1993

Learning Complex Recursive Rules.

Ching-liang Tseng
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9405427

Learning complex recursive rules

Tseng, Ching-Liang, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

LEARNING COMPLEX RECURSIVE RULES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Ching-Liang Tseng
Diploma, Ming-Chi Institute of Technology, 1975
M.S., University of Mississippi, 1982
August 1993

ACKNOWLEDGEMENTS

I wish to express my sincerest thanks and gratitude to Dr. Sukhamay Kundu, who guided my work with effort, patience, and skill.

Thanks also extended to the Department of Computer Science for financial assistance; to Dr. Jianhua Chen, Dr. Donald C. Freshwater, Dr. S. Sitharama Iyengar, Dr. J. Bush Jones, Dr. Subhash C. Kak, and Dr. Donald H. Kraft, for being on my graduate committee.

I also like to thank my wife, Mei-Hsiang, for her love, sacrifice, encouragement, and support throughout the course of my study at LSU.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We consider the problem of learning complex recursive rules which involve new concepts other than those given in the input relations and which must be discovered by the learning algorithm in the course of finding rules. The existing learning methods (FOIL, FORGE, and etc.) create rules based on the given concepts or relations but they cannot create new concepts. However, in many cases one must use new intermediate concepts in order to form the recursive rules. We give a new technique for constructing such intermediate concepts and learning rules based on those concepts. We illustrate the new technique with several examples, none of which can be handled by the existing methods. We have implemented the new technique in Common Lisp and tested many different examples.

CHAPTER I

INTRODUCTION

In recent years, many systems [4-5, 8, 10, 17-19, 22, 25, 33] for learning with first-order Horn-clauses have been developed. Learning recursive rules is particularly important. In FORGE [4, 6] and FOIL [25], recursive rules are constructed directly out of raw data, whereas GOLEM [8] uses inverse resolution to construct recursive rules by starting from non-recursive rules. FORGE combines FOIL's coverage measure with explanation-based learning [2-3, 11, 14-16, 20-21] to enhance the efficiency of the learning process. FOCL [22], an extension of FOIL, uses background knowledge as a hint to guide the rule finding process. These systems are highly dependent on input relations and cannot create new relations (concepts) to form rules. However, sometimes it is not possible to obtain correct recursive rules without introducing new intermediate relations. We describe here a new technique for forming the *necessary* intermediate concepts, and these concepts are formed concurrently with the construction of rules. Our method can be used in extending both FOIL and FORGE in order to learn more complex rules.

We have implemented the new technique on the top of FORGE. FORGE chooses a simplest target tuple t based on a linear ordering of the target tuples, forms the ground explanations of t, and then obtains a rule to cover t by generalizing one of those ground explanations. These steps are repeated until all target tuples are covered. In our learning system, a single rule construction process is similar to FORGE but is in a more efficient way. Each rule construction step is followed by an attempt to merge two rules together into a set of recursive rules, if possible. Two recursive rules are then merged in an attempt to cover more target tuples. The merging process is in a hierarchical fashion until no more increment of the coverage can be obtained. The new concept is introduced by modifying recursive rules in a certain way and introducing a rule for the new concept. At

the very end, we perform an additional clean-up step to eliminate one or more new concepts, if they are *unnecessary*, i.e., they are subsumed by other new concepts or they are not involved in recursion. We adopt the explanation-base learning paradigm as used in FORGE because it uses fewer generalizations and specializations to obtain a rule compared to method used in FOIL. However, our learning system use a more efficient strategy to obtain the rules.

There are other systems, such as SIERES [33], which have the ability to invent new predicates. The method in [33] integrates abduction and induction in a natural way. It uses certain constraints (syntactic least common anti-instance, critical terms, and argument dependency graphs) for predicate invention. However, this method learns one clause at a time and is not capable of learning arbitrary disjunctive definition. On the other hand, the concepts discovered by CIGOL [17], which is based on inverse resolution, are not necessary; they are of an elementary nature, do not involve recursion in their definition, and merely help simplify (shorten) the final rules.

We first develop the concepts of our learning algorithm and then demonstrate the algorithm by considering several examples. The first two examples are artificially made examples to keep things simple, and to illustrate the formation of the rules step by step. Then, we consider four more examples in order to compare with FOIL and FORGE systems.

CHAPTER II

BASIC DEFINITIONS

In this chapter, we will review some basic definitions and paradigms related to our learning system.

2.1 Horn-clauses

The output of our learning system is a set of Horn-clauses [1, 9-10], which are the clauses used in Prolog and is a subset of first order logic. A Horn-clause contains at most one positive literal; i.e., it is of the form:

$$AV\neg B_1 V\neg B_2 V\cdots V\neg B_n$$

where $A, B_1, B_2, \cdots,$ and B_n are literals. We can also write Horn-clauses as implication with the positive literal as the conclusion:

$$A\leftarrow B_1 \Lambda B_2 \Lambda\cdots\Lambda B_n$$

We can also write Horn-clauses in the following form, which is used in Prolog and is the rule obtained in our algorithm:

$$A\leftarrow B_1, B_2, \cdots, B_n.$$

A is the head of the rule and the conjunction of the B_i is the body of the rule.

2.2 Absorption

Absorption [17, 27-29] is one of the operations used in the inversion of resolution. The absorption operation rewrites one rule R_2 by using a concept defined in another rule R_1. In order to perform absorption, the body of R_1 must be able to be unified with a part of the body of R_2. For example, if we have the following two rules:

(2.1) reach(X, Y) ← link(X, Y).

(2.2) reach(X, Y) ← link(X, W), link(W, Y).

In above example, R_1 is rule (2.1), which is the absorbed rule, and R_2 is rule (2.2), which is the absorbent rule. Rule (2.1) can be absorbed by rule (2.2) because its body link(X, Y), can be unified with the subpart link(X, W), of the body of rule (2.2) with the substitution {X/X, Y/W}. Therefore, we can replace link(X, W) of rule (2.2) by reach(X, W) which is the head of rule (2.1) after substitution. The new rule after absorption is:

(2.3) reach(X, Y) ← reach(X, W), link(W, Y).

Combining rule (2.1) and rule (2.3) together, we form a set of recursive rules. Similarly, the body link(X, Y) of rule (2.1) can be unified with the subpart link(W, Y) of the body of rule (2.2) with the substitution {X/W, Y/Y}, and the new rule after absorption is:

(2.4) reach(X, Y) ← link(X, W), reach(W, Y).

2.3 Inductive Learning

Inductive learning [12-13, 23-24, 30] is a process of acquiring knowledge by inducing a general concept description from a sequence of positive examples of the concept and negative examples of the concept. The concept description obtained by the inductive learning can rederive the positive examples via universal instantiation, but none of the negative examples can be rederived. FOIL learning system is an inductive learning system, and the concept description obtained by this learning system is expressed as a set of Horn-clauses rules. This system is also a one-shot inductive learning system, since all the positive and negative examples which are used to produce a concept description will not be considered for further modification.

FOIL is a pure inductive learner (empirical learning approach) since it does not utilize the domain or background knowledge to help the concept finding process. In recent years, many learning systems have been proposed to improve the efficiency of FOIL by using domain or background knowledge. Richards and Mooney [26] have suggested a relational pathfinding method by viewing FOIL rule research as a hill climing algorithm. FORGE and our learning system, use an approach called explanation-based learning to improve efficiency.

2.4 Explanation-based Learning

If we consider inductive learning as an empirical approach, then explanation-based learning can be considered as a knowledge-intensive and analytical approach. The power of explanation-based learning comes from the utilization of the domain or background knowledge to analyze the examples. A positive example is used to analyze why it is the instance of the concept. The explanation identifies the relevant properties of the example, and then is further generalized to obtained the concept description.

FORGE uses explanation-based learning approach to obtain a concept description, which is a set of rules. A target relation tuple is picked to be explained by the base relation tuples and target relation tuples (since FORGE assumes a single concept, the target relation tuples are also considered). The explanations form a tree which are called an explanation tree. A ground explanation, which is a full explanation of the target relation tuple, is then generalization to get the rule. If the created rule does not cover any negative examples, then it is a valid rule. FORGE adopts the idea of *gain* function used in FOIL to guide the expansion of the explanation tree. This heuristic approach counts all positive and negative examples derived from the potential rule. To find all positive and negative examples consumes a lot of time, and we did not find it useful.

Our learning system also uses the explanation-based learning approach as FORGE's, but in a more efficient way. Also, we consider that the concept to be discovered may consist of one or more subconcepts. Therefore, the target relation tuples are not used to form the explanation. The recursive rules are formed in an indirect way via the absorption operation, which enables our learning system to deal with very complex recursive rules.

CHAPTER III

OUR LEARNING SYSTEM

We assume that the input to the learning program consists of a set of one or more base relations and a target relation, where each relation is represented by a set of tuples (of constants). The goal of the learning algorithm is to obtain one or more Horn-clauses (rules) of the form $L \leftarrow L_1, L_2, \cdots, L_k$, where L is a literal involving the target relation and each L_j involves either the target relation or a base relation. In our learning system, L may be a literal involving a new concept and the same is true for L_j.

3.1 Concept Discovery

The rules generated by the learning systems from the given base relations and target relation can be considered as a concept expressed by a set of rules. In the FOIL and FORGE systems, the target concept contains no subconcept expressed as a set of recursive rules. However, in a lot of situations, the target concept to be discovered may consist of several different subconcepts each being expressed as a set of recursive rules. For example, for the membership concept, we consider each item in a list as a member of the list and the concept can be represented as a set of rules as follows,

(3.1.a) member(X, Y) ← comp(Y, X, Z).

(3.1.b) member(X, Y) ← member(X, Z), comp(Y, W, Z).

where member is the target relation and comp is the base relation as given in Table 4.8. However, if the target concept consists of the first and the last members, it really consists of two subconcepts, the two specific members in the list. If both subconcepts can be represented by non-recursive rules, then the intermediate predicate is not required. However, if one of the subconcepts must be represented by a set of recursive rules which does not cover all the target tuples, then without introducing an intermediate predicate for this

recursive subconcept, the target concept cannot be discovered. Consider the first and last member relation as shown in Table 4.9. The rules generated with the intermediate subconcept *lastmember* introduced are given below:

(3.2.a) f_l_member(X, Y) ← comp(Y, X, Z).

(3.2.b) f_l_member(X, Y) ← lastmember(X, Y).

(3.2.c) lastmember(X, Y) ← comp(Z, X, W), null(W), comp(Y, U, Z).

(3.2.d) lastmember(X, Y) ← lastmember(X, Z), comp(Y, W, Z).

where null is the additional base relation. The rule (3.2.a) covers the first membership and the rules (3.2.b)-(3.2.d) cover the last membership. The subconcept *lastmember* does not include the cases covered by the rule (3.2.a), where the list contains one item only.

Therefore, if the target concept consists two or more subconcepts, and at least one of the subconcepts must be represented as a set of recursive rules, then at least one intermediate predicate is required to be introduced in order to discover the target concept.

In our system, we assume that the target concept consists of a number of subconcepts. We generate the rule by explaining one of the target tuples. Once a valid rule is generated, we consider it as a subconcept of the target concept. This subconcept contains only one non-recursive rule, and we consider it as a first level of subconcept. However, if no valid rule can be found for this target tuple, we will consider it as a fact, and the fact is treated as level zero. Since a non-recursive rule will not interfere with other rules via recursion, the new predicate will not be introduced. Two first levels of subconcepts can employ the union and absorption operation (these operations will be explained later). If the absorption process is success, we will create a new subconcept which is the superset of these two first level of subconcepts. The new subconcept is recursive and is considered as a second level of subconcept. A new predicate will be introduced for these new recursive rules to isolate them from other rules in order to prevent unnecessary mutual

recursion. Two second levels of subconcept will be unioned to form a third level of subconcept if it is valid and the coverage can be enlarged. The created third level of subconcept is the superset of these two second level of subconcepts, and, similarly, a new predicate will be introduced for this new recursive rules. The process of union will go on hierarchically until every target tuple is covered or the union operation fails. If the union operation cannot be performed further and there is one or more target tuples not covered, the next target tuple will be explained and the same process will be repeated. After each target tuple is covered by those subconcepts, the redundant subconcepts will be eliminated, and the rules left are the final rules. The construction of different level of rules is summarized in Table 3.1.

Table 3.1. Construction of Rules in Different Levels.

Level	Nature	Way of Construction
0	a fact	A target tuple which is unable to form a valid rule
1	a non-recursive rule	Explaining a target tuple by base tuples
2	a set of recursive rules	Union two rules in level 1 and perform absorption
n (n>2)	a set of recursive rules	Union two sets of rules in level (n-1)

In Figure 3.1, we show how the target concept can be represented by the subconcepts created in our system. Each tuple in the target relation is represented as a •. A, B, C, and D are the subconcepts of target concept T. A, B, and D are the first level of subconcepts and D is the second level of subconcept which is the superset of A and B. The target concept can be represented by the subconcepts C and D together. Subconcept A and B can be eliminated because they are subsets of C.

3.2 Ground Explanation and Valid Rule

A *ground explanation* is basically a ground form of a potential rule. For example, "t(a, c) ← p(a, b), q(b, c)" is a ground explanation of the target relation tuple t(a, c) in

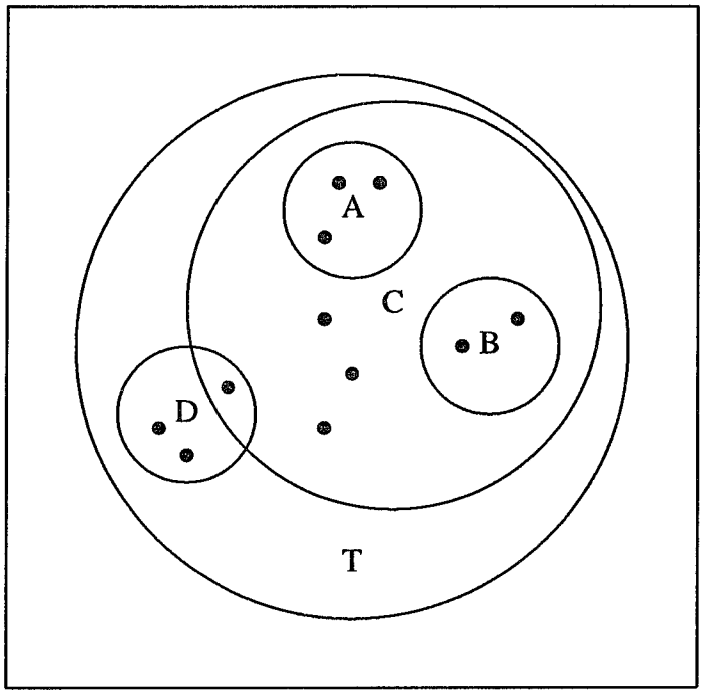

Figure 3.1. A Concept Discovery Diagram. T: Target concept,
A, B, C, D: Subconcepts, •: Target tuple.

terms of the base relation tuples p(a, b) and q(b, c), where each tuple on the right side has an argument in common with the head tuple or a previous tuple in the body. We view this explanation as a ground form of the potential rule "t(X, Y) ← p(X, W), q(W, Y)", which is obtained by replacing each constant in the explanation by a distinct variable. A rule is called a *valid* rule if it covers only positive tuples of the relation in the head of the rule.

We point out that, unlike in FORGE, we restrict the right side of an explanation to contain only base relation tuples. The construction of recursive rules is handled by use of absorption. For brevity, we will require that each argument in the head target tuple appears at least once in the body. Such explanations are called *full* [4].

Assume the target tuple to be explained is t(a, c) and the only base relation tuples are {p(a, b), p(a, c), q(b, c)}. All possible explanations of t(a, c) will form an explanation tree. Consider each tuple in a node as a label of the node. The path from the root to a node corresponds to an explanation. The explanation tree for the target tuple t(a, c) is

shown in Figure 3.2. The tuple with f tag indicates that the explanation is full and is considered as a ground explanation.

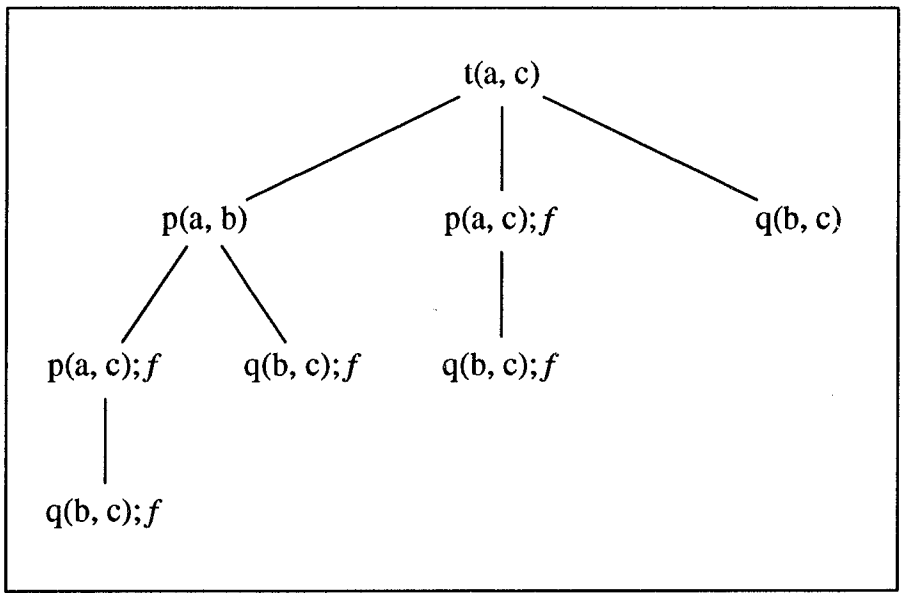

Figure 3.2. An Example of Explanation Tree.

As in [4], we shall consider shorter explanations (with fewer literals on the right side) first in order to obtain the simplest valid rules. We thus expand the explanation tree in breadth-first fashion. Unlike FOIL and FORGE, if a potential rule is found to cover a negative example, we discard the rule, and consider the next ground explanation. Once a valid rule is found, the expansion process is stopped. Therefore, we do not always expand the whole tree. If all ground explanations cannot produce a valid rule, then the target relation tuple is considered as a *fact*. We consider the rule, which is a non-recursive rule, obtained here as the first level of the rules. In FOIL and FORGE, all positive and negative examples covered by a rule are determined to compute its gain function which is then used as a guide to find the future potential rule. In our experiments, we did not find the gain function to be particular useful, and thus we do not make use of the gain function. This also helps to reduce the computation by an order of magnitude.

The algorithm to obtain a valid rule or a fact from a target relation tuple is described as follows:

Input: A target relation tuple and a set of base relations. Each base relation is specified by a set of tuples.

Output: A valid rule or a fact in the form of the first-order Horn-clause.

Algorithm:

1. Let the target relation tuple be the root of the explanation tree.
2. Each node has three sets associated with it: the first one is called excluded set, the second one is called the unexplained set, and the third one is called the link set. For the root node, the excluded set is an empty set; the unexplained set and the link set consist of all the constants of the label tuple.
3. Create a set, called the children set, which is the collection of all base tuples containing at least one of the constants in the link set minus the tuples in the excluded set.
4. If the children set is not empty, then expand the tree from left to right, with each tuple in the children set becoming one of the children of the current node.
5. For each new node, add its label tuple to the excluded set, which is inherited from its very next left sibling or its parent if there is no left sibling. Remove the label tuple's constants from the unexplained set, which is inherited from its parent. Add the label tuple's constants to the link set, which is inherited from its parent.
6. If the unexplained set is empty, then consider the path from the root to the node as a ground explanation. Replace each constant in the ground explanation by a

distinct variable to form a rule. If the rule formed is valid, then stop the explanation process.

7. If a valid rule is not found, repeat the expansion process from step 3 to step 6 by a breadth-first fashion.

8. If every children set in the leaves is empty, then consider the target relation tuple as a fact.

3.3 Linear Ordering of Tuples of Target Relation

FORGE uses a linear ordering on the target tuples to select the simplest tuples first in order to construct rules that cover them. This tends to keep the rules short and is in agreement with the notion of recursive rules which builds on simpler "base cases". The method given in [4] for determining a linear ordering works well when the objects of the universe are themselves structured (as in a list, list of lists, etc.), but it does not work well otherwise. We describe below a different heuristic for determining this ordering in such cases, and we will use this method for ordering the target tuples in our examples.

Let C(j, a) be the count of the number of occurrences of the constant 'a' as the j^{th} argument in the target tuples. Table 3.2 shows the value of c(j, x), given the set of target tuples {t(a, b), t(a, c), t(b, c), t(a, d)}. for each target tuple t(a_1, a_2, ···, a_k), we first associate the tuple of counts C(t) = (n_1, n_2, ···, n_k), where n_j = C(j, a_j). We then let C′(t) be the count-tuple obtained by rearranging the numbers n_j in non-decreasing order. Finally, we order the target tuples t according to the lexicographic ordering of C′(t). Table 3.3 shows the count-tuples C(t) and C′(t) for the data in Table 3.2. This gives the ordering: t(b, c) < t(a, b) < t(a, d) < t(a, c); the positions of the second and the third tuples are interchangeable. The purpose of using this linear ordering is to find the simplest tuple in order to find a shorter rule for concept t(X, Y). Since constant a appears three times in

the first position, some of the tuples which contain a in the first position must associate with recursive rules. The tuple t(b, c) is the simplest tuple and would be considered first in building a rule for the concept t(X, Y). We will show the advantage of using this ordering in an example given later.

Table 3.2. C(j, a) for the Set of Target Tuples {t(a, b), t(a, c), t(b, c), t(a, d)}.

C(1, a) = 3	C(2, a) = 0
C(1, b) = 1	C(2, b) = 1
C(1, c) = 0	C(2, c) = 2
C(1, d) = 0	C(2, d) = 1

Table 3.3. The Count-tuples C(t) and C′(t) for the Data in Table 3.2.

Tuple t	C(t)	C′(t)	Rank of t
t(a, b)	(3, 1)	(1, 3)	2
t(a, c)	(3, 2)	(2, 3)	4
t(b, c)	(1, 2)	(1, 2)	1
t(a, d)	(3, 1)	(1, 3)	3

The ordering obtained in this way is position sensitive. (A simplest method would be to set C(a) be the sum of C(j, a) for all j, then order the tuples t(a_1, a_2, ···, a_k) according to the sum C(a_1) + C(a_2) + ··· + C(a_k).)

3.4 Create New Concepts

Each new concept created by our learning algorithm will be a subconcept of the target concept in the sense that each tuple belonging to a new concept will also be a tuple of the target concept. The new subconcept is represented as a new predicate, and there are two ways in which a new predicate is created by our algorithm.

When a new non-recursive rule are formed, we try to combine them with the existing non-recursive rule to create new recursive rules and enlarge the coverage. If rules

(3.3) $t(X, Y) \leftarrow p(X, Y).$

have been obtained previously and new rules

(3.4) $t(X, Y) \leftarrow p(X, W), q(W, Y).$

are formed, we *union* these two rules together and create a new predicate *n1* for this set of rules. The purpose of unioning these two together is trying to use them to create new recursive rules. The new rules after union are given in the following,

(3.5.a) $t(X, Y) \leftarrow n1(X, Y).$

(3.5.b) $n1(X, Y) \leftarrow p(X, Y).$

(3.5.c) $n1(X, Y) \leftarrow p(X, W), q(W, Y).$

Since n1(X, W) is implied by p(X, W) as given in (3.5.b), p(X,W) will be replaced by n1(X, W) which can be viewed as the absorption operation in the inversion of the resolution [17, 27-29]. The new rules which replace rules (3.5.a)-(3.5.c) are

(3.6.a) $t(X, Y) \leftarrow n1(X, Y).$

(3.6.b) $n1(X, Y) \leftarrow p(X, Y).$

(3.6.c) $n1(X, Y) \leftarrow n1(X, W), q(W, Y).$

Rules (3.6.b)-(3.6.c) are recursive and they will be tested to see whether they are valid rules, which will not create negative example via (3.6.a). If they are, then they will be kept as the second level of the rules, which are recursive, and will be used to build an upper level of recursive rules. Otherwise, these rules will be discarded. The absorption operation is a generalization of two non-recursive rules. The shorter rule (with less number of literals in the right side) is used to generalize the longer rule, and the resulting rules are recursive. If two rules have equal length, then absorption will not be performed.

Second, we may *union* two sets of recursive rules at the same level. Combine two new predicates n1(X, Y) and n2(X, Y) by forming a union so that the resulting concept

n3(X, Y) contains both n1(X, Y) and n2(X, Y) as its subconcepts. This is illustrated below. The reader will notice that the most interesting case of forming the union is when both n1(X, Y) and n2(X, Y) have recursive rules. if any two recursive predicates are formed, a new predicate will be used to test if there is mutual recursion between these two predicates. Suppose we have following rules involving n1(X, Y) and n2(X, Y):

(3.7.a) t(X, Y) ← n1(X, Y).

(3.7.b) n1(X, Y) ← p(X, Y).

(3.7.c) n1(X, Y) ← p(X, W), n1(W, Y).

(3.8.a) t(X, Y) ← n2(X, Y).

(3.8.b) n2(X, Y) ← q(X, Y).

(3.8.c) n2(X, Y) ← q(X, W), n2(W, Y).

Then we create a new predicate n3(X, Y), together with the rules,

(3.9.a) t(X, Y) ← n3(X, Y).

(3.9.b) n3(X, Y) ← p(X, Y).

(3.9.c) n3(X, Y) ← p(X, W), n3(W, Y).

(3.9.d) n3(X, Y) ← q(X, Y).

(3.9.e) n3(X, Y) ← q(X, W), n3(W, Y).

If (3.9.a)-(3.9.e) are valid rules, i.e., they do not cover negative examples via (3.9.a), and if they have larger coverage than both rules (3.7.a)-(3.7.c) and rules (3.8.a)-(3.8.c), then they will be kept; otherwise, they will be removed. The new concept described by n3(X, Y) is a generalization of n1(X, Y) and n2(X, Y), and the rules (3.9.a)-(3.9.c) generalizes rules (3.7.a)-(3.8.c).

We may consider n1(X, Y) and n2(X, Y) as second level recursive predicates, and n3(X, Y) as a third level recursive predicate. If there are two recursive predicates in the third level, then we will try to create a new predicate in the fourth level to test their mutual recursiveness, and so on. Therefore, we build the predicate in a hierarchical way so that it can deal with more complicated recursive conditions.

The reason that we create the new predicate in this learning system is to try to isolate the recursive rules, which are created in some stage, from other rules, so that the recursive rules will not have interaction (mutual recursiveness) with other rules. If the interaction between one recursive rules and the other recursive rules is tested to be valid, then a superset of the recursive rules will be formed. If the target concept does not need a subconcept, then the new predicates created by our system will be eliminated, which will be described in the next section.

There are two operations, union and absorption, involved in our learning algorithm for creating a new subconcept. The union operation combines two sets of rules together and forms a new subconcept. If these rules to be united are in the first level, then the union will not increase the coverage. However, if they are higher than the first level, the union operation may enlarge the coverage due to mutual recursion. The absorption operation will be applied only after two non-recursive rules have been united together. The absorption operation creates a recursion in the rule set, and therefore it will usually enlarge the coverage.

3.5 Elimination of Predicates

If all rules have been formed, i.e., all positive examples of the target relation have been covered by the rules, then we start the elimination process. For every newly created rule, there is a coverage, *COV*, associated with it. COV_{rule1} contains a set of positive tuples of the target relation which are covered by the rule1. If COV_{rule1} is a subset of the

COV_{rule2} then rule1 will be eliminated. If there is only one new predicate left after the elimination process, then obviously, it can be replaced by the target predicate. The second step is to try to remove one rule at a time from the recursive rules and check if the coverage remains the same. If the coverage remains the same, then the rule is extraneous and can be removed.

To eliminate the extraneous rules, we separate the rules into two parts: the recursive part which contain head predicate in the right side, and the non-recursive part. We eliminate the recursive part first and then the non-recursive part. If all of the recursive part can be eliminated, then the rules become non-recursive, and the new predicates which associate with this non-recursive rules can be eliminated. If we cannot eliminate all of the recursive part, but all of the non-recursive rules are eliminated, then the inverse of the absorption operation must be performed. Suppose we have the rules as in (3.7.a)-(3.7.c), rule (3.7.c) is considered as a recursive part and rule (3.7.b) is considered as a non-recursive part. If rule (3.7.c) can be eliminated, then n1 will be removed, and the new rule after we eliminate the non-recursive part is:

(3.10) $\quad t(X, Y) \leftarrow p(X, Y).$

If rule (3.7.c) can not be eliminated, but rule (3.7.b) can be eliminated, the new rule will be:

(3.11) $\quad t(X, Y) \leftarrow p(X, W), p(W, Y).$

In both cases, the rule become non-recursive and the new predicate is removed.

For example, if the final rules formed are the rules (3.7.a)-(3.9.c) as given above, then COV_{n1} and COV_{n2} are the subsets of COV_{n3}, where n1, n2 and n3 indicate all rules associated with them, respectively. Therefore, predicates n1 and n2 will be eliminated. Since there is only one predicate, n3, left, it will be replaced by predicate t. If there is no

extraneous rule which can be removed by not changing the coverage, then the final rules after elimination are as follows:

(3.12.a) $t(X, Y) \leftarrow p(X, Y)$.

(3.12.b) $t(X, Y) \leftarrow p(X, W), t(W, Y)$.

(3.12.c) $t(X, Y) \leftarrow q(X, Y)$.

(3.12.d) $t(X, Y) \leftarrow q(X, W), t(W, Y)$.

3.6 Rule Learning Algorithm with Concept Discovery

We briefly describe below the main steps in our algorithm for constructing the intermediate concepts and the rules involving them.

Input: A target relation and a set of base relations. Each relation is specified by a set of tuples.

Output: A set of valid rules in the form of the first-order Horn-clauses which together cover all tuples of the target relation.

Algorithm:

1. Let T be the tuples of the target relation ordered linearly as described in Section 2.

2 Choose the first tuple $t \in T$ which is not yet covered by the current set of rules. Obtain a shortest valid rule for t as in FORGE [4], without using the target relation on the right side of a rule. If there is no valid rule for t with more than one literal in the body, then t is made into a fact. Eliminate from T those target tuples which are covered by the rule created. If T is empty, then go to step (6).

3. Combine the rule obtained in step (2) with similar non-recursive rules obtained from previous applications of step (2) via union, and then generalize it by

absorption (see Section 3.4), calling the new concept newp. Note that newp is a subconcept of the target concept. If absorption is not applicable, then discard newp. Eliminate from T those target tuples which are covered by the rules created. If T is empty, then go to step (6).

4. If the absorption operation is successful, then combine newp with other recursive rules obtained in previous successful applications of step (3) using union to form rules involving mutual recursion (see Section 3.3), calling the new concept newp′. If newp′ does not lead to an increased coverage of the target tuples, then discard newp′. Note that a similar test is not performed in step (3) because its main goal was to form recursive rules. Also, two concepts are combined by union in steps (3) and (4) only if they are at the same level as described in Section 3.3. The predicate newp′ is of one level higher than newp, and similarly newp is one level higher than rule obtained in step (2). Eliminate from T those target tuples which are covered by the rules created. If T is empty, then go to step (6).

5. Apply step (4) as long as possible for successively higher levels. Eliminate from T those target tuples which are covered by the rules created in steps. If T is not empty, then go to step (2).

6. Eliminate the redundant predicates to get the final rules (see Section 3.5).

The following completeness theorem is now immediate because of the fact that each subconcept newp created in step (3) of our algorithm for the various target tuples t are potentially combined in step (4) with other similar subconcepts.

Theorem 1. If a target concept can be described by rules which involve recursion on one or more subconcepts or the target concept itself, then our algorithm will find at least one such set of rules. In this sense, our algorithm is complete. ♣

If there are many such subconcepts and their corresponding rule-sets, then it is not clear to us at this point whether our algorithm will be able to determine those alternatives by choosing different ordering of the tuples in step (1).

Theorem 2. If the given target relation has n tuples, then at most n/2 new predicates (subconcepts) will be introduced by our algorithm in the final rules.

Proof: Since each non-recursive rule covers at least one target relation tuple, and the second level of rules are the superset of two non-recursive rules, They cover at least two target relation tuples. For the upper level of recursive rules, the coverage is at least one more tuple than the very next level of recursive rules. Otherwise, they will not be created. If one of the new predicates has coverage overlapped with other new predicates, then it covers at least two target relation tuples which have not been covered by other predicates. Otherwise, it is redundant or degrades to a non-recursive rule and will be eliminated. Since every new predicate covers at least two target relation tuples which is not overlapped with other predicates, the final rules contains at most n/2 new predicates. ♣

CHAPTER IV

LEARNING EXAMPLES

Several examples will be shown here to demonstrate how to apply our learning algorithm to obtain the rules. The first two examples are artificially constructed in order to give a clear picture of our learning algorithm at each stage. The others are realistic examples to show how well our algorithm can be used to deal with some problems in the real world.

4.1 Example 1

Suppose that the base relation corresponds to the links of an acyclic digraph. Moreover, suppose that the target relation consists of a special form of reachability relation in that digraph, where one can go forward as many steps (arcs) as possible but one is allowed to take at most one backward step. Such a reachability relation can be considered as a special case of a family chain, where a person knows all his ancestors but knows only his children. An example is given in Figure 4.1. The tuples of the base relation and the target relation are shown in Table 4.1.

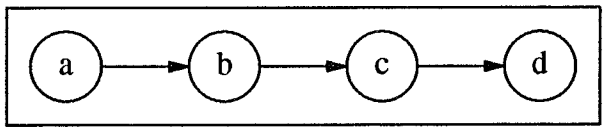

Figure 4.1. An Acyclic Digraph.

Table 4.1. Base Relation and Target Relation of Example 1.

Base Relation: link	Target Relation: reach		
(a, b)	(a, b)	(b, a)	(a, c)
(b, c)	(b, c)	(c, b)	(b, d)
(c, d)	(c, d)	(d, c)	(a, d)

The first step is to arrange all tuples of the target relation in a linear order. The count-tuple of the target tuple is given in Table 4.2, and the ordering of the target tuples is given in Table 4.3.

Table 4.2. C(j, a) for Target Tuples of Example 1

First Argument	Second Argument
C(1, a) = 3	C(2, a) = 1
C(1, b) = 3	C(2, b) = 2
C(1, c) = 2	C(2, c) = 3
C(1, d) = 1	C(2, d) = 3

Table 4.3 The Ordering of the Target Tuples of Example 1

Target Tuple	C(t)	C′(t)	Order of Tuple
reach(a, b)	(3, 2)	(2, 3)	4
reach(b, c)	(3, 3)	(3, 3)	6
reach(c, d)	(2, 3)	(2, 3)	5
reach(b, a)	(3, 1)	(1, 3)	1
reach(c, b)	(2, 2)	(2, 2)	3
reach(d, c)	(1, 3)	(1, 3)	2
reach(a, c)	(3, 3)	(3, 3)	7
reach(b, d)	(3, 3)	(3, 3)	8
reach(a, d)	(3, 3)	(3, 3)	9

Therefore, the order of the tuples, *T*, will be {reach(b, a), reach(d, c), reach(c, b), reach(a, b), reach(c, d), reach(b, c), reach(a, c), reach(b, d), reach(a, d)}.

The first target tuple reach(b, a) is explained by reach(b, a) ← link(b, a). Thus, the shortest valid rule we get is

(4.1) reach(X, Y) ← link(Y, X).

If we do not arrange the tuples in this linear order, we may select reach(a, d) as the first tuple to be explained. Then, the ground explanation will be reach(a, d) ← link(a,b), link(b, c), link(c, d). The corresponding valid rule is reach(X, Y) ← link(X, W), link(W, U), link(U, Y). Although the rule is valid, it will become redundant later and will be eliminated in the final phase of our algorithm. Therefore, by using this linear

ordering, we avoid creating redundant rules like this and save considerable processing time.

The tuples {reach(b, a), reach(d, c), reach(c, b)} covered by (4.1) are removed from *T*, *T* becomes {reach(a, b), reach(c, d), reach(b, c), reach(a, c), reach(b, d), reach(a, d)}. Since *T* is not empty, the next tuple chosen is reach(a, b) and the shortest valid rule we get is

(4.2) reach(X, Y) ← link(X, Y).

The tuples {reach(a, b), reach(c, d), reach(b, c)} covered by (4.2) are removed from *T*, giving *T* {reach(a, c), reach(b, d), reach(a, d)}. Although we have two non-recursive rules, the absorption operation will not be processed because both rules have the same number of literals.

Since *T* is not empty, the same process will be repeated. The tuple chosen is reach(a, c) and we obtain the following valid rule from the shortest full explanation reach(a, c) ← link(a, b), link(b, c).

(4.3) reach(X, Y) ← link(X, W), link(W, Y).

The coverage of rule (4.3) is {reach(b, d), reach(a, c)}.

Unioning rule (4.1) and (4.3), we name the new predicate reach1, and so we can create the following rules:

(4.4.a) reach(X, Y) ← reach1(X, Y).

(4.4.b) reach1(X, Y) ← link(Y, X).

(4.4.c) reach1(X, Y) ← link(X, W), link(W, Y).

We can now make the above rules recursive by replacing link(X, W) by reach1(W, X).

(4.5.a) reach(X, Y) ← reach1(X, Y).

(4.5.b) reach1(X, Y) ← link(Y, X).

(4.5.c) reach1(X, Y) ← reach1(W, X), link(W, Y).

The coverage of rules (4.5.a)-(4.5.c) is {reach(b, a), reach(d, c) reach(c, b), reach(a, c), reach(b, d)}.

Similarly, unioning rule (4.1) and (4.3), we can name the new predicate reach2, and then replace link(W, Y) by reach2(Y, W) to get the following rules:

(4.6.a) reach(X, Y) ← reach2(X, Y).

(4.6.b) reach2(X, Y) ← link(Y, X).

(4.6.c) reach2(X, Y) ← link(X, W), reach2(Y, W).

Rules (4.6.a)-(4.6.c) are valid rules and have the same coverage as rules (4.4.a)-(4.4.c). The union of rules (4.5.a)-(4.5.c) and rules (4.6.a)-(4.6.c) will not be performed because the resulting rules will not enlarge the coverage.

After removing these covered tuples from *T*, the only tuple remaining is reach(a, d). Since there are two sets of recursive rules in the second level, we will try to union them together to get a new set of rules. However, the union will not enlarge the coverage and this new set of rules will not be created.

Unioning rule (4.2) and (4.3), we name the new predicate reach3, then replace link(X, W) by reach3(X, W) to get the following rules:

(4.7.a) reach(X, Y) ← reach3(X, Y).

(4.7.b) reach3(X, Y) ← link(X, Y).

(4.7.c) reach3(X, Y) ← reach3(X, W), link(W, Y).

The coverage of rules (4.7.a)-(4.7.c) is {reach(a, b), reach(c, d) reach(b, c), reach(a, c),

reach(b, d), reach(a, d)}. The rules (4.7.a)-(4.7.c) cover the last remaining tuple reach(a, d) in the *T*. Since the *T* is empty, the rules finding procedure is completed.

The final phase is to remove the redundant rules, if any. Since the coverage of rule (4.1), rule (4.3), and rules (4.6.a)-(4.6.c) form the subset of the coverage of rules (4.5.a)-(4.5.c), while the coverage of the rule (4.2) and rule (4.3) are the subsets of the coverage of rules (4.7.a)-(4.7.c), all the rules associated with reach2, and rule (4.1)-(4.3) will be eliminated. Therefore, the remaining rules are (4.4.a)-(4.4.c) and (4.7.a)-(4.7.c). By removing one rule at a time, we find rule (4.4.c) is redundant and can be removed. After removing (4.4.c), reach1 is associated with only the non-recursive rule (4.4.a)-(4.4.b), so rules (4.4.a)-(4.4.b) will be replaced by (4.8.d). The final rules are given in the following:

(4.8.a) reach(X, Y) ← reach3(X, Y).

(4.8.b) reach3(X, Y) ← link(X, Y).

(4.8.c) reach3(X, Y) ← reach3(X, W), link(W, Y).

(4.8.d) reach(X, Y) ← link(Y, X).

The rules (4.8.a)-(4.8.c) indicate that reachability can go forward in a arbitrary step while rule (4.8.d) indicates that reachability can go backward only one step as per our definition in the problem statement above.

4.2 Example 2

A second example will be considered here. The given input of base relations and target relation are shown in Table 4.4. The names of base relations, l-link, c-link, and r-link are the abbreviations of left-link, center-link, and right-link, respectively. The base relations may be visualized as the digraph in Figure 4.2. The target relation consists of

special reachability relations in that digraph. The reachability relations can go forward with any combination of r-links and l-links, or with a sequence of c-links.

Table 4.4. Base Relations and Target Relation of Example 2.

Three Base Relations:			Target Relation:				
l-link	c-link	r-link	reach				
(a, b)	(a, c)	(a, d)	(a, b)	(a, c)	(a, d)	(a, e)	(a, m)
(b, e)	(b, f)	(b, g)	(b, e)	(b, f)	(b, g)	(a, g)	(a, t)
(e, n)	(e, o)	(e, p)	(e, n)	(e, o)	(e, p)	(a, n)	(a, v)
(c, h)	(c, i)	(c, j)	(c, h)	(c, i)	(c, j)	(a, p)	(b, n)
(i, q)	(i, r)	(i, s)	(i, q)	(i, r)	(i, s)	(a, i)	(b, p)
(d, k)	(d, l)	(d, m)	(d, k)	(d, l)	(d, m)	(a, r)	(c, r)
(m, t)	(m, u)	(m, v)	(m, t)	(m, u)	(m, v)	(a, k)	(d, t)
			(d, v)				

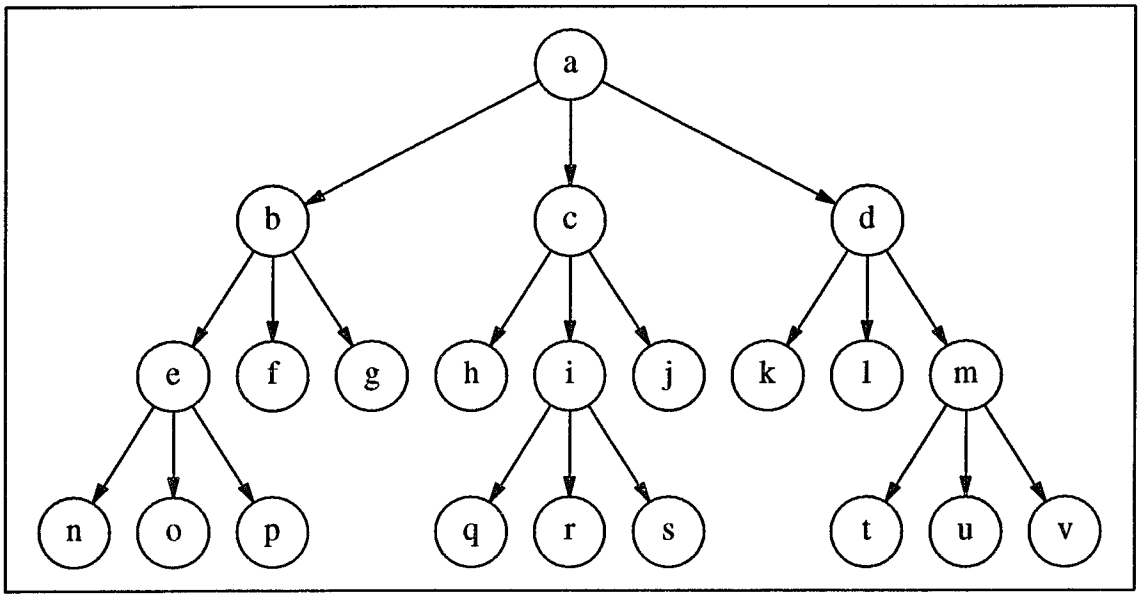

Figure 4.2. Digraph of Examples 2 and 3.

The first step is to arrange the tuples of the target relation in a linear order. The count of occurrences of the constants is given in Table 4.5, and the order of the target tuples is given in Table 4.6. Therefore, the order of the tuples, T, will be {reach(i, q), reach(e, o), reach(m, u), ··· , reach(a, v)}. The first tuple of the T is reach(i, q), which will be chosen first to be explained by the base relations. The shortest rule we get is

Table 4.5 C(j, a) for Target Tuples of Example 2

First Argument	Second Argument
C(1, a) = 13	C(2, a) = 0
C(1, b) = 5	C(2, b) = 1
C(1, c) = 4	C(2, c) = 1
C(1, d) = 5	C(2, d) = 1
C(1, e) = 3	C(2, e) = 2
C(1, f) = 0	C(2, f) = 1
C(1, g) = 0	C(2, g) = 2
C(1, h) = 0	C(2, h) = 1
C(1, i) = 3	C(2, i) = 2
C(1, j) = 0	C(2, j) = 1
C(1, k) = 0	C(2, k) = 2
C(1, l) = 0	C(2, l) = 1
C(1, m) = 3	C(2, m) = 2
C(1, n) = 0	C(2, n) = 3
C(1, o) = 0	C(2, o) = 1
C(1, p) = 0	C(2, p) = 3
C(1, q) = 0	C(2, q) = 1
C(1, r) = 0	C(2, r) = 3
C(1, s) = 0	C(2, s) = 1
C(1, t) = 0	C(2, t) = 3
C(1, u) = 0	C(2, u) = 1
C(1, v) = 0	C(2, v) = 3

(4.9) reach(X, Y) ← l-link(X, Y).

The coverage of rule (4.9) is {reach(a, b), reach(b, e), reach(e, n), reach(c, h), reach(i, q), reach(d, k), reach(m, t)}, and those tuples will be removed from *T*. Thus, *T* is now {reach(e, o), reach(m, u), reach(i, s), ···, reach(a, v)}.

Since *T* is not empty, the next tuple chosen is reach(e, o) and the shortest rule is

(4.10) reach(X, Y) ← c-link(X, Y).

The coverage of rule (4.10) is {reach(a, c), reach(b, f), reach(e, o), reach(c, i), reach(i, r), reach(d, l), reach(m, u)}, and those tuples will be removed from *T*. Thus, *T* is now {reach(i, s), reach(c, j), reach(a, d), ···, reach(a, v)}. These two non-recursive rules have same number of literals in the right side, so the absorption operation will not be performed.

Table 4.6 The Ordering of the Target Tuples of Example 2

Target Tuple	C(t)	C′(t)	Order of Tuple
reach(a, b)	(13, 1)	(1, 13)	9
reach(b, e)	(5, 2)	(2, 5)	13
reach(e, n)	(3, 3)	(3, 3)	22
reach(c, h)	(4, 1)	(1, 4)	5
reach(i, q)	(3, 1)	(1, 3)	1
reach(d, k)	(5, 2)	(2, 5)	14
reach(m, t)	(3, 3)	(3, 3)	23
reach(a, c)	(13, 1)	(1, 13)	10
reach(b, f)	(5, 1)	(1, 5)	7
reach(e, o)	(3, 1)	(1, 3)	2
reach(c, i)	(4, 2)	(2, 4)	12
reach(i, r)	(3, 3)	(3, 3)	24
reach(d, l)	(5, 1)	(1, 5)	8
reach(m, u)	(3, 1)	(1, 3)	3
reach(a, d)	(13, 1)	(1, 13)	11
reach(b, g)	(5, 2)	(2, 5)	15
reach(e, p)	(3, 3)	(3, 3)	25
reach(c, j)	(4, 1)	(1, 4)	6
reach(i, s)	(3, 1)	(1, 3)	4
reach(d, m)	(5, 2)	(2, 5)	16
reach(m, v)	(3, 3)	(3, 3)	26
reach(a, e)	(13, 2)	(2, 13)	17
reach(a, g)	(13, 2)	(2, 13)	18
reach(a, n)	(13, 3)	(3, 13)	32
reach(a, p)	(13, 3)	(3, 13)	33
reach(a, i)	(13, 2)	(2, 13)	19
reach(a, r)	(13, 3)	(3, 13)	34
reach(a, k)	(13, 2)	(2, 13)	20
reach(a, m)	(13, 2)	(2, 13)	21
reach(a, t)	(13, 3)	(3, 13)	35
reach(a, v)	(13, 3)	(3, 13)	36
reach(b, n)	(5, 3)	(3, 5)	28
reach(b, p)	(5, 3)	(3, 5)	29
reach(c, r)	(4, 3)	(3, 4)	27
reach(d, t)	(5, 3)	(3, 5)	30
reach(d, v)	(5, 3)	(3, 5)	31

The next tuple chosen is reach(i, s) and the shortest rule is

(4.11) reach(X, Y) ← r-link(X, Y).

The coverage of rule (4.11) is {reach(a, d), reach(b, g), reach(e, p), reach(c, j), reach(i, s), reach(d, m), reach(m, v)}, and those tuples will be removed from *T*. Thus, *T* is now

{reach(a, e), reach(a, g), reach(a, i), ···, reach(a, v)}. Similarly, no absorption operation will be performed.

The next tuple chosen is reach(a, e) and the shortest rule is

(4.12) reach(X, Y) ← l-link(X, W), l-link(W, Y).

Now, we can union rule (4.12) and rule (4.9), and name the new predicate reach1. We can perform the absorption operation to make the rule recursive by replacing l-link(X, W) by reach1(X, W). The rules created are

(4.13.a) reach(X, Y) ← reach1(X, Y).

(4.13.b) reach1(X, Y) ← l-link(X, Y).

(4.13.c) reach1(X, Y) ← reach1(X, W), l-link(W, Y).

Similarly, we can create the following rules where l-link(W, Y) is replaced by reach2(W, Y):

(4.14.a) reach(X, Y) ← reach2(X, Y).

(4.14.b) reach2(X, Y) ← l-link(X, Y).

(4.14.c) reach2(X, Y) ← l-link(X, W), reach2(W, Y).

Both set of rules are valid, recursive, and have the same coverage. We consider both of them as rules in level 2. The additional set of the tuples covered by them is {reach(a, e), reach(a, n), reach(b, n)}, and those tuples will be removed from *T*. The new reduced *T* is {reach(a, g), reach(a, i), reach(a, k), ···, reach(a, v)}.

Since *T* is not empty, the next tuple chosen is reach(a, g), and the shortest rule is

(4.15) reach(X, Y) ← l-link(X, W), r-link(W, Y).

The coverage of rule (4.15) is {reach(a, g), reach(b, p)}. We can union rule (4.9) and (4.15) and then perform the absorption operation. The new rules are

(4.16.a) reach(X, Y) ← reach3(X, Y).

(4.16.b) reach3(X, Y) ← l-link(X, Y).

(4.16.c) reach3(X, Y) ← reach3(X, W), r-link(W, Y).

The new rules are valid and recursive. Although these new rules do not increase the coverage, we will keep them for the sake of creating more complex recursive rules later.

We can union rules (4.16.a)-(4.16.c) with rules (4.13.a)-(4.13.c) since both sets of the rules are considered as level 2. The new rules are

(4.17.a) reach(X, Y) ← reach4(X, Y).

(4.17.b) reach4(X, Y) ← l-link(X, Y).

(4.17.c) reach4(X, Y) ← reach4(X, W), r-link(W, Y).

(4.17.d) reach4(X, Y) ← reach4(X, W), l-link(W, Y).

Similarly, We can union rules (4.16.a)-(4.16.c) with rules (4.14.a)-(4.14.c), and the new rules are

(4.18.a) reach(X, Y) ← reach5(X, Y).

(4.18.b) reach5(X, Y) ← l-link(X, Y).

(4.18.c) reach5(X, Y) ← reach5(X, W), r-link(W, Y).

(4.18.d) reach5(X, Y) ← l-link(X, W), reach5(W, Y).

Both reach4 and reach5 are valid and considered as rules in level 3. They have same coverage, and the additional set of tuples covered by these new rules is {reach(a, p)}. We can union these two level 3 rules together. However, the coverage will be enlarged, and we abandon the union.

We can now union rule (4.11) and (4.15) and then perform the absorption operation. The new rules are

(4.19.a) reach(X, Y) ← reach6(X, Y).

(4.19.b) reach6(X, Y) ← r-link(X, Y).

(4.19.c) reach6(X, Y) ← l-link(X, W), reach6(W, Y).

These new rules are valid and are considered as rules in the level 2. Although these new rules will not increase the coverage, however, we will keep them for the sake of creating more complex recursive rules later.

We can union rules (4.19.a)-(4.19.c) with rules (4.13.a)-(4.13.c), and the new rules are

(4.20.a) reach(X, Y) ← reach7(X, Y).

(4.20.b) reach7(X, Y) ← r-link(X, Y).

(4.20.c) reach7(X, Y) ← l-link(X, W), reach7(W, Y).

(4.20.d) reach7(X, Y) ← l-link(X, Y).

(4.20.e) reach7(X, Y) ← reach7(X, W), l-link(W, Y).

These new rules are considered as rules in the level 3, and the additional set of tuples covered is {reach(a, k), reach(d, t)}.

Now, we can union the two sets of rules at level 3 to form rules at level 4. By unioning rules (4.20.a)-(4.20.e) with rules (4.17.a)-(4.17.d), or unioning rules (4.20.a)-(4.20.e) with rules (4.18.a)-(4.18.d), we create same set of rules as follows:

(4.21.a) reach(X, Y) ← reach8(X, Y).

(4.21.b) reach8(X, Y) ← r-link(X, Y).

(4.21.c) reach8(X, Y) ← l-link(X, W), reach8(W, Y).

(4.21.d) reach8(X, Y) ← l-link(X, Y).

(4.21.e) reach8(X, Y) ← reach8(X, W), l-link(W, Y).

(4.21.f) reach8(X, Y) ← l-link(X, W), reach8(W, Y).

The additional set of tuples covered by these new rules is {reach(d, v), reach(a, m), reach(a, v), reach(a, t)}.

We can also union rules (4.19.a)-(4.19.c) with rules (4.14.a)-(4.14.c), and the new rules are

(4.22.a) reach(X, Y) ← reach9(X, Y).

(4.22.b) reach9(X, Y) ← r-link(X, Y).

(4.22.c) reach9(X, Y) ← l-link(X, W), reach9(W, Y).

(4.22.d) reach9(X, Y) ← l-link(X, Y).

These new rules have more coverage than combining rules (4.19.a)-(4.19.c) and rules (4.14.a)-(4.14.c) together, and so they will be kept. However, some of the tuples covered by these new rules are overlapped with other rules. Therefore, no additional tuples can be removed form *T*.

We can now union rules (4.22.a)-(4.22.d) with rules (4.17.a)-(4.17.d). However, the new rules are the same as the rules (4.21.a)-(4.21.f). We can also union rules (4.22.a)-(4.22.d) with rules (4.18.a)-(4.18.d), or union rules (4.22.a)-(4.22.d) with rules (4.20.a)-(4.20.e); and we create the same set of rules at level 4 as follows:

(4.23.a) reach(X, Y) ← reach10(X, Y).

(4.23.b) reach10(X, Y) ← r-link(X, Y).

(4.23.c) reach10(X, Y) ← l-link(X, W), reach10(W, Y).

(4.23.d) reach10(X, Y) ← l-link(X, Y).

(4.23.e) reach10(X, Y) ← reach10(X, W), r-link(W, Y).

Although the union operation does increase the coverage, the additional tuples covered by the new rules are overlapped with other rules. Therefore, no additional tuples can be

removed form *T*. Since both rules (4.21.a)-(4.21.f) and rules (4.23.a)-(4.23.e) are at level 4, we can union these two sets of rules together. However, the union operation will not increase the coverage, so the rules obtained from the union are abandoned.

We can also union rules (4.19.a)-(4.19.c) with rules (4.16.a)-(4.16.c) since both are rules at level 2. The new rules created are at level 3, but they have the same form as rules (4.23.a)-(4.23.e). We can perform the union operation by unioning (4.23.a)-(4.23.e) with other rules at level 3. However, the only valid rules are created have the same form as rules (4.21.a)-(4.21.f), and no more union operations can be perform. After all the tuples covered by the previous rules are removed from *T*, *T* is {reach(a, i), reach(c, r), reach(a, r)}.

Since *T* is not empty, the next tuple chosen is reach(a, i), and the shortest rule is

(4.24) reach(X, Y) ← c-link(X, W), c-link(W, Y).

The coverage of the rule (4.24) is {reach(a, i), reach(c, r)}. We can union rule (4.24) and (4.10), and then perform the absorption operation. The new rules obtained are

(4.25.a) reach(X, Y) ← reach11(X, Y).

(4.25.b) reach11(X, Y) ← c-link(X, Y).

(4.25.c) reach11(X, Y) ← reach11(X, W), c-link(W, Y).

The additional set of the tuples covered by reach11 is {reach(a, r)}. After removing those tuples, *T* is empty.

The final phase is to remove the redundant rules. We will have two new predicates reach11 and reach8 left, and the final rules are

(4.26.a) reach(X, Y) ← reach8(X, Y).

(4.26.b) reach11(X, Y) ← c-link(X, Y).

(4.26.c) reach11(X, Y) ← reach2(X, W), c-link(W, Y).

(4.26.d) reach(X, Y) ← reach11(X, Y).

(4.26.e) reach8(X, Y) ← l-link(X, Y).

(4.26.f) reach8(X, Y) ← r-link(X, Y).

(4.26.g) reach8(X, Y) ← reach8(X, W), l-link(W, Y).

(4.26.h) reach8(X, Y) ← reach8(X, W), r-link(W, Y).

Rules (4.26.a)-(4.26.c) indicate that reachability can go forward in a sequence of c-links, and rules (4.26.d)-(4.26.h) indicate that reachability can go forward with any combination of l-links and r-links.

Although rules (4.13.a)-(4.13.c) have the same coverage as rules (4.14.a)-(4.14.c), we keep both sets of rules. These two rules are equivalent at this moment, they represent a reach-relation of a sequence of l-links and have same coverage. The reason we keep both sets of rules is because if other rules are formed, these two rules might union with them and form different rules. Assume we have following rules at level 2:

(4.27.a) reach(X, Y) ← reach12(X, Y).

(4.27.b) reach12(X, Y) ← r-link(X, Y).

(4.27.c) reach12(X, Y) ← reach12(X, W), l-link(W, Y).

The union of rules (4.13.a)-(4.13.c) and rules (4.27.a)-(4.27.c) is different from the union of rules (4.14.a)-(4.14.c) and rules (4.27.a)-(4.27.c). The new two sets of rules are given in the following:

(4.28.a) reach(X, Y) ← reach13(X, Y).

(4.28.b) reach13(X, Y) ← l-link(X, Y).

(4.28.c) reach13(X, Y) ← r-link(X, Y).

(4.28.d) reach13(X, Y) ← reach13(X, W), l-link(W, Y).

(4.29.a) reach(X, Y) ← reach14(X, Y).

(4.29.b) reach14(X, Y) ← l-link(X, Y).

(4.29.c) reach14(X, Y) ← r-link(X, Y).

(4.29.d) reach14(X, Y) ← reach14(X, W), l-link(W, Y).

(4.29.e) reach14(X, Y) ← l-link(X, W), reach14(W, Y).

The rules in (4.28.a)-(4.28.d) represent a reach-relation starting with either a l-link or a r-link and ending with a sequence of l-links, while the rules in (4.29.a)-(4.29.e) represent a reach-relation either starting with a sequence of l-links and/or ending with a sequence of l-links. That shows why we need to keep two sets of rules which seem to be equivalent. If we only keep one set of these two equivalent rules, we will lose the ability to form a variety of rules.

4.3 Example 3

In the previous two examples, we have considered abstract relations to show how well our learning system can perform. Here, we consider a more realistic example. Consider Figure 4.2 as a family-tree showing parent-child relationships. In particular, the tree is unordered. The base relation parent(X, Y) and the target relation guardian(X, Y) are now shown in Table 4.7. The guardian relation consists of two subconcepts: X is a guardian of Y if X is an ancestor (parent, grandparent, etc.) of Y and X is an uncle (or aunt) of Y. The final rules obtained by our learning algorithm are as follows, where (4.30.b)-(4.30.c) gives the ancestor-relationship and (4.30.d) gives the uncle-relationship. Obviously, without the ability to create the intermediate relation *ancestor*, the learning system can not obtain these final rules.

Table 4.7. Base Relation and Target Relation of Example 3.

Base Relation: parent		Target Relation: guardian						
(a, b)	(b, e)	(a, b)	(a, c)	(a, d)	(a, e)	(a, f)	(a, g)	(a, h)
(e, n)	(c, h)	(a, i)	(a, j)	(a, k)	(a, l)	(a, m)	(a, n)	(a, o)
(i, q)	(d, k)	(a, p)	(a, q)	(a, r)	(a, s)	(a, t)	(a, u)	(a, v)
(m, t)	(a, c)	(b, e)	(b, f)	(b, g)	(b, n)	(b, o)	(b, p)	(b, h)
(b, f)	(e, o)	(b, i)	(b, j)	(b, k)	(b, l)	(b, m)	(c, h)	(c, i)
(c, i)	(i, r)	(c, j)	(c, q)	(c, r)	(c, s)	(c, e)	(c, f)	(c, g)
(d, l)	(m, u)	(c, k)	(c, l)	(c, m)	(d, k)	(d, l)	(d, m)	(d, t)
(a, d)	(b, g)	(d, u)	(d, v)	(d, e)	(d, f)	(d, g)	(d, h)	(d, i)
(e, p)	(c, j)	(d, j)	(e, n)	(e, o)	(e, p)	(i, q)	(i, r)	(i, s)
(i, s)	(d, m)	(m, t)	(m, u)	(m, v)	(f, n)	(f, o)	(f, p)	(g, n)
(m, v)		(g, o)	(g, p)	(h, q)	(h, r)	(h, s)	(j, q)	(j, r)
		(j, s)	(k, t)	(k, u)	(k, v)	(l, t)	(l, u)	(l, v)

(4.30.a) guardian(X, Y) ← ancestor(X, Y).

(4.30.b) ancestor(X, Y) ← parent(X, Y).

(4.30.c) ancestor(X, Y) ← ancestor(Z, Y), parent(X, Z).

(4.30.d) guardian(X, Y) ← parent(Z, Y), parent(W, Z), parent(W, X).

We have demonstrated our new learning algorithm with three examples, none of which can be handled by FOIL and FORGE. Since the rule for the target relations in these examples requires one or more new recursive concepts, these rules cannot be found by FOIL or FORGE. For the cases which FOIL and FORGE can handle, our system will also produces similar results (rules). The intermediate predicates created in the process are eliminated in the final phase of our algorithm.

4.4 Example 4

Consider the append-relation as shown in Table 4.8, where append is the target relation and comp and null are the base relations. The literal append(X, Y, Z) indicates that Z is the new list after appending list Y to list X. The literal null(W) indicates that W is a empty list and the literal comp(P, Q, R) indicates that P is a list, Q is the head of P, and R is the remaining list after removing Q from P.

Table 4.8. Example of Append Relation.

Two Base Relations:		Target Relation: Append	
null	comp		
([])	([b, [a], d], b, [[a], d])	([], [b, [a], d], [b, [a], d])	([b], [[a], d], [b, [a], d])
	([[a], d], [a], [d])	([b, [a]], [d], [b, [a], d])	([b, [a], d], [], [b, [a], d])
	([b, [a]], b, [[a]])	([], [[a], d], [[a], d])	([[a]], [d], [[a], d])
	([[a]], [a], [])	([[a], d], [], [[a], d])	([], [b, [a]], [b, [a]])
	([a], a, [])	([b], [[a]], [b, [a]])	([b, [a]], [], [b, [a]])
	([b], b, [])	([], [[a]], [[a]])	([[a]], [], [[a]])
	([d], d, [])	([], [a], [a])	([a], [], [a])
		([], [b], [b])	([b], [], [b])
		([], [d], [d])	([d], [], [d])
		([], [], [])	

The rules we obtain are listed as follows:

(4.31.a) append(X, Y, X) ← null(Y), comp(Z, W, X).

(4.31.a) append(X, Y, Y) ← null(X), comp(Y, W, U).

(4.31.b) append(X, Y, Z) ← newapd(X, Y, Z).

(4.31.c) newapd(X, Y, Z) ← comp(Z, U, Y), comp(X, U, V), null(V).

(4.31.d) newapd(X, Y, Z) ← newapd(V, Y, U), comp(Z, W, U),
comp(X, W, V).

The rule (4.31.a) considers a special case where X is an empty list, while rules (4.31.b)-(4.31.d) consider other cases of append. The rules obtained by FOIL and FORGE are shown in (4.32.a)-(4.32.d) and (4.33.a)-(4.33.b), respectively [4, 6, 25]. The rules obtained from FORGE are the simplest. The rules obtained from FOIL are similar to the rules obtained by our learning system. The reason that FORGE will get the simplest rules is because it assumes that the target concept is a single concept. Therefore, FORGE will try to include the target relation in the body of the rules. In our case, we assume that there are multiple subconcepts. Hence, the new relation newapd is put into the body of the rules by the absorption operation.

(4.32.a) append(X, Y, Z) ← X=Z, null(Y).

(4.32.b) append(X, Y, Z) ← Y=Z, null(X).

(4.32.c) append(X, Y, Z) ← comp(Z, U, Y), comp(X, U, V), null(V).

(4.32.d) append(X, Y, Z) ← append(V, Y, U), comp(Z, W, U), comp(X, W, V).

(4.33.a) append(X, Y, Y) ← null(X), comp(Y, Z, W).

(4.33.b) append(X, Y, Z) ← append(V, Y, U), comp(Z, W, U), comp(X, W, V).

4.5 Example 5

Consider the member-relation [4, 6, 25] shown in Table 4.9, where member is the target relation while comp and null are the base relations. The literal member(X, Y) indicates that X is the member of list Y. The literals comp(P, Q, R) and null(W) have the same meaning as in the last example. The rules we obtain are the same as the rules obtained by FOIL and FORGE.

Table 4.9. Example of Member Relation.

Two Base Relations:		Target Relation: member	
null	comp		
([])	([a], a, [])	(a, [a])	(b, [b, [a], d])
	([d], d, [])	(d, [d])	([a], [b, [a], d])
	([[a], d], [a], [d])	([a], [[a], d])	(d, [b, [a], d])
	([b, [a], d], b, [[a], d])	(d, [[a], d])	

(4.34.a) member(X, Y) ← comp(Y, X, Z).

(4.34.b) member(X, Y) ← member(X, Z), comp(Y, W, Z).

4.6 Example 6.

Consider a special member relation in which only the first and the last member in the list are included in our target relation. Therefore, the target concept actually contains two subconcepts. The base relations and target relation are shown in table 4.10, which is similar to Table 4.9. The only difference between these two tables is an additional list in the Table 4.10 to make the case more interesting. Therefore, the last membership relation will have recursive rules involved.

Table 4.10. Example of First-Last Member Relation.

Two Base Relations:		Target Relation: f_l_member	
null	comp		
([])	([a], a, [])	(a, [a])	(b, [b, [a], d])
	([d], d, [])	(d, [d])	(d, [b, [a], d])
	([[a], d], [a], [d])	([a], [[a], d])	(c, [c, b, [a], d])
	([b, [a], d], b, [[a], d])	(d, [[a], d])	(d, [c, b, [a], d])
	([c, b, [a], d], c, [b, [a], d])		

The final rules we obtain are given in the following:

(4.35.a) f_l_member(X, Y) ← comp(Y, X, Z).

(4.35.b) f_l_member(X, Y) ← lastmember(X, Y).

(4.35.c) lastmember(X, Y) ← comp(Z, X, W), null(W), comp(Y, U, Z).

(4.35.d) lastmember(X, Y) ← lastmember(X, Z), comp(Y, W, Z).

Rule (4.35.a) covers the first membership and rules (4.35.b)-(4.35.d) cover the last membership. Since the subconcept lastmember involves recursive rules, the rules we obtain can not be obtained by FOIL and FORGE.

A equivalent set of rules are given in the following:

(4.36.a) f_l_member(X, Y) ← comp(Y, X, Z).

(4.36.b) f_l_member(X, Y) ← lastmember(X, Y).

(4.36.c) lastmember(X, Y) ← comp(Y, X, Z), null(Z).

(4.36.d) lastmember(X, Y) ← lastmember(X, Z), comp(Y, W, Z).

This set of rules is simpler in the sense that the rules are shorter. However, the lastmember relation has a coverage overlapped with the first member in the case of a single item list. The reason we will get the former set of rules (4.35.a)-(4.35.d) is as follows. A target tuple is explained by the base tuples to get the shortest valid rule. Once a valid rule is obtained, we stop the explanation process and remove the tuples covered by the rule from the set of target tuples, *T*. Therefore, after rule (4.35.a) is obtained, every target tuple which represents the first member of the list is removed from *T*. For the single item list, the first member is also the last member. Therefore, we can not obtain rule (4.36.c) instead of rule (4.35.c). Although we cannot guarantee that the rules we obtained are the simplest rules, we do save a lot of computation time by not checking all the valid rules and then select the best rule (since we cannot determine what is the best rule).

4.7 Implementation

Our learning algorithm is implemented in Allegro Common Lisp [31-32] running on the Unix system. The input file contains two lists. The first list is for base relation tuples, and the second list is for target relation tuples. Since there are one or more base relations, each base relation is represented as a sub-list. The name of the base relation is the first item in the sub-list, followed by a list of the tuples of that base relation. The list for the target relation tuples is in a similar form except not in a sub-list form since there is only one target relation. A typical input is shown in Table 4.11 for example 3 above.

Each global variable is enclosed by a pair of asterisks to distinguish from other variables. The global variables used in the program are explained in Table 4.12. The coding of our learning program is listed in the Appendix.

Table 4.11. A Typical Input of Our Learning Program.

```
; The input for example 5
; Base relation tuples
((null (()))
 (comp ((a) a ()) ((d) d ()) (((a) d) (a) (d)) ((b (a) d) b ((a) d)))
)
; Target relation tuples
(member (a (a)) (d (d)) ((a) ((a) d)) (d ((a) d)) (b (b (a) d)) ((a) (b (a) d)) (d (b (a) d)))
```

Table 4.12. Global Variables of Our Learning Program.

input-file	Set input file name as "test.in"
base-tuples	The first list of the input file
target-tuples	The second list of the input file
named-base-tuples	Similar to *base-tuples* except that each tuple has a base relation name in front of it
uncovered	The target tuples which are not covered by the existing rules
ui	Indicates that a variable is un-instantiated
variables	A set of variables names which start with a capital letter
new-pred	A set of new predicate names
rule-set	A set of rule pointers which point to all existing rules. Each rule pointer is a property list and has following attributes: rules: A set of rules cover: The coverage of the rules level: 0 - A fact 1 - A non-recursive rule >1 - A set of recursive rules sons: Two set of rules which form the current rules
current-level	A list of tree nodes at the current level
next-level	A list of tree nodes at the next level

CHAPTER V

DISCUSSION AND CONCLUSION

We have developed a learning system which can learn a set of rules described as first order Horn-clauses. During the learning process, new predicates are created as necessary in a hierarchical fashion to enable the system to learn highly complex recursive rules. Our learning system has been implemented in Allegro Common Lisp on a Unix system, and many different learning situations have been tested. The computer program for our learning system is listed in the Appendix.

Our system not only can handle very complex examples which can not be handled by FOIL and FORGE, as given in the previous chapter, but it is also very efficient as compared to FOIL and FORGE. We show the comparison of our learning system with FOIL and FORGE in Table 5.1 for the examples in the last chapter.

Table 5.1. Comparison of Our System with FOIL and FORGE.

	Our Learning System	FOIL [25]	FORGE [6]
Language	Allegro Common Lisp	C	Franz Lisp
System	DECstation 5000	DECstation 3100	Encore Multimax 320
Example1	<1 sec	Can't find the rules	Can't find the rules
Example2	10 secs	Can't find the rules	Can't find the rules
Example3	8 secs	Can't find the rules	Can't find the rules
Example4	6 secs	188 secs	About 18 hours [7]
Example5	<1 sec	0.1 sec	Not Available
Example6	2 sec	Can't find the rules	Can't find the rules

The run time for our learning system is real time instead of CPU time, and the smallest time unit we can measure is one second.

There is a limitation for the new predicate to be created in our learning system, i.e., the coverage of the new predicate must be a subset of the coverage of the target predicate. Therefore, the following rules, which have been discussed in [5], can not be learned by our system.

(5.1.a) target(X, Y) ← base1(X, W), newp(W, V), base1(V, Y).

(5.1.b) newp(X, Y) ← base2(X, Y).

(5.1.c) newp(X, Y) ← base2(X, W), newp(W, Y).

where base1 and base2 are the two base relations, and newp is the new created predicate. One may notice that the coverage of newp is not a subset of the coverage of the target.

Hence, one area to be further studied is how to extend the algorithm to deal with such problems. Also, our learning system can not handle currently uncertain or incremental data. Therefore, how to extend our learning system to deal with those conditions could be the subject of the future research.

BIBLIOGRAPHY

[1] Clocksin, W.F., and Mellish, C.S. (1984). *Programming in Prolog*, Springer-Verlag.

[2] DeJong, G.F. and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning 1*, pp. 145-176.

[3] Ellman, T. (1985). Generalizing logic circuit designs by analyzing proofs of correctness. *Proceedings of IJCAI-85*, pp. 643-646. Los Angeles.

[4] Kundu, S. and Langley, P. (1992). Inductive learning of rules by combining explanation-based learning and rule-coverage. *Proceedings of 5th International symp. on Artificial Intelligence, ISAI-92*, Cancun, Mexico, pp. 311-318.

[5] Kundu, S. (1992). Absorption by decomposition - a more powerful form of absorption. *Tech. Rept.*, Dept. of Computer Science, LSU

[6] Langley, P. (1992). Learning Horn-clauses as classification rules for relations. Ph.D. Dissertation, LSU, Baton Rouge.

[7] Langley, P. Personal communication.

[8] Lapointe, S. and Matwin S. (1992). Sub-unification: A tool for efficient induction of recursive programs. *Machine Learning*, 9, pp. 273-281.

[9] LLoyd, J.W. (1984). *Foundation of Logic Programming*, Springer-Verlag.

[10] Luger, G. and Stubblefield (1989). *Artificial Intelligence and the Design of Expert Systems*, Benjamin/Cummings.

[11] Mahadevan, S. (1985). Verification-based learning: A generalization strategy for inferring problem reduction methods. *Proceedings of IJCAI-85*, pp. 616-623. Los Angeles.

[12] Michalski, R.S., and Negri, P. (1977). An experiment on inductive learning in chess end games. *Machine Representation of Knowledge, Machine Intelligence 8*, pp. 175-192.

[13] Michalski, R. S. (1983). A theory and methodology of inductive learning. *Machine Learning: An Artificial Intelligence Approach*, pp. 83-134.

[14] Minton, S. (1984). Constraint-based generalization. *Proceedings of AAAI-84*, pp. 251-254. Austin.

[15] Minton S., Carbonell J., Knoblock C., Kuokka, D., Etzioni, O., and Gil, Y. (1990). Explanation-based learning: a problem solving perspective. *Machine Learning Paradigms and Methods*, The MIT Press.

[16] Mitchell, T., Keller, R. and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning 1*, pp. 47-80.

[17] Muggleton, S. and Buntine W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339-352. Morgan Kaufmann.

[18] Muggleton, S. (1990). Inductive logic programming. *Proceedings of the First Conference on Algorithmic Learning Theory*, Ohmsha.

[19] Muggleton, S. and Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory*, pp. 1-14. Ohmsha.

[20] O'Rorke, P. (1984). Generalization for explanation-based schema acquisition. *Proceedings of AAAI-84*, pp. 260-263. Austin.

[21] Pazzani, M., Dyer, M. and Flower, M. (1986). The role of prior causal theories in generalization. *Proceedings of AAAI-86*, pp. 545-550. Philadelphia.

[22] Pazzani, M. and Kibler, D., (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9, pp. 57-94.

[23] Plotkin, G.D. (1971). A further note on inductive generalization *Machine Intelligence*, pp. 101-124. Elsevier, Edinburgh.

[24] Quinlan, J.R. (1979). Discovering rules from large collections of examples: a case study. *Expert Systems in the Micro Electronic Age*, Edinburgh University Press.

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

[26] Ricchards, B.L., and Mooney, R.J. (1992). Learning relations by pathfinding. *proceedings of AAAI-92*,

[27] Rouverirol, C. (1990). Saturation: Postponing choices for Inverting Resolution, *Proceedings of the nineth European Conference on AI*, pp. 557-562. Pitman.

[28] Rouveirol C., and Puget, J.F. (1990). Beyond inversion of resolution. *proceedings of the fifth International Conference on Machine Learning*, pp. 122-130.

[29] Rouverirol, C. (1991). ITOU: Induction of first order theories. *Proceedings of the First Inductive Learning Programming Workshop.*

[30] Shapiro, A., and Niblett, T. (1982). Automatic induction of classification rules for a chess endgame. *Advances in Computer Chess, volume 3*, Edinburgh University Press.

[31] Wilensky, R (1986). *Common LISPcraft*, W. W. Norton and Company, New York.

[32] Winston, P.H. (1984). *Artificial Intelligence*, Addison-Wesley, Reading.

[33] Wirth, R. and O'Rorke, P. (1991). Constraints on predicate invention. *ML-91*, pp. 457-461.

APPENDIX
OUR LEARNING PROGRAM

```
;there are two set of catch-throws in this program
;
;1.      "catch 'invalid" and "throw 'invalid""
;       "catch 'invalid" called from valid-rule and valid-rule-set
;       "throw 'invalid" called from rule-coverage
;
;2.      "catch 'valid-found" and "throw 'valid-found"
;       "catch 'valid-found" called from find-rules
;       "throw 'valid-found" called from expand-a-node

;initialize the global variables
;
;all predicates must be distinct and different from object constants which
;may be atoms or lists
;
;*input-file*:          set input file name "test.in"
;                       input file contains *base-tuples* & *target-tuples*
;*base-tuples*:                 ((base-name1 (a b ...) (b c ...) ...)
;                       (base-name2 (c d ...) (e f ...) ...)
;                       ...
;                       (base-namei (t u ...) (v w ...) ...))
;*target-tuples*:       (target-name (a b ...) (c d ...) ...)
;                       contains no constants which are not in base tuples
;*named-base-tuples*:           ((base-name1 a b ...) (base-name1 b c ...) ...
;                       (base-name2 c d ...) (base-name2 e d ...) ...
;                       ...
;                       (base-namei t u ...) (base-namei v w ...) ...)
;*uncovered*:           uncovered target tuples
;*ui*:                  indicates that a variable is uninstantiated
;*variables*:           a set of names which starts with a capital letter
;                       constants must not start with capital letter
;*new-pred*:            a set of new predicate names
;*rule-set*:            a set of rule pointers.  each rule pointer is a property
;                       list and has following attributes:
;                       rules: a set of rules
;                       cover: the coverage of the rules
;                       level:  0 -  a fact
;                               1 -  a non-recursive rule
;                               >1 - a set of recursive rules
;                       sons:   two subsets of the current rules (if level>1)
;                               they form the current rules via mutual recursion
;*coverage*:            the coverage of a rule
;*set-coverage*:        the coverage of a set of rules
;*current-level*:       a list of the tree nodes at the current level;
;                       these have been verified as not giving any valid rule
;*next-level*:          a list of the tree nodes at the next level; these are
;                       yet to be created and tested for a valid rule,
;                       one node at a time, until a valid rule is formed
;
(defun init-data ()
  (setq *input-file* (open "test.in" :direction :input))
  (setq *base-tuples* (read *input-file*))
  (setq *target-tuples* (read *input-file*))
  (close *input-file*)
```

```
  (setq *named-base-tuples* (get-named-tuples *base-tuples*))
  (setq *target-tuples* (order-tuples *target-tuples*))
  (setq *uncovered* (cdr *target-tuples*))  ;uncovered target tuples
  (setq *ui* (gensym))  ;to indicate that a variable is uninstantiated
  (setq *variables* '(X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12))
  (setq *new-preds* '(newp1 newp2 newp3 newp4 newp5 newp6
    newp7 newp8 newp9 newp10 newp11 newp12))
  (setq *rule-set* nil))

;*********************************************************************
;get linear ordering of the target tuples

;collect argument in jth position from each tuple, for j>=1
;input: ((a1 b1 c1 ...) (a2 b2 c2 ...) ...)
;output: ((a1 a2 ...) (b1 b2 ...) (c1 c2 ...) ...)
(defun args-pos-j-all (arg-tuples)  ;arg-tuples non-empty
  (cond ((null (car arg-tuples)) nil)
        (t (cons (mapcar #'car arg-tuples)
                 (args-pos-j-all (mapcar #'cdr arg-tuples))))))

;count the number of occurrences of an item in the list
;input: a, (a a b c a)
;output: 3
(defun count-item (item a-list)
  (cond ((null a-list) 0)
        ((equal item (car a-list)) (1+ (count-item item (cdr a-list))))
        (t (count-item item (cdr a-list)))))

;form a list of pairs (item, item-count) for all distinct items
;input: (a b a c b a)
;output: ((a 3) (b 2) (c 1))
(defun get-item-count-lists (a-list)
  (cond ((null a-list) nil)
        (t (let* ((first (car a-list))
                  (num (count-item first a-list)))
             (cons (list first num)
                   (get-item-count-lists
                     (remove first a-list :test 'equal)))))))

;get the corresponding count numbers of the first input list
;the count number of the constant in the position j of a sublist is getting
;from the jth sublist of the second input list
;input: ((a c) (b d) ...), (((a 3) (c 2) (b 1) ...) ((a 5) (c 4) (d 3) ...))
;output: ((3 4) (1 3) ...)
(defun get-count-lists (lists1 lists2)
  (cond ((null lists1) nil)
        (t (cons (mapcar #'(lambda (x y) (cadr (assoc x y :test 'equal)))
                         (car lists1) lists2)
                 (get-count-lists (cdr lists1) lists2)))))

;sort each numeric sublist in increasing order
;input: ((2 3 1) (3 4 2) ...)
;output: ((1 2 3) (2 3 4) ...)
(defun sort-lists (lists)
```

```
  (cond ((null lists) nil)
        (t (cons (sort (car lists) '<)
                 (sort-lists (cdr lists))))))

;compare two numeric lists which are in increasing order.
;if first argument of list1 is less than list2 then return t
;if first argument of list2 is less than list1 then return nil
;if they have same arguments then compare them by the next arguments
;input: (1 1 4 ...), (1 2 3 ...)
;output: t
(defun list-less (list1 list2)
  (cond ((null list1) nil) ;to keep the original order in sorting
        ((< (car list1) (car list2)) t)
        ((> (car list1) (car list2)) nil)
        (t (list-less (cdr list1) (cdr list2)))))

;compare two lists based on the second arguments which are
;the numeric lists in increasing order
;input: ((a b c ...) (1 2 3 ...)), ((a c d ...) (1 1 4 ...))
;output: nil
(defun arg-2-less (list1 list2)
  (list-less (cadr list1) (cadr list2)))

;get linear ordering of the target tuples based on the count
;numbers of constants of each argument
(defun order-tuples (target-tuples)
  (let* ((tuple-lists (cdr target-tuples))
         (item-count-lists (mapcar #'get-item-count-lists
           (args-pos-j-all tuple-lists)))
         (count-lists (get-count-lists tuple-lists item-count-lists)))
        (cons (car target-tuples)
              (mapcar #'car (stable-sort (mapcar #'list tuple-lists
                        (sort-lists count-lists)) 'arg-2-less)))))

;*****************************************************************
;check if a single rule is valid

;return all variables in the rule
(defun rule-vars (rule)
 (rem-dup (apply 'append (mapcar #'cdr rule))))

;check if the non-recursive rule, body is made of base relations, is valid
;return coverage of the rule if it is valid, otherwise, return nil
;*ui* is used to indicate that a variable is uninstantiated
(defun valid-rule (rule)
  (setq *coverage* nil)
  (let* ((rule-head (car rule))
         (rule-body (cdr rule))
         (rule-vars (rem-dup (apply 'append (mapcar #'cdr rule))))
         (all-var-vals (mapcar #'(lambda (x) (list x *ui*)) rule-vars)))
        (catch 'invalid (rule-coverage rule-head rule-body all-var-vals))))

;get all the tuples in the target tuples which are covered by the rule
(defun rule-coverage (rule-head rule-body all-var-vals &aux tuple p-tuples)
```

```
  (cond ((null rule-body)
         (setq tuple (mapcar #'(lambda (x)
           (cadr (assoc x all-var-vals))) (cdr rule-head)))
         (cond ((member tuple *target-tuples* :test 'equal)
                (setq *coverage* (rem-dup (cons tuple *coverage*))))
               (t (throw 'invalid (setq *coverage* nil)))))
        (t (setq p-tuples (cdr (assoc (caar rule-body) *base-tuples*)))
           (do* ((tuples p-tuples (cdr tuples)))
             ((null tuples) *coverage*)
             (let* ((tuple (car tuples))
                    (values (mapcar #'(lambda (x)
                      (cadr (assoc x all-var-vals))) (cdar rule-body))))
               (cond ((not (match tuple values)) nil)
                     (t (let* ((vars (cdar rule-body))
                               (v-vs (mapcar #'(lambda (x y) (list x y))
                                 vars tuple))
                               (var-vals (new-all-var-vals
                                 all-var-vals v-vs)))
                          (rule-coverage rule-head
                            (cdr rule-body) var-vals)))))))))

;instantiate variables which have not been instantiated
(defun new-all-var-vals (all-var-vals v-vs)
  (cond ((null v-vs) all-var-vals)
        (t (new-all-var-vals (subst (car v-vs) (list (caar v-vs) *ui*)
             all-var-vals :test 'equal) (cdr v-vs)))))

;match the tuple with values which may contain *ui*
;at any position, *ui* in values matches any constants in tuple
(defun match (tuple values)
  (cond ((null tuple) t)
        ((equal *ui* (car values)) (match (cdr tuple) (cdr values)))
        ((equal (car tuple) (car values)) (match (cdr tuple) (cdr values)))
        (t nil)))

;*****************************************************************
;check if a set of rules are valid

;separate rules into two subsets, non-recursive rules and recursive rules
(defun nonrec-rec-rules (rules)
  (cond ((null rules) nil)
        (t ((lambda (lists) (list (apply 'append (mapcar #'car lists))
              (apply 'append (mapcar #'cadr lists))))
            (mapcar #'(lambda (rule)
              (cond ((equal (caar rule) (caadr rule)) (list nil (list rule)))
                    (t (list (list rule) nil)))) rules)))))

;check if a set of the rules are valid
;the rules are separated into two groups
;check the coverage of the nonrecursive group first then recursive group
;return the coverage if the rules are valid, otherwise, return nil
(defun valid-rule-set (rules)
  (let* ((rules (nonrec-rec-rules rules))
         (nonrec-rules (car rules))
```

```
        (rec-rules (cadr rules))
        (cover (apply 'append (mapcar #'valid-rule nonrec-rules))))
    (setq *set-coverage* cover)
    (cond ((null rec-rules) cover)
          (t (let* ((ncov (catch 'invalid (rec-rule-cover rec-rules cover))))
               (cond ((null ncov) (setq *set-coverage* nil))
                     (t (rem-dup (append cover ncov)))))))))

;return coverage of a set of rules for same head predicate, which also appears
;in the head of each body
;  p(x, y) :- p(x, z), ...
;  p(x, y) :- p(x, z), ...
(defun rec-rule-cover (rec-rules cover)
  (cond ((null cover) nil)
        (t (let* ((ncov (rule-set-newcover rec-rules cover)))
             (setq *set-coverage* (append *set-coverage* ncov))
             (append ncov (rec-rule-cover rec-rules ncov))))))

;return new coverage of a set of recursive rules
(defun rule-set-newcover (rec-rules cover)
  (cond ((null rec-rules) nil)
        (t (let* ((ncov (rule-newcover (car rec-rules) cover))
                  (extcov (append cover ncov)))
             (append ncov (rule-set-newcover (cdr rec-rules) extcov))))))

;return new coverage of a recursive rule
(defun rule-newcover (rec-rule cover)
  (let* ((rule-head (car rec-rule))
         (rule-body-1 (cadr rec-rule))
         (rule-body (cddr rec-rule))
         (rule-vars (rem-dup (apply 'append (mapcar #'cdr rec-rule))))
         (all-var-vals (mapcar #'(lambda (x) (list x *ui*)) rule-vars))
         (newcover nil))
    (do* ((tuples cover (cdr tuples)))
         ((null tuples) newcover)
         (setq *coverage* nil)      ;for the init value in rule-coverage
         (let* ((tuple (car tuples))
                (vars (cdr rule-body-1))
                (v-vs (mapcar #'(lambda (x y) (list x y)) vars tuple))
                (var-vals (new-all-var-vals all-var-vals v-vs))
                (nc (rule-coverage rule-head rule-body var-vals))
                (nc (list-minus nc *set-coverage*))   ;avoid endless cycle
                (nc (list-minus nc newcover)))        ;avoid endless cycle
               (setq newcover (append newcover nc))
               (setq tuples (append tuples nc))))))

;*****************************************************************
;create explanation tree and finds all valid rules

;convert *base-tuples* to *named-base-tuples*
;input: ((base1 (a b) (a c) ...) (base2 (b c) (c d) ...) ...)
;output: ((base1 a b) (base1 a c) ... (base2 b c) (base2 c d) ...)
(defun get-named-tuples (lists)
  (apply 'append (mapcar #'(lambda (sublist &aux name tuples)
```

```
                    (setq name (car sublist))
                    (setq tuples (cdr sublist))
                    (mapcar #'(lambda (tuple)
                                (cons name tuple))
                      tuples))
                lists)))

;expand the tree by breadth-first fashion, one level at a time
(defun expand-tree ()
  (mapcar #'rem-node-prop *current-level*)
  (setq *current-level* *next-level*)
  (setq *next-level* nil)
  (cond ((null *current-level*) ;if valid rule can not be found
                        ;then treat it as a fact
       (setq *coverage* (list (car *uncovered*)))
       (list (cons (car *target-tuples*) (car *uncovered*)))) ;return rule
      (t (mapcar #'expand-a-node *current-level*) ;may terminate early
                                    ;via catch-throw
        (expand-tree)))) ;needed if previous mapcar
                  ;did not find a valid rule

;expand a node one level down
;each node is a property list and has following attributes
;constants:     list of all constants in the rule head and current rule body
;               (new constants added to the end)
;unexplained:   list of all constants in the rule head which
;               are not yet explained
;rule-body:     the current rule body
;exclude-list:  a set of base tuples which can not be used for node expansion
;               (new items are added to the end)
;
;whether a rule is subsumed by some other rule will not be checked
(defun expand-a-node (node)
  (let* ((rule nil)
       (consts (get node 'constants))
       (unexpl (get node 'unexplained))
       (rulebody (get node 'rule-body))
       (excludelist (get node 'exclude-list))
       (availlist (avail-tuples consts *named-base-tuples*))
       (availlist (list-minus availlist excludelist)))
    (do* ((al availlist (cdr al))
        (nn (gensym) (gensym)))
       ((null al) nil)
       (setq excludelist (append excludelist (list (car al))))
       (setf (get nn 'constants) (rem-dup (append consts (cdar al)))
           (get nn 'unexplained) (list-minus unexpl (cdar al))
           (get nn 'rule-body) (append rulebody (list (car al)))
           (get nn 'exclude-list) excludelist)
       (cond ((null (get nn 'unexplained))
           (setq rule (cons (cons (car *target-tuples*)
                                (car *uncovered*))
                        (get nn 'rule-body)))
           (setq rule (generalize-ground-rule rule (get nn 'constants)))
           (cond ((null (valid-rule rule)) nil)
```

```
                (t (throw 'valid-found rule))))
            (t nil))
        (setq *next-level* (append *next-level* (list nn))))))

;replace rule constants by distinct variables
(defun generalize-ground-rule (rule rule-constants &aux var const)
  (do* ((consts rule-constants (cdr consts))
        (vars *variables* (cdr vars)))
       ((null consts) rule)
       (setq var (cond ((null vars) (gensym "V"))
                       (t (car vars))))
       (setq const (car consts))
       (setq rule (mapcar #'(lambda (x) (substitute var const x :test 'equal))
                          rule))))

;remove the property list of a tree node
(defun rem-node-prop (node)
  (remprop node 'constants)
  (remprop node 'unexplained)
  (remprop node 'rule-body)
  (remprop node 'exclude-list))

;get a list of the base tuples which contain at least one of the constants
(defun avail-tuples (constants named-base-tuples)
  (cond ((null named-base-tuples) nil)
        (t (cond ((intersection (cdar named-base-tuples)
                                constants :test 'equal)
                  (cons (car named-base-tuples)
                        (avail-tuples constants (cdr named-base-tuples))))
                 (t (avail-tuples constants (cdr named-base-tuples)))))))

;find all valid rules until all target tuples are covered
(defun find-rules (&aux root)
  (cond ((null *uncovered*) nil)
        (t (setq root (gensym 1))
           (setf (get root 'constants) (car *uncovered*)
                 (get root 'unexplained) (car *uncovered*)
                 (get root 'rule-body) nil
                 (get root 'exclude-list) nil)
           (setq *current-level* nil)
           (setq *next-level* (list root))
           (let* ((np (gensym))
                  (rule (catch 'valid-found (expand-tree))))
                 (setf (get np 'rules) (list rule)
                       (get np 'cover) *coverage*
                       (get np 'level) (cond ((null (cdr rule)) 0)
                                             (t 1)))
                 (setq *rule-set* (append *rule-set* (list np)))
                 (mapcar #'rem-node-prop *current-level*)
                 (mapcar #'rem-node-prop *next-level*)
                 (setq *uncovered* (list-minus *uncovered* *coverage*))
                 (cond ((= 0 (get np 'level)) nil)
                       (t (generalize-rules np (length rule)))))
           (find-rules))))
```

```
;*****************************************************************
;perform absorption operation and builds succesive level of the rules

;use rule1 to generalize other rules
;rule1 and rule2 are two non-recursive rules
;use shorter rule to generalize another rule
(defun generalize-rules (rptr1 ln1 &aux rptr2 ln2)
  (do* ((rule-set *rule-set* (cdr rule-set)))
       ((null rule-set) nil)
       (setq rptr2 (car rule-set))
       (cond ((/= 1 (get rptr2 'level)) nil)
             ;for level=1 only
             (t (setq ln2 (length (car (get rptr2 'rules))))
                (cond ((= ln1 ln2) nil)
                      ((< ln1 ln2) (absorption rptr1 rptr2))
                      (t (absorption rptr2 rptr1)))))))

;substitute each variable in rule1 with a unique name (gensym)
;call absorp-generalize to perform absorption operation
;the following two rules
; r1:   p(x, y) :- q(x, y).
; r2:   p(x, y) :- r(x, z), q(z, y).
;produce
; r3:   p(x, y) :- q(x, y).
;       p(x, y) :- p(z, y), r(x, z).
(defun absorption (rptr1 rptr2)
  (let* ((r1 (car (get rptr1 'rules))) ;rule1
         (r2 (car (get rptr2 'rules))) ;rule2
         (vs1 (rem-dup (apply 'append (mapcar #'cdr r1))))) ;vars in rule1
        (do* ((vs vs1 (cdr vs))) ;subst each var in rule1 by (gensym)
             ((null vs) nil)
             (setq r1 (subst (gensym) (car vs) r1)))
        (absorp-generalize rptr1 rptr2 (cdr r1) (cdr r2) r1 nil)))

;perform absorption operation
;if success, build successive level of the rules to enlarge the coverage
(defun absorp-generalize (rptr1 rptr2 rest-b1 rest-b2 s1 s2 &aux r1 r2 cov np)
  (cond ((null rest-b1) ;rest of body of rule1 is empty
         (cond ((not (equal (cdr s1) s2)) nil) ;not matched
               (t (setq r2 (car (get rptr2 'rules)))
                  (setq r2 (cons (car r2) (cons (car s1) rest-b2)))
                  (setq r1 (car (get rptr1 'rules)))
                  (setq cov (valid-rule-set (list r1 r2)))
                  (cond ((null cov) nil)
                        (t (setq np (gensym))
                           (setq *rule-set* (append *rule-set* (list np)))
                           (setf (get np 'rules) (list r1 r2)
                                 (get np 'level) 2
                                 (get np 'cover) cov
                                 (get np 'sons) (list rptr1 rptr2))
                           (setq *uncovered* (list-minus *uncovered* cov))
                           (build-next-level np 2))))))
        (t (do* ((b2 rest-b2 (cdr b2))
                 (sa s1 s1)
```

```
            (sb s2 s2))
           ((null b2) nil)
           (cond ((not (equal (caar b2) (caar rest-b1))) nil)
                 ;literal name must be the same
                 (t (do* ((vs2 (cdar b2) (cdr vs2))
                          (vs1 (cdar rest-b1) (cdr vs1)))
                         ((null vs2)
                          (setq sb (append sb (list (car b2))))
                          (absorp-generalize rptr1 rptr2 (cdr rest-b1)
                            (remove (car b2) rest-b2) sa sb))
                         (setq sa (subst (car vs2) (car vs1) sa)))))))))


;build successive level of the rules to enlarge the coverage
(defun build-next-level (rptr1 level &aux rules cov rptr2 rptr3)
  (do* ((rule-set *rule-set* (cdr rule-set)))
       ((or (null rule-set) (null *uncovered*)) nil)
       (setq rptr2 (car rule-set))
       (cond ((equal rptr1 rptr2) nil)
             ((/= level (get rptr2 'level)) nil)
             (t (setq rules (rem-dup (append (get rptr1 'rules)
                                             (get rptr2 'rules))))
                (setq cov (valid-rule-set rules))
                (cond ((null cov) nil)
                      ((null (list-minus cov (get rptr1 'cover))) nil)
                      ((null (list-minus cov (get rptr2 'cover))) nil)
                      (t (setq rptr3 (gensym))
                         (setf (get rptr3 'rules) rules
                               (get rptr3 'level) (1+ level)
                               (get rptr3 'cover) cov
                               (get rptr3 'sons) (list rptr1 rptr2))
                         (setq *rule-set* (append *rule-set* (list rptr3)))
                         (setq *uncovered* (list-minus *uncovered* cov))
                         (build-next-level rptr3 (1+ level)))))))))

;*****************************************************************
;remove redundant rules

;remove the rules in the children nodes
(defun rem-sons (rule-set)
  (cond ((null rule-set) *rule-set*)
        (t (cond ((>= 1 (get (car rule-set) 'level))
                  (rem-sons (cdr rule-set)))
                 (t (setq *rule-set* (remove
                      (car (get (car rule-set) 'sons)) *rule-set*))
                    (setq *rule-set* (remove
                      (cadr (get (car rule-set) 'sons)) *rule-set*))
                    (rem-sons (cdr rule-set))))))))

;remove the rules which are the subset of the other rules
;based on the coverage measurement
(defun rem-sub-cover (rptr &aux rlist)
  (setq rlist (member rptr *rule-set*))
  (cond ((null rlist) nil)
```

```
      (t (mapcar #'(lambda (x) (rem-rules rptr x)) (cdr rlist)))))

;compare two set of the rules, if one set of the rules is the subset
;of the other set of the rules, them remove it
(defun rem-rules (rptr1 rptr2)
  (cond ((not (member rptr2 *rule-set*)) nil)
        (t (let* ((cov1 (get rptr1 'cover))
                  (ln1 (length cov1))
                  (cov2 (get rptr2 'cover))
                  (ln2 (length cov2))
                  (diff (length (list-minus cov1 cov2))))
             (cond ((= 0 diff)
                    (setq *rule-set* (remove rptr1 *rule-set*)))
                   ((= ln1 (+ ln2 diff))
                    (setq *rule-set* (remove rptr2 *rule-set*)))
                   (t nil))))))

;if the last non-recursive rule is removed from a recursive rule set
;the recursive rules will be changed to non-recursive
(defun de-absorp (rule t-rules &aux vs1)
  (setq vs1 (rem-dup (apply 'append (mapcar #'cdr rule))))
  (do* ((vs vs1 (cdr vs)))
       ((null vs) nil)
       (setq rule (subst (gensym) (car vs) rule)))
  (mapcar #'(lambda (t-rule &aux r-head r-body t-b1)
              (setq r-head (car rule))
              (setq r-body (cdr rule))
              (setq t-b1 (cadr t-rule))
              (do* ((rvs (cdr r-head) (cdr rvs))
                    (cvs (cdr t-b1) (cdr cvs)))
                   ((null rvs)
                    (append (list (car t-rule)) r-body (cddr t-rule)))
                   (setq r-body (subst (car cvs) (car rvs) r-body))))
          t-rules))

;if a rule were removed and the coverage remains the same,
;then it is a redundant rule and remove it
(defun rem-extra-rules (&aux rptr cov minicov rules rule t-rules t-cov)
  (do* ((rule-set *rule-set* (cdr rule-set)))
       ((null rule-set) *rule-set*)
       (setq rptr (car rule-set))
       (cond ((>= 1 (get rptr 'level)) nil)
             (t (setq cov
                      (do* ((rest-rs (remove rptr *rule-set*) (cdr rest-rs))
                            (cov (get (car rest-rs) 'cover)
                                 (append cov (get (car rest-rs) 'cover))))
                           ((null rest-rs) cov)))
                (setq minicov (list-minus (cdr *target-tuples*) cov))
                (setq rules (nonrec-rec-rules (get rptr 'rules)))
                (setq rules (append (cadr rules) (car rules)))
                (setq cov (get rptr 'cover))
                (do* ((rs rules (cdr rs)))
                     ((null rs)
                      (cond ((null rules)
```

```
                    (setq *rule-set* (remove rptr *rule-set*)))
                   (t (setf (get rptr 'rules) rules
                            (get rptr 'cover) cov)
                      (cond ((null (cadr (nonrec-rec-rules rules)))
                             (setf (get rptr 'level) 1))
                            (t nil)))))
              (setq rule (car rs))
              (setq t-rules (remove rule rules :test 'equal))
              (cond ((equal (caar rule) (caadr rule)) nil)
                    ((null t-rules) nil)
                    (t (cond ((null (car (nonrec-rec-rules t-rules)))
                              (setq t-rules (de-absorp rule t-rules)))
                             (t nil))))
              (setq t-cov (valid-rule-set t-rules))
              (cond ((null (list-minus minicov t-cov))
                     (setq rules t-rules)
                     (setq cov t-cov))
                    (t nil)))))))

;remove all redundant rules to form a minimum set of the rules
(defun rem-redundant-rules ()
  (rem-sons *rule-set*)
  (mapcar #'rem-sub-cover *rule-set*)
  (rem-extra-rules))

;*****************************************************************

;remove duplicate elements from a list; keep first occurence
(defun rem-dup (a-list)
  (cond ((null a-list) nil)
        (t (cons (car a-list)
                 (rem-dup (remove (car a-list) (cdr a-list) :test 'equal))))))

;return list of items in list1 which are not in list2
;input: (a b d a c), (b a)
;output: (d c)
(defun list-minus (list1 list2)
  (cond ((null list1) nil)
        ((member (car list1) list2 :test 'equal)
         (list-minus (cdr list1) list2))
        (t (cons (car list1) (list-minus (cdr list1) list2)))))

;add new predicates into the rules
;add new predicates when there are more than one subset of rules
;and the subset of the rules include recursive rule
(defun add-newp (new-preds &aux nonfact-rule-set newp rules)
  (setq nonfact-rule-set (apply 'append
    (mapcar #'(lambda (x) (cond ((= 0 (get x 'level)) nil)
                                (t (list x)))) *rule-set*)))
  (cond ((>= 1 (length nonfact-rule-set)) nil)
        (t (do* ((rule-set nonfact-rule-set (cdr rule-set)))
                ((null rule-set) nil)
                (cond ((>= 1 (get (car rule-set) 'level)) nil)
                      (t (setq newp (cond ((null new-preds)
```

```
                        (gensym "newp"))
                       (t (car new-preds))))
                (setq new-preds (cdr new-preds))
                (setq rules (get (car rule-set) 'rules))
                (setq rules (cons (list (caar rules)
                  (subst newp (caaar rules) (caar rules)))
                  (subst newp (caaar rules) rules)))
                (setf (get (car rule-set) 'rules) rules)))))))

;*****************************************************************************

;print a literal in the form pred(var1, var2, ...)
(defun prt-literal (literal)
  (princ (car literal))
  (princ "(")
  (do* ((vars (cdr literal) (cdr vars)))
      ((null vars) nil)
      (princ (car vars))
      (cond ((null (cdr vars)) (princ ")"))
          (t (princ ", ")))))

;print a prolog rule
(defun prt-rule (rule)
  (prt-literal (car rule))
  (cond ((null (cdr rule)) (princ ".") (terpri))
       (t (princ " :- ")
         (do* ((rule-body (cdr rule) (cdr rule-body)))
             ((null rule-body) nil)
             (prt-literal (car rule-body))
             (cond ((null (cdr rule-body)) (princ ".") (terpri))
                 (t (princ ", ")))))))

;print output data
(defun prt-outdata ()
  (prog ()
    (terpri)
    (princ "base tuples:")
    (terpri)
    (princ *base-tuples*)
    (terpri)
    (terpri)
    (princ "target tuples:")
    (terpri)
    (princ *target-tuples*)
    (terpri)
    (terpri)
    (princ "rules, level, and coverage:")
    (terpri)
    (mapcar #'(lambda (x) (princ "rules ") (terpri)
                    (mapcar #'prt-rule (get x 'rules))
                    (princ "level: ") (princ (get x 'level)) (terpri)
                    (princ "coverage: ") (terpri)
                    (princ (get x 'cover)) (terpri) (terpri))
      *rule-set*)))
```

```
;*****************************************************************
;main function

(defun main ()
  (init-data)
  (find-rules)
;; print out all rules before removing redundant rules
;; (prt-outdata)
  (rem-redundant-rules)
  (add-newp *new-preds*)
  (prt-outdata))
```

VITA

Ching-Liang Tseng was born in Taiwan, Republic of China on June 4, 1955. He received a diploma in Chemical Engineering from Ming-Chi Institute of Technology which he attended from September of 1970 to June of 1975.

After graduating he served as a second-lieutenant in Chinese Air Force for two years. In September of 1977, he returned to Ming-Chi Institute of Technology as a full time teaching assistant. He entered the University of Mississippi in September of 1980 and completed his master program in Chemical Engineering in May of 1982. Form August of 1982 to July of 1984, he taught, as an instructor, at Ming-Chi Institute of Technology. In August of 1984, he joined the Ph. D. program in Chemical Engineering at Louisiana State University. In January of 1989, he changed his major to Computer Science and pursued the Ph. D. degree in the same university.

In May of 1981, he was married to Mei-Hsiang Lai of Taiwan, Republic of China.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Ching-Liang Tseng

Major Field: Computer Science

Title of Dissertation: Learning Complex Recursive Rules

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

June 11, 1993