3.2 Correcting Errors in a Theory

MIS's approach to both inductive program synthesis and theory revision is to process examples one at a time and incrementally refine the theory to classify all previously seen examples. The system is incremental in that once a theory has been proposed and eliminated, it will never be proposed again. Shapiro then is able to show that MIS can correctly learn or revise a theory in the limit. The approach involves enumerating all possible theories and marking those that are inconsistent with the input examples.

The error diagnosis phase identifies a single incorrect clause in the theory. Shapiro describes two possible approaches to repairing the clause. One is to throw away the clause and learn a new one from scratch. The other method is to identify an equivalence class of common programming errors and to refine the incorrect clause by applying operators that transform clauses into others in the equivalence class. If the system is unable to find a new clause, it asks the user to supply the answer.

MIS's search for clauses to correct an error is based on the notion of a refinement graph. This graph organizes clauses according to generality with the empty clause at the root and more specific clauses at the intermediate nodes. Refinements are applied to nodes in the graph to produce new, more specific clauses. Essentially, these refinements add new literals to the body of the clause. Subtrees of the graph are pruned once a contradicting example is found. The refinement graph is analogous to the G set in version spaces (Mitchell, 1978). The approach that A3 and FOIL take is to find a node in this graph by heuristically chosing the refinement with the best information gain or coverage of the examples. While this approach may fail to find a new clause consistent with the examples, it has the advantage of exploring a much smaller search space which is especially large for first-order theories (Haussler, 1987).

6.4 Other Related Systems

This section describes other learning systems that perform different aspects of first-order theory revision and compares them to A3.

6.4.1 FOIL

The inductive learning component of A3 is largely based on FOIL. Induction is used in A3 to either specialize an existing clause by adding new literals or by learning new clauses from scratch. Because A3 performs induction within the context of a larger theory, its control structure must be different than the one used for FOIL (Section 4.6).

A3 evaluates the addition of each new literal in terms of the improvement to the overall accuracy on the training data. In contrast, because each additional clause added by FOIL strictly generalizes the concept being built, once a positive example as been covered by a clause, there is no further need to check if it remains covered. However, because A3 may have to revise a theory containing negation, it can not assume that once a positive example has been covered that it will remain covered. It must, therefore, evaluate each new literal with respect to the entire training data. This can actually be detrimental to A3's performance as seen in the following example.

For one trial of 75 examples of `illegal` from the king-rook-king domain, FOIL produced the following concept description which is about 96% accurate:

```
illegal(V0,V1,V2,V3,V4,V5) :- equal_row(V4,V2).
illegal(V0,V1,V2,V3,V4,V5) :- equal_column(V5,V3).
illegal(V0,V1,V2,V3,V4,V5) :-
          equal_row(V4,V0), near_column(V5,V1).
illegal(V0,V1,V2,V3,V4,V5) :-
          near_row(V4,V0), near_column(V5,V1).
illegal(V0,V1,V2,V3,V4,V5) :-
          equal_column(V5,V1), not(less(V0,V4)).
illegal(V0,V1,V2,V3,V4,V5) :-
          equal_row(V2,V0), equal_column(V3,V1).
```

A3, on the other hand, given no initial theory and the same example set will only learn the following clauses which are about 90% accurate:

```
illegal(V0,V1,V2,V3,V4,V5) :- equal_row(V4,V2).
illegal(V0,V1,V2,V3,V4,V5) :- equal_column(V5,V3).
```

The reason that A3 is not able to learn the complete definition that FOIL does is due to evaluating the revised theory in terms of the overall accuracy on the training data. A3 is able to learn the first clause in the manner that FOIL does because it improves the accuracy of the empty theory from about 66% to 79% by covering some positives but excluding most of the negatives. The second clause can also be learned by A3 because it only requires adding a single literal in order to improve the accuracy to about 90%. At this point, however, A3 is unable to learn the other clauses that provide the additional 5% improvement in accuracy. The problem is that each of these other clauses requires the addition of two literals. To learn a third clause A3 must go through an intermediate step in which the clause has only one of the two correct literals. This overly general clause then makes the theory overly general and consequently perform worse than a theory containing just the first two clauses. Therefore A3 will not find a literal to add to the third clause that can improve the theory's accuracy and will stop only having learned the first two clauses. On other trials of 75 examples, A3 was able to learn a more complete concept description that included more of the clauses learned by FOIL. However, this general problem still can arise and is dependent on the distribution of examples in the training set.

Further research is required to overcome this problem in A3 when the theory to be revised contains negation. If the theory does not contain negation, A3 could be modified to adopt the control structure of FOIL in which previously covered positives are not checked when adding a new clause or specializing an existing clause.

6.4.2 EITHER

EITHER (Ourston & Mooney, 1990) is a theory revision system that only handles propositional domains. The representational power of propositional theories is greatly reduced because they do not allow recursive definitions and can only perform classification tasks such as deciding if an input example, say `odd(4)`, is true or not rather than being able to generate results such as determining a binding for `X` in the example `member(X, [1,2,3])`.

As with other theory revision systems, EITHER revises theories in two phases. First, the locations of errors in the theory are identified and second, the theory is repaired at the identified locations. EITHER identifies errors in an overly-specific theory by constructing all partial proofs of all failing positive examples. Each partial proof contains a set of assumptions (literals assumed to be true) that are required to prove the example. EITHER then searches for a minimal set of assumptions that cover the most number of failing positive examples. The theory is then generalized to cover the failing positive examples by generalizing all clauses that use an assumption. The clause is generalized by first removing the assumed literal. If this leads to any negative examples being covered, EITHER inductively adds new literals to the clause to exclude any covered negatives.

Overly general theories require finding overly general clauses. EITHER uses the proof of all incorrectly classified negative examples to find a minimal set of clauses at the leaves of the proof tree that can be deleted and will prevent the proof of negative examples. If deleting a clause causes some positive examples to become unprovable, EITHER inductively learns new clauses.

Because EITHER works only on propositional theories it can afford to search for a larger set of assumptions. In contrast, all first-order theory revision systems including A3, FORTE, and MIS only identify a single error at a time. However, EITHER searches for assumptions at the leaves of the theory and may not find simpler, intermediate-level repairs. Also, EITHER only allows negation of literals at the leaves of the proof tree and so can make the simplifying assumption that an uncovered positive example is the result of a missing or overly-specific clause or clauses and that a provable negative example is the result of an overly-general clause or clauses.

6.4.3 RLA

Tangkitvanich et al. (1992) present a theory revision system called $\mathcal{RLA}$ whose basic approach to theory revision is very similar to KR-FOCL's. $\mathcal{RLA}$ first operationalizes the theory based on the input examples. The operationalization differs somewhat from FOCL's; but, it does use information-gain to guide the process. The operationalization phase may delete literals from a clause if they are found to be useless which they define to be a literal whose gain falls below some predefined threshold. If the operationalized theory fails to cover some positive examples, new clauses are learned using FOIL. Concepts in the original theory are then repaired based on information from the operationalization phase.

Both $\mathcal{RLA}$ and KR-FOCL operationalize incorrect theories and then run an inductive process to find either new clauses or literals to add to existing clauses. The problem for these systems is that they do not explain how to operationalize through a negated goal and hence have not demonstrated an ability to detect or correct errors within a negation. Other problems arise when the initial theory contains recursive concept definitions. Also, if a clause is operationalized more than once and each operationalization is repaired differently using induction, these systems have no way of deciding which repair to make to the original clause.

6.4.4 CLINT

MIS is able to identify errors within negation. However, both the error detection and error correction phases require user input to make important decisions to guide these phases. CLINT (De Raedt & Bruynooghe, 1989; De Raedt, 1991) was designed

as an interactive concept learner and is based in part on MIS. It accepts incorrect first-order theories as input and requires both examples of concepts in the theory and the use of an oracle to repair the theory. CLINT uses methods for detecting errors that are based in large part on those found in MIS. Because CLINT does not accept theories containing negation, it repairs overly general theories by deleting clauses and overly specific theories by adding new clauses. New clauses are learned by using a specific to general search of the space of clauses. The initial most specific clause is constructed in a manner common to many systems based on performing least-general generalization (e.g., GOLEM), which is to add all or a large subset of ground literals known to be true over terms present in the examples. It is not clear if or how CLINT is able to repair clauses for intermediate concepts in the theory.

Although CLINT is based in part on MIS, it does not accept theories containing negation. De Raedt and Bruynooghe (1990) describe an extension to the form of input theories to handle negation where both a concept and its negation must be explicitly defined. The system then learns for each concept a set of clauses identifying the positive examples of the concept and a set of clauses defining the negation. This extension has advantages in some learning contexts because examples can be classified as positive, negative, unknown, and inconsistent. However, revising theories such as those in Table 4.3 remains problematic because the initial theory must have clauses that define both the `member` and `not_member` relations which would be difficult to express. Consequently, the system would have to perform the additional overhead of checking the consistency of the two definitions.

6.4.5 GOLEM

GOLEM (Muggleton & Feng, 1990) is a system that induces logic programs from examples and background knowledge. While the system can make use of an initial background theory, it can only construct new theories rather than revise incorrect ones. GOLEM takes the opposite approach from many other first-order learners in that it constructs new clauses through generalization rather than specialization by forming least-general generalizations of very specific clauses that represent single examples. Each example can be represented by a clause whose body contains all literals known to be true of terms in the example. These literals are those provable using bounded computation. The specific clauses for examples can then be generalized to form clauses in the final theory. Overly general clauses are eliminated if they cover negative examples.

Learning clauses containing negation are difficult for GOLEM in part because the initial clauses would by exceedingly large if all true negations were included (e.g., `not(member(1, []))`, `not(member(2, []))`, `not(equal(1, 2))` ...). Bain and Muggleton (1991) describe a method called Closed-World Specialization (CWS) that has been used in both CIGOL and GOLEM for handling both overly general theories

and noise in the input examples (Srinivasan, Muggleton & Bain, 1992). The method invents new predicates that are used as negated literals in clauses under construction. Although CWS provides a powerful method of specializing theories, it has not been shown how concepts in an initial theory used within the scope of negation can be corrected when they are either too specific or too general.

6.4.6 LATEX

Tangkitvanich and Shimura (1993) present a theory revision system, LATEX, that handles noisy training data. Their work builds upon the minimum description length principle (MDL) (Rissanen, 1978). Rather than choosing revisions according to the accuracy on the training data, LATEX measures the amount of compression of the training data afforded by the revision. This compression is measured as the number of bit required to encode the positive examples according to revised theory minus the bits required to encode the positive examples according to the initial theory minus the bits required to encode the revisions themselves. Revisions that provide the greatest compression are favored.

The revision algorithm in LATEX is fairly simple. The initial theory is not allowed to have intermediate level concepts and only one top-level predicate is allowed. This is essentially the class of concepts that FOIL can represent. Given an initial theory, LATEX evaluates all possible revisions to the theory according to a few simple revision operators. These operators include deleting a clause, adding a clause with a single literal, deleting a literal from a clause, and adding a literal to a clause. The revision providing the best compression of the data is selected and the process continues until no more compression can be achieved.

Their approach to evaluating revisions using the MDL is an interesting one to pursue because it may be applicable to other theory revision systems such as A3. However, the class of input theories the system can handle is much more restricted than in either A3 or FORTE.

6.5 Discussion

This chapter compared A3 with other theory revision systems and with systems that perform related tasks. The most similar first-order theory revision system to A3 is FORTE. Although A3 and FORTE operate similarly in many respects, there are many differences between the two. A3 accepts a larger class of theories as input because it has capabilities to repair theories containing negation. Improving FORTE to handle negation would be a very difficult task because its architecture makes certain assumptions about the relationship between positive and negative examples

and whether a theory is overly general or overly specific. By treating all examples in a uniform manner, A3 uses simple general methods for determining which parts of a theory are overly general or overly specific.

Other differences between the two systems are the use of assumptions in A3 and revision points in FORTE, both of which are used to locate potential errors in a theory. A3 generates fewer assumptions per example than FORTE does revision points and consequently performs less search. A3 also uses a distance metric in its evaluation function which causes it to prefer minimal modifications of the initial theory. FORTE has no such evaluation component and therefore has no explicit bias towards minimal revisions which is part of the theory revision task.

The inductive components of A3 and FORTE are both based on FOIL and each one has strengths over the other. On the one hand, A3 does not have the problem of reformulating top-level examples into examples for intermediate concepts that FORTE does because A3 evaluates the addition of new literals by attempting to classify all the input training examples. On the other hand, FORTE uses the more sophisticated technique of relational pathfinding to overcome problems inherent in FOIL's hill-climbing search for literals. The advantages of these techniques could easily be added to either A3 or FORTE.

Finally, empirical comparisons of A3 and FORTE show them to both be comparable in terms of the accuracy of the revised theories but with A3 performing fewer modifications to the initial theory. This is not too surprising because A3 uses the distance metric to bias it towards making minimal revisions. Further empirical studies are needed to understanding better the differences between these two systems.

A3 was also compared with FOCL, which performs theory guided induction rather than actual theory revision. Both systems are able to overcome errors in an initial theory in order to learn accurate new theories; but, A3 does this by repairing the input theory whereas FOCL only uses it as a guide. Consequently, the theories produced by FOCL are of a much different form than those produced by A3. Empirical comparisons highlight this difference by measuring both the distance between the revised and input theories as well as the revised and correct theories. A3 performs much less revision of the initial theory and produces revised theories much closer to the correct one than does FOCL.

KR-FOCL is an extension of FOCL that performs some of aspects of the theory revision task. It uses the results of FOCL and suggestions by the user in order to decide how the original theory should be repaired. Although KR-FOCL is a step towards a full theory revision system, it requires user intervention and suffers from problems with trying to recover information from the results of learning in FOCL that are needed to decide what repairs to make. No empirical comparisons have been made between A3 and KR-FOCL because KR-FOCL requires human intervention and has not undergone extensive empirical studies on its own.

Other related systems are also discussed but they all only perform certain aspects of the first-order theory revision task. Shapiro's MIS and De Raedt's CLINT system perform theory revision but both require extensive use of an oracle. MIS can handle negation in theories but CLINT cannot. Other systems such as $\mathcal{RLA}$ perform theory guided induction and consequently have difficulties using the learned knowledge to go back and repair the original theory. GOLEM learns complex logic programs but has no facility for revising incorrect theories. LATEX demonstrated an approach to handling noisy data in a theory revision system, but the class of theories it can revise it greatly restricted.

Chapter 7
Conclusions

This chapter reviews the contributions and limitations of the work presented in this dissertation and suggests directions for future research.

7.1 Contributions

The main contributions of this work include the introduction and use of the theory distance metric, the ability to revise first-order theories containing negation, the integration of an existing first-order learning method, and the ability to utilize examples of multiple concepts in a theory.

7.1.1 Distance Metric

One of the goals of theory revision is to minimally revise a given theory in order to correctly classify a set of training examples. Almost all learning systems are evaluated with respect to the accuracy of the learned concepts, but to date, no theory revision system has been evaluated with respect to the degree of modification made to the initial theory.

This work has proposed and used a methodology for evaluating revised theories. The edit-distance between two theories is defined as the minimum number of literal-level edit operations that can transform one theory into another. Measuring the edit-distance between an initial and revised theory provides a quantitative evaluation of the degree of revision performed. The distance metric has proved to be a useful tool in many respects.

Along with accuracy, the distance metric can be used to compare the theories learned by two different systems. Just as one system can outperform another with respect to the accuracy of the revised theories, one system may make smaller changes to the initial theory that the other. Section 6.1.3 presented such results comparing A3 and FORTE on the king-rook-king domain. It was shown that both systems

learned theories that were within 1% accuracy of one another but that A3 learned theories that were on average 11% closer in distance to the initial theory. Although one cannot make broad claims about the relative performance of the two systems, these experiments do show how such systems can be compared with respect to how well they perform the theory revision task.

Because the learned theories should be close to the original, it is natural that the learner take this into account during the learning process. Just as improvement in accuracy usually guides most learning systems, A3 uses an evaluation function that favors more accurate theories and breaks ties according to the distance of the revision from the initial theory. In some of the experiments reported here A3 had to break ties according to distance in up to 50% of the cases where competing revisions were evaluated. Other tradeoffs between accuracy and distance could be built into the system if such a tradeoff were known for the specific domain. Without such a tradeoff, A3 favors more accurate revisions and only uses distance to choose among equally accurate revisions.

Using the distance metric as an evaluation tool has provided some interesting insights into the theory revision task. In all of the experiments reported here a correct theory for the domain was known and the distance between the revised and correct theories was measured. One might expect that with increasing numbers of training examples that as the accuracy of a theory increased, so would the distance to the correct theory. However, this is not the case. The most likely explanation for this behavior is that the number of accurate theories is quite large and that without additional constraints, the learner is able to produce small revisions to the initial theory that achieve a high level of accuracy even if those revisions do not make the theory syntactically closer to the correct theory. This result is encouraging in that clearly the system is able to find minimal revisions of the theory. However, it also seems desirable that the learned theories would converge on the correct one given sufficient examples. In terms of recovering the correct theory, the theory revision task is underconstrained.

7.1.2 Intermediate concepts

One approach to further constrain the theory revision task is to present examples of intermediate-level concepts in the theory along with the top-level ones as reported in Section 5.2.5. Because revisions at different points within a theory may have the same effect on accuracy of the top-level examples there is no principled way to chose among them. However, presenting examples of the intermediate-level concepts does add this missing constraint. The results on the student loan domain demonstrate that by presenting examples of intermediate-level concepts A3 was able to learn theories of somewhat greater accuracy that were also closer to the correct theory than if only top-level examples had been presented. Furthermore, increasing the number

of intermediate examples led to theories that were closer to the correct one. These experiments show how A3 is able to effectively use examples of multiple concepts in a uniform manner.

7.1.3 Negation

Existing first-order theory revision systems cannot correct errors within the scope of a negation. The ability to express concepts in terms of the negation of other concepts provides greater flexibility of representation and can provide greater comprehensibility for humans. A3 was designed in part to repair theories with negation. The system handles both positive and negative examples as well as examples of multiple concepts in a uniform manner which allows the system to correctly identify the location and types of errors in first-order theories containing negation.

Modifying other theory revision systems to handle negation is often quite difficult. Most of these systems were designed around the notion that an incorrectly classified positive example requires that some concept in the theory be generalized and that an incorrectly classified negative example requires that some concept be specialized. This is simply not the case for theories containing negation as was discussed in Section 4.2.

A3's assumption finding mechanism locates errors within a theory and identifies whether the error is due to some intermediate concept being overly general or specific. Because assumptions are generated in the same way for both positive and negative examples as well as examples from different classes, a given assumption may be used by these different classes of examples. For instance, a positive and negative example may both indicate that some intermediate concept is overly general. It is therefore the assumption itself that directly determines the location and type of error rather than the training examples.

A3 uses simple and general mechanisms for locating and repairing errors that allow it to repair theories containing errors within the scope of a negation. Adding the complexity of handling negation actually allowed A3 to be designed in a more elegant, general, and symmetric way than other systems performing first-order theory revision including AUDREY (Wogulis, 1991) which was an earlier version of A3 that performed separate generalization and specialization phases.

7.1.4 Clause Induction

Because A3 performs first-order theory revision it builds upon other work in first-order concept learning. For example, when a concept is found to be overly specific

A3 attempts to induce a new clause to cover some subset of uncovered examples. New clauses are learned and existing ones specialized by using techniques based on the first-order learner FOIL.

The basic FOIL algorithm is intended to be used to learn a set of clauses for a single concept in isolation. Although variations on FOIL such as FOCL are able to use background knowledge during learning, concepts are learned without respect to how they may be used elsewhere. For example, FOIL is able to learn a definition for the **append** relation from examples and non-examples of appending two lists together. However, the learning does not take place within a context where the concept may be used such as while attempting to learn a definition of `sort` that depends on having the **append** relation.

The FOIL algorithm is able to make certain assumptions that are only valid when learning single concepts in isolation. Such assumptions include being able to stop considering negative examples once the clause being built excludes them and the stopping criteria that no negative examples are covered by the clause being built. Systems such as FORTE that use the FOIL technique within the larger framework of theory revision often try to construct the task such that FOIL's assumptions are still valid. However, as discussed in Section 4.6, certain assumptions about the task required by the FOIL algorithm do not hold or are difficult to maintain when used for the theory revision task. These assumptions are not valid for theories containing negation.

Because a clause being learned through induction may be used within the larger context of a theory containing negation, it is no longer true that adding new literals to a clause will specialize the theory with respect to the examples. Rather, if the clause is within the scope of a negation, it may actually generalize the theory. A3 has modified the basic FOIL algorithm to consider the classification of all of the training examples when evaluating the addition of a new literal to a clause.

Because generating examples for an intermediate-level clause being learned from top-level examples can be problematic, or even impossible, A3 evaluates the addition of new literals based on classifying the original training examples. FORTE, on the other hand, goes to great lengths in order to generate a set of training examples for the clause being learned in order to use the FOIL algorithm without modification. A3 uses a generalized FOIL method that circumvents the need to force the task to conform to the solution and retains many of the positive aspects of the original FOIL method.

7.2 Limitations and Future Work

Although A3 has been shown to perform well at first-order theory revision it has a number of limitations that are interesting areas for future work. One of the main drawbacks to A3 is that it can only isolate and repair a single fault at a time. Although the system can repair theories with multiple errors there are cases where its performance could be greatly enhanced it if were to identify and repair multiple errors at once.

A3 could also be improved upon in areas currently being addressed in other first-order learning systems. Problems such as predicate invention, clause induction, noise, and efficiency have been the subject of investigation and incorporating solutions to these problems into solutions to the theory revision task provide interesting topics for future work.

7.2.1 Multiple Assumptions

A3 locates and repairs single errors in a theory at a time until no more errors can be found. An error is identified using an assumption. Because A3 only finds a single assumption to base its repairs on there are classes of theories containing multiple errors for which A3 would not find the desired solution. A3 can repair theories with multiple errors, even if those errors interact with one another. However, if two errors in a theory interact, A3 may not locate the correct source of the errors. For example, consider the following incorrect theory for `uncle`:

```
uncle(U, N) :- brother(U, P), father(P, N).
brother(A, B) :- sibling(A, B), male(A),
                 grandparent(B, A).
father(A, B) :- parent(A, B), male(A), older(B, A).
```

The theory has two errors, the clause for `brother` and the clause for `father` are both too specific because they each have an extra literal. Generating assumptions based on either of the incorrect literals would not allow any positive examples to be correctly classified. The only single assumption that A3 would find for a set of positive example would be of the form `uncle(p1, p2)`. In this case, A3 could only isolate the fault at the level of `uncle` whereas the actual error occurs at two locations deeper in the theory.

If A3 were also provided with examples of the `brother` and `father` concepts, it might be able to make repairs to those concepts one at a time. In the absence of such examples, A3 would be forced to repair the theory at the higher level `uncle` concept.

By finding multiple assumptions per example, A3 would have a better chance of finding the two errors in the theory by assuming the conjunction of the literals `grandparent(p2, p1)` and `older(p2, p1)`. However, allowing for multiple assumptions raises some difficult issues. The number of assumption sets with more than one element is likely to be quite large and the efficiency of the theory revision system would be greatly hampered. Effective ways of choosing the appropriate set of assumptions would be required.

Performing repairs based on multiple assumptions is also problematic. It may be necessary to make simultaneous repairs at both error locations in order to see an improvement in accuracy. This is likely to be the case because the errors interact and must both be solved before an incorrect example can be correctly classified. The number of combinations of simultaneous repairs to make will certainly be much larger than if only one repair were made at a time.

7.2.2 Multiple Incorrect Clauses

Similar to the problem of locating multiple errors in a theory is the problem of handling multiple overly general clauses. A3's operator for specializing concepts by selecting the best clause to specialize (Table 4.11) will fail if multiple clauses must be repaired simultaneously. For example, assume that a correct theory for `illegal` had the following two additional clauses:

```
illegal(A,B,C,D,E,F) :- A > 0.
illegal(A,B,C,D,E,F) :- B > 0.
```

These two clauses are true for any board configuration and so all negative examples will be incorrectly classified and all positive examples will be correctly classified. However, fixing just one clause at a time will not give any improvement in accuracy because the other clause will still match. Only by deleting both clauses can there be any improvement in accuracy.

This is the plateau problem with hill-climbing search and is similar in spirit to a problem encountered in FOIL (Section 2.1.3 page 15.) A hill-climbing approach to adding literals to a clause will not work if a conjunction of literals is needed to provide any information gain. The various methods for overcoming this problem in FOIL have to do with controlled look-ahead. For example, clichés (Silverstein & Pazzani, 1991), determinate literals (Quinlan, 1991) and relational pathfinding (Richards & Mooney, 1992) can be viewed as a form of constrained n-step look-ahead in the space of literals to add.

The problem with deciding which clauses of a concept to modify may require trying all 2^n possible combinations of clauses before finding the correct subset that is

faulty. Even if the number of clauses n is small there is still considerable computation required to repair the fault for any given combination of clauses. In fact, the problem with induction in FOIL may arise again if the faulty clauses need specializing: adding a new literal to only one clause will not change the accuracy and so gives no guidance for what literal to add or which clause to add it to. Clearly, further research in this area is required in order to find methods of identifying and repairing theories with multiple interacting errors at the clause level.

7.2.3 Constraints on Theory Revision

As previously discussed, certain aspects of the theory revision task are under-constrained. In particular, it is unlikely that the intended correct theory will be produced if only top-level examples are available. Often repairs at multiple locations will produce equally accurate theories and there is no information available to decide which repair to choose.

Section 5.2.5 presented one possible constraint that could help with this problem. By presenting examples of intermediate-level concepts, A3 was able to learn theories that were closer to the intended correct theory than if only top-level examples were used. However, this is only a partial solution in that each concept in the theory could be represented in many different ways.

Suppose the system were to learn a definition for the predicate `sort` and was only provided background knowledge for such concepts as `append` and `greater`. With no further constraints, the system could learn definitions based on any number of different sorting algorithms such as quicksort, or bubblesort. These solutions would all be correct in that they would correctly classify examples but may be different from some unknown intended correct solution.

Clearly, further information or constraints must be provided to the learner if the revision task also requires the theory produced to be of some particular form. Possible solutions to this problem might be to provide information to the theory revision system in the form of rule models (Davis, 1978) or higher-order relationships between predicates such as those expressible in CYC (Lenat & Guha, 1990).

7.2.4 Predicate Invention

While A3 can revise a given initial theory by repairing concepts within the theory, it does not create any new concepts. Inventing new predicates and their definitions is an important area of research that has received little attention. Methods for performing predicate invention can be found in CIGOL (Muggleton & Buntine,

1988), LFP2 (Wirth, 1989), SIERES (Wirth & O'Rorke, 1991), and CHAMP (Kijsirikul, Numao & Shimura, 1992). The introduction of new concepts into a theory can prove to be very useful and in some cases required in order to learn certain classes of concepts.

Inventing new concepts can serve to improve the accuracy of a theory but at the same time will produce revisions further away from the initial theory. The conditions under which new concepts should be created requires further investigation. Clearly, if the only way to learn a desired concept is by creating a new concept, the learner should do so. However, if there are multiple ways in which to repair a theory, the decision of whether or not to add a new concept becomes more difficult. Introducing new predicates into a theory may have an adverse effect on its distance from the original but it might actually make the theory more coherent, more maintainable, and more understandable to a human. Some means of measuring these aspects of a theory are needed beyond the measures of accuracy and distance proposed here.

7.2.5 Improvements to Clause Induction

While A3 uses a generalized FOIL algorithm as its inductive component there are still many interesting problems to be solved for first-order clause induction. In particular, FOIL has the previously mentioned problem with its hill-climbing approach in that it may not find the correct conjunction of literals to add to a clause if all are required before any improvement in accuracy is found.

Solutions to this problem such as limited lookahead, clichés, determinate literals, and relational pathfinding could all be incorporated into A3 in order to improve its inductive learning component. One of the advantages of using a FOIL based approach to induction is the possibility of improving the theory revision system with new results from first-order inductive learning as mentioned above.

Section 6.4.1 described a problem with using the FOIL approach in A3 when the theory to be revised contains a negation. The basic problem is that revisions to a theory must be evaluated in terms of the accuracy of the revision on the entire training set rather than only on the remaining uncovered positives. Consequently, A3 may only learn a subset of the clauses that would be learned by FOIL alone. This problem is easy to overcome if the theory contains no negation, but requires further research in order to match the performance of FOIL in the more general case where theories may contain negation. Even without these improvements to the basic FOIL method, A3 has been shown to handle a large variety of first-order theory revision tasks.

7.2.6 Noise

An important area of research in machine learning is the handling of noisy data. Some work has been done with handling noise in first-order learning (Brunk & Pazzani, 1991; Dzeroski & Lavrac, 1991; Ali & Pazzani, 1993) as well as in theory revision (Tangkitvanich & Shimura, 1993).

Currently, A3 could accept noisy training data and would produce a revised theory based upon it. However, it is doubtful that the resulting theory would perform very well in terms of accuracy, only because the system was designed without taking noise into consideration.

Tangkitvanich and Shimura (1993) presented a method for handling noisy data within their theory revision system LATEX. Their approach is based on the minimum description length principle that favors the least revision of the initial theory that can accurately cover the training data. However, the class of input theories it can revise is greatly restricted. It would be relatively simple to change A3's evaluation function to use their approach which might provide a better method of handling noisy data.

Because many of A3's repair operators are based on FOIL, any improvements to FOIL that handle noise are likely to be applicable in A3 as well. However, the system's method of selecting the best assumption upon which to base revisions to the theory would have to be changed. Assumptions would still be generated for both noisy and non-noisy examples but the criteria for selecting the best assumption may have to be based on something more than just the most commonly occurring assumption.

7.2.7 Efficiency

Theory revision is a computationally expensive task. No longer is the system learning just a single concept in isolation, but rather in the context of a larger theory. As implemented, A3 evaluates competing revisions to a theory based upon the accuracy of the revised theories on the training examples. This accuracy is obtained by attempting to classify each training example which involves proving the example using the theory. This is a costly process and is one of the most common operations in the system. The inductive component is an especially heavy user because it must evaluate the addition of large numbers of literals to a clause. This is probably the most fruitful area to explore in order to improve the efficiency of the system.

One approach taken but not reported here was to cache all intermediate results of the theorem proving process so that these results may be resued when re-proving an example with a different modification to the theory. However, changes to the theory will cause some or all of these cached result to be invalidated and to be removed from the cache. In the domains presented here, the overhead of caching was essentially

equivalent to benefits received so that there was no overall net gain in the system's performance. It is possible that for other domains, especially ones with larger numbers of concepts that caching might prove beneficial if large areas of the theory remained unaffected by a single change.

Other ways of improving the efficiency of the inductive component is to weed out candidate literals before evaluating them. Using constraints such as type and mode of the arguments to a literal can also greatly improve efficiency. Further work into identifying useful literals to try becomes especially important as the size of the background theory grows. One would hope that with more knowledge, the learner would be more efficient than with less.

7.3 Final Thoughts

This dissertation has presented a solution to the theory revision problem that can handle theories with multiple concepts expressed as function-free Horn clauses including negation. Systems performing theory revision are able to take advantage of, and even repair, previously acquired knowledge. This offers a great advantage over other machine learning systems that can only learn single concepts at a time without the ability to repair incorrect knowledge. As machine learning solutions are applied to real-world problems, this ability to repair previously acquired knowledge, be it from a human or other source, becomes increasingly important. A3 has been shown to repair a class of theories with greater representational power than previous systems performing theory revision and has pointed out directions for future research.

References

Ali, K. M. & Pazzani, M. J. (1993). HYDRA: A noise tolerant relational concept learning algorithm. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1064–1070)., Chambéry, France. Morgan Kaufmann.

Bain, M. & Muggleton, S. H. (1991). Non-monotonic learning. In J. E. Hayes, D. Michie, & E. Tyugu (Eds.), *Machine Intelligence*, volume 12. Oxford University Press.

Bergadano, F. & Giordana, A. (1988). A knowledge intensive approach to concept induction. In *Proceedings of the Fifth International Conference on Machine Learning*, (pp. 305–317)., Ann Arbor, Michigan. Morgan Kaufmann.

Bratko, I. (1986). *Prolog programming for artificial intelligence.* Addison-Wesley.

Brunk, C. A. & Pazzani, M. J. (1991). An investigation of noise-tolerant relational concept learning algorithms. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 389–393)., Evanston, IL. Morgan Kaufmann.

Buntine, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, *36*, 149–176.

Clancey, W. J. (1992). Model construction operators. *Artificial Intelligence*, *53*, 1–115.

Clark, P. & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*, 251–283.

Cohen, W. (1992a). Abductive explanation-based learning: A solution to the multiple inconsistent explanation problem. *Machine Learning*, *8*, 167–219.

Cohen, W. (1992b). Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 100–110)., Aberdeen, Scotland. Morgan Kaufmann.

Darley, J. M. & Shultz, T. R. (1990). Moral rules: Their content and acquisition. *Annual Review of Psychology*, *41*, 525–556.

Davis, R. T. (1978). Model-directed learning of production rules. In D. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems.* Academic Press.

De Raedt, L. (1991). *Interactive concept learning.* PhD thesis, Katholieke Universiteit Leuvan, Belgium.

De Raedt, L. & Bruynooghe, M. (1989). Towards friendly concept-learners. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (pp. 849–854)., Detroit, MI. Morgan Kaufmann.

De Raedt, L. & Bruynooghe, M. (1990). On negation and three-valued logic in interactive concept-learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*. Pitman.

Dzeroski, S. & Lavrac, N. (1991). Learning relation from noisy examples: An empirical comparison of LINUS to FOIL. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 399–402)., Evanston, IL. Morgan Kaufmann.

Flann, N. & Dietterich, T. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, *4*, 187–226.

Ginsberg, A. (1988). Theory revision via prior operationalization. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, (pp. 590–595)., St. Paul, MN. Morgan Kaufmann.

Ginsberg, A. (1990). Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 777–782)., Boston, MA. Morgan Kaufmann.

Haussler, D. (1987). Learning conjunctive concepts in structural domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, (pp. 466–470)., Seattle, WA. Morgan Kaufmann.

Hirsh, H. (1989). Combining empirical and analytical learning with version spaces. In *Proceedings of the Sixth International Workshop on Machine Learning*, (pp. 29–33)., Ithaca, NY. Morgan Kaufmann.

Kijsirikul, B., Numao, M., & Shimura, M. (1991). Efficient learning of logic programs with non-determinate, non-discriminating literals. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 417–421)., Evanston, IL. Morgan Kaufmann.

Kijsirikul, B., Numao, M., & Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 44–49)., San Jose, CA. Morgan Kaufmann.

Kuhn, T. (1962). *The structure of scientific revolutions.* University of Chicago Press.

Lawler, E. L. (1976). *Combinatorial optimization : networks and matroids.* Holt, Rinehart and Winston.

Lenat, D. & Guha, R. V. (1990). *Building large knowledge-based systems: Representation and inference in the CYC project.* Addison-Wesley.

Lloyd, J. W. (1984). *Foundations of logic programming.* Springer-Verlag.

Michalski, R. S. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *2*, 349–361.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In *Machine learning: An artificial intelligence approach*, volume 1 (pp. 83–134). Tioga Publishing Co.

Mingers, J. (1989). An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, *3*, 319–342.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*, 47–80.

Mitchell, T. M. (1978). *Version spaces: An approach to concept learning.* PhD thesis, Stanford University.

Mooney, R. & Richards, B. (1992). Automated debugging of logic programs via theory revision. In *Proceedings of the International Workshop on Inductive Logic Programming*, Tokyo, Japan.

Muggleton, S., Bain, M., Hayes-Michie, J., & Michie, D. (1989). An experimental comparison of human and machine learning formalisms. In *Proceedings of the Sixth International Workshop on Machine Learning*, (pp. 113–118)., Ithaca, NY. Morgan Kaufmann.

Muggleton, S. & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Workshop on Machine Learning*, (pp. 339–352)., Ann Arbor, MI. Morgan Kaufmann.

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

Nilsson, N. J. (1980). *Principles of artificial intelligence.* Tioga Publishing Co.

Norvig, P. (1992). *Paradigms of artificial intelligence programming: Case studies in Common Lisp.* Morgan Kaufmann.

Ourston, D. & Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 815–820)., Boston, MA. Morgan Kaufmann.

Pagallo, G. & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, *5*, 71–99.

Pazzani, M. J. & Brunk, C. A. (1991). Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning. *Knowledge Acquisition*, *3*, 157–173.

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

Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer, B. & Michie, D. (Eds.), *Machine Intelligence*, volume 5, (pp. 153–163).

Plotkin, G. D. (1971). *Automatic methods of inductive inference.* PhD thesis, Edinburgh University.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

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

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 442–446)., Evanston, IL. Morgan Kaufmann.

Rabinowitz, H., Flamholtz, J., Wolin, E., & Euchner, J. (1991). Nynex Max: A telephone trouble screening expert system. In R. G. Scott & A. C. Scott (Eds.), *Innovative applications of artificial intelligence*, volume 3 (pp. 213–230). AAAI Press.

Richards, B. & Mooney, R. (1991). First-order theory revision. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 447–451)., Evanston, IL. Morgan Kaufmann.

Richards, B. & Mooney, R. (1992). Learning relations by pathfinding. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 50–55)., San Jose, CA. Morgan Kaufmann.

Richards, B. L. (1992). *An operator-based approach to first-order theory revision.* PhD thesis, University of Texas, Austin, TX.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, *14*, 465–471.

Shapiro, E. Y. (1983). *Algorithmic program debugging.* MIT Press.

Shasha, D. & Zhang, K. (1990). Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, *11*, 581–621.

Shavlik, J. & Towell, G. (1989). Combining explanation-based learning and artificial neural networks. In *Proceedings of the Sixth International Workshop on Machine Learning*, (pp. 90–93)., Ithaca, NY. Morgan Kaufmann.

Shultz, T. R. (1990). A rule based model of judging harm-doing. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, (pp. 229–236)., Cambridge, MA. Lawrence Erlbaum.

Silverstein, G. & Pazzani, M. J. (1991). Relational clichés: constraining constructive induction during relational learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 203–207)., Evanston, IL. Morgan Kaufmann.

Srinivasan, A., Muggleton, S., & Bain, M. (1992). Distinguishing exceptions from noise in non-monotonic learning. In *Proceedings of the International Workshop on Inductive Logic Programming*, Tokyo, Japan.

Swartout, W. (1981). Explaining and justifying in expert consulting programs. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, (pp. 203–208)., Vancouver, B.C. Morgan Kaufmann.

Tai, K. C. (1979). The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, *26*, 422–433.

Tangkitvanich, S., Numao, M., & Shimura, M. (1992). Correcting multiple faults in the concept and subconcepts by learning and abduction. In *Proceedings of the International Workshop on Inductive Logic Programming*, Tokyo, Japan.

Tangkitvanich, S. & Shimura, M. (1993). Learning from an approximate theory and noisy examples. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, (pp. 466–471)., Washington, DC. AAAI Press.

Towell, G. (1991). *Symbolic knowledge and neural networks: Insertion, refinement and extraction.* PhD thesis, University of Wisconsin, Madison, WI.

Towell, G. G., Shavlik, J. W., & Noordewier, M. O. (1990). Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 861–866)., Boston, MA. Morgan Kaufmann.

Vere, S. A. (1975). Induction of concepts in the predicate calculus. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, (pp. 281–287)., Tiblisi, USSR. Morgan Kaufmann.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The psychology of computer vision.* McGraw-Hill.

Wirth, R. (1989). Completing logic programs by inverse resolution. In *Proceedings of the Fourth European Working Session on Learning*, (pp. 239–250)., Montpellier, France. Morgan Kaufmann.

Wirth, R. & O'Rorke, P. (1991). Constraints on predicate invention. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 457–461)., Evanston, IL. Morgan Kaufmann.

Wogulis, J. (1991). Revising relational domain theories. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 462–466)., Evanston, IL. Morgan Kaufmann.

Wogulis, J. (1993). Handling negation in first-order theory revision. In *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, Chambéry, France.

Wogulis, J. & Pazzani, M. J. (1993). A methodology for evaluation theory revision systems: Results with AudreyII. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1128–1134)., Chambéry, France. Morgan Kaufmann.

Appendix A
The Moral Reasoner Theory

A.1 Moral Reasoner Theory Clauses

The following are the clauses comprising the moral reasoner domain.

```
guilty(X) :- blameworthy(X).
guilty(X) :- vicarious_blame(X).

blameworthy(X) :-
  responsible(X),
  not(justified(X)),
  severity_harm(X,H),
  benefit_victim(X,V),
  greater(H,V).

vicarious_blame(X) :-
  vicarious(X),
  not(justified(X)),
  severity_harm(X,H),
  benefit_victim(X,V),
  greater(H,V).

vicarious(X) :-
  someone_else_cause_harm(X,Y),
  outrank_perpetrator(X,Y),
  control_perpetrator(X,Y), yes(Y).

justified(X) :-
  achieve_goal(X,Y), yes(Y),
  goal_outweigh_harm(X,Y),
  goal_achievable_less_harmful(X,Z), no(Z).
```

```
responsible(X) :-
  cause(X),
  not(accident(X)),
  voluntary(X),
  foreseeable(X),
  not(intervening_cause(X)).

cause(X) :- produce_harm(X,H), yes(H).
cause(X) :- necessary_for_harm(X,H), yes(H).
cause(X) :- sufficient_for_harm(X,H), yes(H).

accident(X) :-
  not(intend(X)),
  not(reckless(X)),
  not(negligent(X)).

foreseeable(X) :- foreseeability(X,F), high(F).
foreseeable(X) :- foreseeability(X,F), low(F).
foreseeable(X) :- foreseeability(X,F), yes(F).

reckless(X) :- mental_state(X,M), reckless_m(M).

reckless(X) :-
  careful(X,C),
  no(C),
  not(strong_intend(X)),
  foreseeability(X,F),
  high(F).

negligent(X) :-
  mental_state(X,M),
  negligent_m(M).

negligent(X) :-
  careful(X,C), no(C),
  not(strong_intend(X)),
  foreseeability(X,F),
  low(F).

intend(X) :- strong_intend(X).
intend(X) :- weak_intend(X).
```

```
strong_intend(X) :-
  mental_state(X,M),
  intend_m(M).

strong_intend(X) :-
  plan_known(X,Y),
  plan_include_harm(X,Y),
  harm_caused_as_planned(X,Y), yes(Y).

weak_intend(X) :-
  weak_intend1(X),
  not(reckless(X)),
  not(negligent(X)).

weak_intend1(X) :- not(discount_intent(X)).
weak_intend1(X) :- monitor(X,Y), yes(Y).
weak_intend1(X) :- benefit_protagonist(X,Y), yes(Y).

discount_intent(X) :- external_cause(X,Y), yes(Y).

voluntary(X) :- external_force(X,Y), no(Y).

intervening_cause(X) :-
  intervening_contribution(X,Y), yes(Y),
  foresee_intervention(X,Z), no(Z).

greater(X,Y) :- X > Y.

eq(X,X).

yes(y).
no(n).
high(high).
low(low).

reckless_m(reckless).
negligent_m(negligent).
intend_m(intend).
```

A.2 Seventeen classes of positive examples in the Moral Reasoner domain.

The following are the seventeen classes of positive examples of `guilty` used for the experiments reported in Section 5.2.1. Each class is represented as the conjunction of literals from the background knowledge. For example, the first class listed below covers all positive examples where the person caused the event, their mental state was negligent, the action was not justified, there was no external force, the event was highly foreseeable, there were no intervening cases, there was severe harm to the victim, and the victim received no benefit. Section 5.2.1 discusses how these classes were determined.

```
cause(X), mental_state(X, negligent), not(justified(X)),
external_force(X,no), foreseeability(X,high),
not(intervening_cause(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), mental_state(X, negligent), not(justified(X)),
external_force(X,no), foreseeability(X,low),
not(intervening_cause(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), careful(X,no), not(intervening_cause(X)),
foreseeability(X,low), external_force(X,no),
severity_harm(X,1), benefit_victim(X,0),
not(justified(X)).

cause(X), mental_state(X, reckless), not(justified(X)),
external_force(X,no), foreseeability(X,high),
not(intervening_cause(X)),
severity_harm(X,1), benefit_victim(X,0).

cause(X), mental_state(X, reckless), not(justified(X)),
external_force(X,no), foreseeability(X,low),
not(intervening_cause(X)), severity_harm(X,1),
benefit_victim(X,0).
```

```
cause(X), careful(X,no), foreseeability(X,high),
external_force(X,no), not(intervening_cause(X)),
not(justified(X)) severity_harm(X,1),
benefit_victim(X,0).

cause(X), mental_state(X, intend), external_force(X,no),
foreseeability(X,high), not(intervening_cause(X)),
not(justified(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), mental_state(X, intend), external_force(X,no),
foreseeability(X,low), not(intervening_cause(X)),
not(justified(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), harm_caused_as_planned(X,yes),
plan_include_harm(X,yes), plan_known(X,yes),
external_force(X,no), foreseeability(X,high),
not(intervening_cause(X)), not(justified(X)),
severity_harm(X,1), benefit_victim(X,0).

cause(X), harm_caused_as_planned(X,yes),
plan_include_harm(X,yes), plan_known(X,yes),
external_force(X,no), foreseeability(X,low),
not(intervening_cause(X)), not(justified(X)),
severity_harm(X,1), benefit_victim(X,0).

cause(X), monitor(X,yes), external_force(X,no),
foreseeability(X,high), not(intervening_cause(X)),
not(justified(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), monitor(X,yes), external_force(X,no),
foreseeability(X,low), not(intervening_cause(X)),
not(justified(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), benefit_protagonist(X,yes), not(justified(X)),
external_force(X,no), foreseeability(X,high),
not(intervening_cause(X)), severity_harm(X,1),
benefit_victim(X,0).
```

```
cause(X), benefit_protagonist(X,yes), not(justified(X)),
external_force(X,no), foreseeability(X,low),
not(intervening_cause(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), not(discount_intent), external_force(X,no),
foreseeability(X,high), not(intervening_cause(X)),
not(justified(X)), severity_harm(X,1),
benefit_victim(X,0).

cause(X), not(discount_intent), external_force(X,no),
foreseeability(X,low), not(intervening_cause(X)),
not(justified(X)), severity_harm(X,1),
benefit_victim(X,0).

vicarious(X), not(justified(X)),
severity_harm(X,1), benefit_victim(X,0).
```

Appendix B
The Student Loan Domain

B.1 Correct Student Loan Theory

The following are the clauses from the correct theory for the student loan domain.

```
no_payment_due(Person) :-
    continuously_enrolled(Person).
no_payment_due(Person) :-
    eligible_for_deferment(Person).

continuously_enrolled(Person) :-
    never_left_school(Person),
    enrolled_in_more_than_n_units(Person, 5).

never_left_school(Person) :-
    longest_absence_from_school(Person, Months),
    Months <= 6.

eligible_for_deferment(Person) :-
    military_deferment(Person).
eligible_for_deferment(Person) :-
    peace_corps_deferment(Person).
eligible_for_deferment(Person) :-
    financial_deferment(Person).
eligible_for_deferment(Person) :-
    student_deferment(Person).
eligible_for_deferment(Person) :-
    disability_deferment(Person).
```

```
military_deferment(Person) :-
    enlist(Person, Organization),
    armed_forces(Organization).

peace_corps_deferment(Person) :-
    enlist(Person, Organization),
    peace_corps(Organization).

financial_deferment(Person) :-
    filed_for_bankruptcy(Person).

financial_deferment(Person) :-
    unemployed(Person).

student_deferment(Person) :-
    enrolled_in_more_than_n_units(Person, 11).

disability_deferment(Person) :-
    disabled(Person).

/* background clauses */

enrolled_in_more_than_n_units(Person, Number) :-
    enrolled(Person, School, Units),
    school(School),
    Units > Number.
```

B.2 Incorrect Student Loan Theory

The following clauses form the incorrect student loan theory. The contains four errors.

```
no_payment_due(Person) :-
    continuously_enrolled(Person).
no_payment_due(Person) :-
    eligible_for_deferment(Person).

/* missing literal: never_left_school(Person) */
```

```
continuously_enrolled(Person) :-
    never_left_school(Person),
    enrolled_in_more_than_n_units(Person, 5).

never_left_school(Person) :-
    longest_absence_from_school(Person, Months),
    Months <= 6.

eligible_for_deferment(Person) :-
    military_deferment(Person).
eligible_for_deferment(Person) :-
    peace_corps_deferment(Person).
eligible_for_deferment(Person) :-
    financial_deferment(Person).
eligible_for_deferment(Person) :-
    student_deferment(Person).
eligible_for_deferment(Person) :-
    disability_deferment(Person).

military_deferment(Person) :-
    enlist(Person, Organization),
    armed_forces(Organization).

peace_corps_deferment(Person) :-
    enlist(Person, Organization),
    peace_corps(Organization).

financial_deferment(Person) :-
    filed_for_bankruptcy(Person).

/* missing clause
financial_deferment(Person) :-
   unemployed(Person).
*/

student_deferment(Person) :-
    enrolled_in_more_than_n_units(Person, 11).

disability_deferment(Person) :-
    filed_for_bankruptcy(Person),   /* extra literal */
    disabled(Person).

/* extra clause */
```

```
financial_deferment(Person) :-
    enrolled(Person, School, Units),
    uci(School).

/* background clauses */

enrolled_in_more_than_n_units(Person, Number) :-
    enrolled(Person, School, Units),
    school(School),
    Units > Number.
```

Appendix C
A3 and Forte King-Rook-King Results

The two tables in this Appendix report the results for each trial run comparing A3 and FORTE on randomly mutated king-rook-king theories at 50 and 250 examples. See Section 6.1.3 for further details and comparisons.

Table C.1: Accuracy and distance measures for A3 and FORTE on 20 incorrect theories for `illegal` with 50 training examples.

A3			FORTE		
Accuracy	Distance		Accuracy	Distance	
	Initial	Correct		Initial	Correct
97.33	3	8	97.33	3	5
92.21	2	3	86.62	3	4
98.46	2	2	98.67	3	5
97.13	5	9	93.49	6	11
88.00	2	6	98.00	3	6
83.69	6	10	94.82	8	8
98.36	2	5	99.69	2	5
95.64	5	9	98.36	3	3
92.05	6	9	91.74	6	12
97.03	2	7	97.03	3	4
97.08	1	3	97.08	1	3
98.05	1	5	98.05	1	5
96.92	1	4	96.92	1	4
96.56	1	5	94.62	4	8
96.82	4	5	98.46	3	3
93.03	1	5	93.03	1	5
98.67	1	5	98.67	1	5
96.00	5	9	98.46	4	4
96.56	4	5	97.08	5	6
92.15	1	4	93.13	1	4
95.09	**2.75**	**5.90**	**96.06**	**3.10**	**5.50**

Table C.2: Accuracy and distance measures for A3 and FORTE on 18 incorrect theories for `illegal` with 250 training examples.

A3			FORTE		
Accuracy	Distance		Accuracy	Distance	
	Initial	Correct		Initial	Correct
99.71	1	5	99.89	1	3
98.91	4	7	99.66	4	8
100.00	4	7	100.00	4	3
99.54	4	7	99.89	7	13
96.86	2	4	100.00	2	4
97.83	0	4	99.71	4	8
99.89	2	4	99.89	2	4
97.03	4	8	99.71	7	7
100.00	2	3	100.00	2	3
99.89	3	1	99.89	5	6
99.71	8	9	99.54	6	9
96.86	3	4	100.00	4	5
99.89	5	7	99.89	5	7
99.66	5	9	99.71	6	11
96.40	4	7	99.71	4	8
100.00	5	6	100.00	4	3
98.00	8	11	100.00	5	11
99.89	5	9	99.89	6	4
98.89	**3.83**	**6.22**	**99.85**	**4.33**	**6.50**

Appendix D
King-Rook-King Mutation Results

This appendix contains the averages and standard deviation for the experiments with mutated king-rook-king theories. The results are averaged over 20 trials. More detailed analyses of these results are reported in Section 5.2.4.

Table D.1. King-Rook-King Mutation Results for All Mutations

Examps.	Accuracy			
	1	2	4	8
0	88.97 ± 23.14	79.46 ± 23.91	57.06 ± 25.17	52.64 ± 23.30
75	99.68 ± 0.51	98.03 ± 2.18	97.85 ± 2.05	93.81 ± 3.57
200	99.47 ± 1.21	98.89 ± 1.78	98.38 ± 1.81	96.54 ± 2.86

Examps.	Distance from Original			
	1	2	4	8
0	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
75	0.45 ± 0.59	1.30 ± 0.95	2.55 ± 1.43	2.85 ± 1.19
200	0.55 ± 0.59	1.90 ± 1.41	2.85 ± 1.53	4.00 ± 1.90

Examps.	Distance to Correct			
	1	2	4	8
0	1.15 ± 0.36	2.25 ± 0.70	4.25 ± 0.70	7.15 ± 1.49
75	1.40 ± 0.73	2.75 ± 1.58	5.15 ± 1.65	7.95 ± 2.22
200	1.35 ± 0.65	3.20 ± 1.94	5.10 ± 2.39	9.00 ± 2.88

Table D.2. King-Rook-King Mutation Results for Add Clause

Examps.	Accuracy			
	1	2	4	8
0	82.61 ± 25.46	62.28 ± 27.01	53.90 ± 23.35	43.72 ± 17.81
75	99.98 ± 0.07	98.84 ± 2.56	98.14 ± 2.72	97.34 ± 3.71
200	99.98 ± 0.08	99.05 ± 1.89	98.88 ± 2.02	97.82 ± 2.95

Examps.	Distance from Original			
	1	2	4	8
0	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
75	0.45 ± 0.50	1.10 ± 0.77	2.35 ± 1.24	3.10 ± 1.30
200	0.45 ± 0.50	1.75 ± 1.58	2.75 ± 1.76	3.45 ± 2.13

Examps.	Distance to Correct			
	1	2	4	8
0	1.60 ± 0.49	2.85 ± 0.65	5.60 ± 0.80	10.65 ± 1.59
75	1.50 ± 0.81	2.85 ± 1.46	5.20 ± 2.16	9.70 ± 2.70
200	1.50 ± 0.81	3.20 ± 2.25	6.25 ± 2.72	9.95 ± 3.64

Table D.3. King-Rook-King Mutation Results for Delete Clause

Examps.	Accuracy			
	1	2	4	8
0	95.78 ± 5.54	92.25 ± 6.24	81.91 ± 6.89	71.35 ± 4.44
75	98.03 ± 2.13	97.97 ± 1.69	95.61 ± 2.58	91.14 ± 2.63
200	98.99 ± 1.26	98.63 ± 1.46	97.68 ± 1.95	94.29 ± 2.21

Examps.	Distance from Original			
	1	2	4	8
0	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
75	1.00 ± 0.84	1.25 ± 0.83	2.85 ± 0.96	4.00 ± 1.70
200	1.15 ± 0.85	1.80 ± 0.93	3.45 ± 1.47	5.00 ± 2.43

Examps.	Distance to Correct			
	1	2	4	8
0	1.45 ± 0.50	2.75 ± 0.62	5.80 ± 1.12	13.45 ± 1.75
75	2.35 ± 1.06	3.45 ± 1.36	7.15 ± 1.71	14.05 ± 2.40
200	2.55 ± 1.12	3.70 ± 1.27	7.20 ± 1.99	14.40 ± 2.18

Table D.4. King-Rook-King Mutation Results for Add Literal

Examps.	Accuracy			
	1	2	4	8
0	96.91 ± 3.56	94.74 ± 5.57	89.75 ± 6.33	83.33 ± 7.32
75	99.32 ± 1.36	99.01 ± 1.33	97.93 ± 2.83	97.24 ± 2.60
200	99.74 ± 0.73	99.75 ± 0.48	99.43 ± 0.92	98.65 ± 1.13

Examps.	Distance from Original			
	1	2	4	8
0	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
75	0.80 ± 0.93	1.20 ± 1.12	1.80 ± 1.12	3.25 ± 1.26
200	0.80 ± 0.75	1.50 ± 1.24	2.55 ± 1.28	3.75 ± 1.76

Examps.	Distance to Correct			
	1	2	4	8
0	1.00 ± 0.00	1.95 ± 0.22	4.00 ± 0.00	7.95 ± 0.22
75	1.50 ± 1.12	2.95 ± 1.20	5.20 ± 1.17	10.30 ± 1.38
200	1.40 ± 1.02	2.50 ± 1.47	5.85 ± 1.49	10.15 ± 1.59

Table D.5. King-Rook-King Mutation Results for Delete Literal

Examps.	Accuracy			
	1	2	4	8
0	78.51 ± 29.19	49.12 ± 25.90	43.76 ± 22.77	34.20 ± 0.00
75	99.71 ± 0.70	98.04 ± 2.52	98.90 ± 1.24	93.08 ± 3.61
200	99.93 ± 0.09	99.35 ± 1.14	99.14 ± 0.98	95.69 ± 3.12

Examps.	Distance from Original			
	1	2	4	8
0	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
75	0.55 ± 0.50	1.50 ± 0.67	1.85 ± 1.19	2.95 ± 1.24
200	0.55 ± 0.50	1.85 ± 0.91	2.35 ± 1.82	3.60 ± 1.83

Examps.	Distance to Correct			
	1	2	4	8
0	1.00 ± 0.00	1.95 ± 0.22	3.55 ± 0.50	6.35 ± 0.57
75	0.90 ± 0.30	1.90 ± 0.62	3.55 ± 1.12	7.15 ± 1.06
200	0.80 ± 0.40	1.85 ± 1.28	3.95 ± 1.66	7.30 ± 1.27