LEARNING NONRECURSIVE DEFINITIONS OF RELATIONS WITH LINUS

Nada Lavrač, Sašo Džeroski and Marko Grobelnik
Jožef Stefan Institute, Jamova 39
61000 Ljubljana, Yugoslavia
Phone: (+38)(61) 214 399, Fax: (+38)(61) 219 385
E-mail: nada@ijs.ac.mail.yu

Abstract

Many successful inductive learning systems use a propositional attribute-value language to represent both training examples and induced hypotheses. Recent developments are concerned with systems that induce concept descriptions in first-order logic. The deductive hierarchical database (DHDB) formalism is a restricted form of Horn clause logic in which nonrecursive logical definitions of relations can be expressed. Having variables, compound terms and predicates, the DHDB formalism allows for more compact descriptions of concepts than an attribute-value language. Our inductive learning system LINUS uses the DHDB formalism to represent concepts as definitions of relations. The paper gives a description of LINUS and presents the results of its successful application to several inductive learning tasks taken from the machine learning literature. A comparison with the results of other first-order learning systems is given as well.

1 Introduction

The general framework for machine learning can be stated as follows. Given a set of *positive training examples* E_T, a (possibly empty) set of *negative training examples* E_F, and *background knowledge* B, find an *hypothesis* or *concept description* H such that $B, H \vdash E_T$ and $B, H \not\vdash E_F$.

The development of inductive learning systems can focus on different problems, such as restricted representation language, inability to make use of background knowledge, noise in training examples, and bias of vocabulary. Our work deals with the first three problems. Our system LINUS can effectively use background knowledge (B) in inducing hypotheses (H). Hypotheses have the form of DHDB (deductive hierarchical database) clauses, i.e., typed nonrecursive Horn clauses with negation, and represent logical definitions of relations.

Many successful inductive learning systems use a propositional attribute-value language to represent training examples and concept descriptions, for example, the members of the AQ (e.g., Michalski et al. 1986) and TDIDT (top-down induction of decision trees, e.g., Quinlan 1986) families of inductive learning programs. Recent developments are concerned with systems that induce concept descriptions in first-order logic. Very promising approaches are used in CIGOL (Muggleton & Buntine 1988), GOLEM (Muggleton & Feng 1990) and FOIL (Quinlan 1989, 1990), which induce descriptions of complex relations in Horn clause logic.

Our approach can be best compared to FOIL, since both systems are based on ideas that have proved effective in attribute-value learning programs. Each extends these ideas to a more expressive first-order logical formalism in its own way. The idea in LINUS is to incorporate existing attribute-value learning programs into the DHDB environment. LINUS now incorporates ASSISTANT (Cestnik, Kononenko & Bratko 1987), a member of the TDIDT family, and NEWGEM (Mozetič 1985), a member of the AQ family, which are used in learning attribute-value descriptions. The incorporation into the DHDB environment is done by a special interface implemented in Prolog. The DHDB formalism is used to enhance the expressiveness of the propositional attribute-value languages used in ASSISTANT and NEWGEM. In DHDB, a typed language is used; the increase in expressiveness is due to universally quantified variables, compound terms, and utility predicates and functions, which can be used in the induced hypotheses. Despite the fact that recursively defined predicates and infinite terms (i.e., terms which can take a value from an infinite set) are not allowed in concept descriptions, the formalism is appropriate for a large scale of real-life problems.

An initial algorithm for learning in the DHDB formalism is a part of QuMAS (Qualitative Model Acquisition System, Mozetič 1987). Its further development is described in (Mozetič & Lavrač 1988, Lavrač & Mozetič 1988). The algorithm was first used to learn functions of components of a qualitative model of the heart in the KARDIO system (Bratko, Mozetič & Lavrač 1989). A detailed description of its successor LINUS and results of a number of other experiments can be found in (Lavrač 1990).

The aim of this paper is to show how LINUS can be used to learn nonrecursive logical definitions of relations. Section 2 introduces the DHDB formalism, gives an overview of the system, and describes the learning algorithm. Section 3 gives results of the successful application of LINUS to several inductive learning tasks taken from machine learning literature. Our results are compared to the results obtained by FOIL (Quinlan 1989, 1990). In the chess endgame domain, we compare the classification accuracy and efficiency with FOIL, DUCE and CIGOL (Muggleton et al. 1989).

2 Learning in the DHDB formalism

2.1 The DHDB formalism

A *deductive database* is a finite set of typed *Horn clauses with negation* of the form:

$$A \leftarrow L_0, \ldots, L_n$$

where A is an atom (predicate symbol applied to terms) and each L_i is a literal (positive or negative atom).

A deductive database is called *hierarchical* if its predicates can be partitioned into levels so that the bodies of the clauses in the definitions of higher level predicates contain only lower level predicates (Lloyd 1987). Consequently, recursive predicate definitions are not allowed in *deductive hierarchical databases* (DHDB).

In deductive databases a *typed language* is used. Types provide a natural way of specifying the domain of a database. We emphasize that, in contrast to the usual restriction in deductive databases, in our formalism compound terms are allowed to appear as arguments of predicates in a database and in queries. However, only hierarchical (nonrecursive) types are allowed. This restriction bans recursive data types, which means that there are only a finite number of ground (non-variable) terms of each type, and consequently, that each query can only have a finite number of answers (Lloyd 1987). A comparison of DHDB to other logical formalisms is given in (Mozetič & Lavrač 1988).

2.2 Learning in LINUS

In attribute-value learning, given is a set of training instances (given as n-tuples of values of a fixed collection of *attributes*), each belonging to exactly one of the possible *classes*. Classes are values of the *decision* or *dependent variable* and attributes are called *independent variables*. The learning task is to find a rule, called a *hypothesis* or a *concept description*, that can be used to predict the class of an unseen object as a function of its attribute values.

In DHDB, a set of training instances is given as a set of ground facts specifying a *relation* between n domain entities. By analogy with an object's *class*, each ground fact is labeled $\oplus$ or $\ominus$ to indicate whether or not it is in the relation, i.e., whether it is to be treated as a positive or as a negative instance for learning.

In LINUS, hypothesis H (i.e., the description of the *target* relation to be learned), has the form of a predicate definition (a set of typed nonrecursive Horn clauses with negation, having the same predicate in the head); positive examples E_T are given ground facts; negative examples E_F are either given or generated ground facts; and background knowledge B consists of predicate definitions in the form of typed (possibly recursive) Horn clauses with negation.

The main idea in LINUS is to incorporate existing attribute-value learning programs into a more powerful DHDB environment. This is done by a special DHDB interface, consisting of over 2000 lines of Prolog code. The structure of the system is shown in Figure 1. The DHDB interface transforms positive instances (given facts) and negative instances (possibly generated by the DHDB interface) from the DHDB form into attribute-value tuples and vice versa, from induced if-then rules into the DHDB form. The most important feature of this interface is that, by taking into account type theory, utility predicates and functions are considered as possible new attributes for learning by an attribute-value learning program.

Currently LINUS incorporates two attribute-value learning programs, ASSISTANT and NEWGEM. In our environment it is easy to incorporate other learning algorithms, e.g., GINESYS (Gams 1989) and CN2 (Clark & Niblett 1989), which were already used in some of our experiments. Having different systems in the same DHDB environment, the idea (promoted in GINESYS and LOGART (Cestnik & Bratko 1988)) that multiple knowledge can increase system performance can naturally be further exploited.

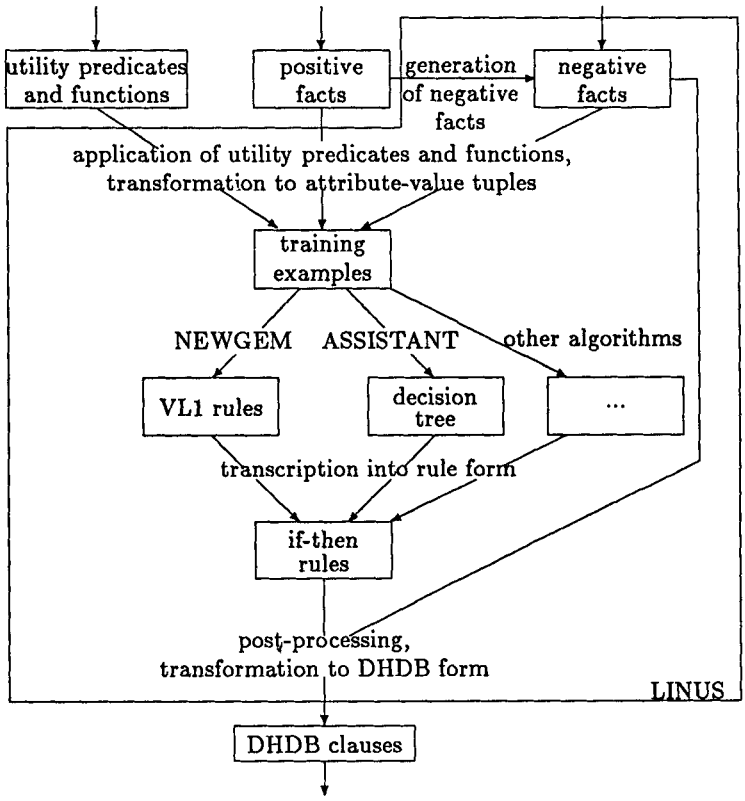

Figure 1: An overview of the LINUS system

2.3 Training examples and domain knowledge

We briefly describe the input to LINUS and illustrate it on a simple example.

Given are training instances in the form of ground facts determining a *relation* between n domain entities. Suppose that relation r(a,b,c) is defined by the following three ground facts labeled $\oplus$:

```
r(a1,b1,b1).
r(a1,b2,b2).
r(a2,b1,b2).
```

Facts labeled $\ominus$, not belonging to the relation, may also be provided. Associated with relation r(a,b,c) is the declaration of the names and the *types* of its arguments, stating that the first argument of relation r named a is of type_a, and the other two arguments named b and c are of type_b. Each type has an associated set of possible values, e.g., {a1,a2} for type_a, and {b1,b2} for type_b. In general, values are constants or compound terms with constant arguments. The use of compound terms allows for the description of compound or structured objects. Values can be nominal, linear, continuous, or structured (hierarchical). The type boolean is defined by default.

Background knowledge consists of predicate definitions specifying relations between domain entities. These predicate definitions can be recursive. We distinguish between what we call *utility predicates* and *utility functions*. Utility functions are predicates with an annotation which determines the *input* and *output* arguments, while utility predicates only have *input* arguments. This is similar to mode declarations in GOLEM (Muggleton & Feng 1990). Utility predicates can be declared *symmetric* in certain pairs of arguments of the same type. For example, a binary predicate $q(X,Y)$ is symmetric in X and Y if they are both of type T and $q(X,Y) = q(Y,X)$ for every value of X and Y. A built-in symmetric utility predicate *equality* (=/2) is defined by default on arguments of the same type.

For each relation, utility predicate and utility function there is a definition of the *types* of arguments. In our example, a binary utility predicate p is defined on arguments of type_a and type_b, respectively.

```
p(a1,b1).
p(a1,b2).
p(a2,b2).
```

The form of examples to be used in learning is determined by the so-called *dependence* relation. We may select any subset of the n arguments of the given relation for learning the target relation. Apart from the k_{Arg} selected arguments, we can also select k_{Util} utility predicates and functions as interesting for the given learning task. In this way we have $k = k_{Arg} + k_{Util}$ selected arguments. In our example, the dependence relation states that all arguments (a, b and c) are selected, and that the possible applications of the utility predicates p and equality (=/2) should be considered as new attributes for learning ($k = 3 + 2 = 5$).

Having selected the k arguments, some of them can be treated as *dependent* variables (analogous to *class* variables) and the others as *independent* variables (analogous to attributes in attribute-value learning). This distinction is important since LINUS can be used both for learning definitions of relations (*relation* learning mode) and for inducing descriptions of particular classes (*class* learning mode). In *class* learning mode, dependent variables are not used for learning, but are only used to determine the different *classes*, where a *class* is a combination of values of the dependent variables. In

relation learning mode, on the other hand, the two classes to be learned are $\oplus$ and $\ominus$, denoting whether the arguments do or do not satisfy the target relation. When generating negative examples (see Section 2.4) we also need to distinguish between dependent and independent variables. In the paper we restrict our scope only to relation learning.

2.4 The learning algorithm

The outermost level of the LINUS learning algorithm consists of the following steps:

- establish the set of positive and negative facts,
- transform facts from the DHDB form into attribute-value tuples,
- induce a concept description by an attribute-value learning program,
- transform the induced if-then rules into the form of DHDB clauses.

Below is a short description of the individual steps illustrated by an example.

In the **first step**, the sets of positive and negative facts are determined. Positive facts are always given explicitly, while negative facts may be either given explicitly, or generated automatically. In *class* learning mode, where examples from different classes are provided, no negative examples are needed. On the other hand, in *relation* learning mode, where there are only two classes ($\oplus$ and $\ominus$), negative examples are necessary. Consequently, if they are not given explicitly, they have to be generated. The generation of negative facts takes into account the type theory. In LINUS, there are several options for generating negative examples.

- When generating negative facts under the *closed-world assumption* (cwa mode), all possible combinations of values of k_{Arg} arguments of the target relation are generated.
- In *partial closed-world assumption* (pcwa) mode, for a given combination of values of the k_{ArgInd} independent variables all combinations of values of the k_{ArgDep} dependent variables are generated.
- In near_misses mode, facts are generated by varying only the value of one of the k_{Arg} variables at a time, where $k_{Arg} = k_{ArgDep} + k_{ArgInd}$.

The generated facts are used as negative instances of the target relation, except for those given as positive. In our example, negative facts are generated under the closed-world assumption. The following facts labeled $\ominus$ are generated:

```
r(a1,b1,b2).
r(a1,b2,b1).
r(a2,b1,b1).
r(a2,b2,b1).
r(a2,b2,b2).
```

In the **second step** of the algorithm, positive and negative facts are transformed into an attribute-value form. The algorithm first checks which are the possible applications of the utility predicates and functions on the arguments of the target relation.

Next, attribute-value tuples are generated by assigning values to the enlarged set of attributes. Values true or false are assigned to each application of a utility predicate on the argument values of the target relation. Similar computation is performed for utility functions, except that values of all output arguments of a function are computed (instead of only assigning values true or false).

In our simple example, the possible applications of predicate p are p(a,b) and p(a,c); for the equality predicate the only possible application is b=c on arguments of the same type type_b. Positive tuples of the form ⟨ a, b, c, (b=c), p(a,b), p(a,c) ⟩ are then generated.

⟨ a1, b1, b1, true, true, true ⟩
⟨ a1, b2, b2, true, true, true ⟩
⟨ a2, b1, b2, false, false, true ⟩

A similar computation is performed for the negative facts.

To show the complexity of the so obtained learning task, let us consider the number of attributes to be used for learning. The total number of attributes k_{Attr} equals:

$$k_{Attr} = k_{Arg} + \sum_{s=1}^{k_{Util}} k_{New,p_s}$$

where k_{Arg} is the number of arguments of the target relation, and k_{New,p_s} is the number of new attributes, resulting from the possible (with regard to given types) applications of one of the k_{Util} utility predicates/functions p_s on the arguments of the target relation.

Suppose that the arguments of a n-ary utility predicate p_s are of types T_i, $i = 1,\ldots,u$. k_{New,p_s} is then equal to the product of the numbers of variations (with repetition) of the numbers of arguments of the same type:

$$k_{New,p_s} = \prod_{i=1}^{u} (k_{ArgT_i})^{n_i}$$

where k_{ArgT_i} is the number of arguments of type T_i in the target relation, n_i is the number of arguments of type T_i in the utility predicate, and u is the number of different types of arguments in the utility predicate ($n = \sum_{i=1}^{u} n_i$). Since a relation where two arguments are identical can be represented by a relation of a smaller arity, we could restrict the arguments of a utility predicate to be different. In this case the following formula would hold:

$$k_{New,p_s} = \prod_{i=1}^{u} \binom{k_{ArgT_i}}{n_i} \cdot n_i! \qquad (1)$$

Utility predicates can be declared as *symmetric* in certain pairs of arguments. In case that p_s is symmetric in all pairs of arguments, the number of variations (without repetition) in (1) is replaced by the number of combinations:

$$k_{New,p_s} = \prod_{i=1}^{u} \binom{k_{ArgT_i}}{n_i}$$

For example, the number of possible applications of the built-in symmetric utility predicate *equality* (=/2) equals to:

$$k_{New,=} = \prod_{i=1}^{r} \binom{k_{ArgT_i}}{2} = \prod_{i=1}^{r} \frac{k_{ArgT_i} \cdot (k_{ArgT_i} - 1)}{2}$$

where k_{ArgT_i} denotes the number of arguments of the same type T_i, and r is the number of different types in the target relation. Similar formulas hold for utility functions.

The **third step** of the algorithm is the induction of the concept description which depends on the choice of the learning algorithm. Training examples in the form of tuples are transformed into the appropriate input form for learning by ASSISTANT or NEWGEM, learning by the chosen attribute-value algorithm is invoked and the obtained concept description (in the form of a decision tree or VL1 rules) is transcribed into the form of if-then rules.

In the **fourth step**, the induced if-then rules are transformed into DHDB form. Before this transformation is performed, a special post-processor checks whether rules can be made more compact by eliminating irrelevant literals and by discarding redundant clauses. A literal in a clause is *irrelevant* if, after it has been eliminated, the clause does not cover any new negative examples. A clause is *redundant* if it is covered by some more general clause. Post-processing is especially effective when transforming decision trees into rules (Quinlan 1987).

In our example, the if-then rules induced by NEWGEM are the following:

class = ⊕ if a=a1 ∧ (b=c) = true.
class = ⊕ if c=b2 ∧ p(a,b) = false.

The DHDB interface transforms these rules into the DHDB clauses below.

r(A,B,C) ← A=a1, B=C.
r(A,B,C) ← C=b2, not p(A,B).

For comparison, the description induced by LINUS using NEWGEM without background relations *p* and *equality* (which is equivalent to NEWGEM itself) was exactly the same as the set of ground facts given as positive examples.

3 Experimental results and comparison with other approaches

This section discusses the performance of LINUS on three learning tasks taken from the machine learning literature. The descriptions of the domains are taken from Quinlan (1989,1990) and our results are compared to the ones obtained by his system FOIL.

LINUS was used in *relation* learning mode to learn definitions of relations from examples of the relation and background relations given as utility predicates. No predicates of the form *Attribute* = *Value* for binding a variable to a constant were allowed

in concept descriptions; this was achieved by selecting only applications of background relations as independent variables for learning ($k_{Attr} = \sum_{s=1}^{k_{Util}} k_{New,p_s}$, $k_{Arg} = 0$).

The arguments of the background relations used in FOIL are not typed and the same relations are sometimes used for different types of arguments which can be confusing. In LINUS, each such relation was replaced by several relations, one for each combination of types, e.g., the precedes relation in the Eleusis example was replaced by the two relations precedes_rank and precedes_suit.

Using two algorithms ASSISTANT and NEWGEM, we typically got slightly different results on the same domain; they were all comparable to the results obtained by FOIL. In the chess endgame domain we were able to compare the classification accuracy and efficiency. Our results are also compared to the ones obtained by DUCE and CIGOL (Muggleton et al. 1989).

3.1 Learning the concept of an arch

In this example, taken from Winston (1975) and described by Quinlan (1990), four objects are given. Two of them are arches and two are not, as shown in Figure 2.

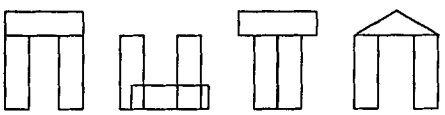

Figure 2: Arches and near misses, from Winston (1975) and Quinlan (1990)

For the target relation arch(A,B,C), stating that A, B and C form an arch with columns B and C and lintel A, the following background relations were used: supports(X,Y), left_of(X,Y), touches(X,Y), brick(X), wedge(X) and parallelepiped(X).

LINUS was first run with explicitly given (two) negative examples, and then with negative examples generated in the *closed-world assumption* mode. Results of LINUS and FOIL are listed below.

```
% LINUS
% negative examples: explicitly
arch(A,B,C) ←
  supports(B,A),
  not touches(B,C).
% negative examples: cwa
arch(A,B,C) ←
  left_of(B,C),
  supports(B,A),
  not touches(B,C).
```

```
% FOIL
% negative examples: explicitly
% no clauses generated because
% of encoding length restriction

% negative examples: cwa
arch(A,B,C) ←
  left_of(B,C),
  supports(B,A),
  not touches(B,C).
```

The definition induced with explicitly given negative examples can be paraphrased as follows: "A, B and C form an arch if B supports A and B does not touch C". Note that this definition is more general than the one obtained in cwa mode; the latter has an additional condition stating that B must be left of C.

With explicitly given negative instances FOIL was unable to learn the concept description because of the encoding length restriction. Namely, a heuristic used in FOIL restricts the total length of the concept description to the number of bits needed to enumerate the training instances explicitly. Our result in cwa mode is the same as the one by FOIL.

On an enlarged set of training instances containing two more negative examples (Lavrač, Džeroski & Grobelnik 1990) the result using both ASSISTANT and NEWGEM was essentially the same as reported by Winston (1975):

arch(A,B,C) ← supports(B,A), supports(C,A), not touches(B,C).

3.2 Eleusis - Learning rules that govern card sequences

The Eleusis learning problem was originaly attacked by the SPARC/E system (Dietterich & Michalski 1986). The description here is taken from Quinlan (1990). In the Eleusis card game, the dealer invents a secret rule specifying a condition under which a card can be added to a sequence of cards. The players attempt to add a card to the current sequence. If a card is a legal successor it is placed to the right of the last card, otherwise it is placed under the last card. The horizontal *main line* represents the sequence as developed so far, while the vertical *side lines* show incorrect plays. Three layouts, reproduced from Quinlan (1990), are given in Figure 3.

Each card other than the first in the sequence provides an example for learning the target relation can_follow. The example is labeled $\oplus$ if the card appears in the main line, and $\ominus$ if it is in a side line. The target relation can_follow(A,B,C,D,E,F) states that a card of rank A and suit B can follow a sequence ending with: a card of rank C and suit D; E consecutive cards of suit D; and F consecutive cards of the same color. Background relations that can be used in induced rules are the following: precedes_rank(X,Y), precedes_suit(X,Y), lower_rank(X,Y), face(X), same_color(X,Y), odd_rank(X), and odd_num(X).

In the first layout, the intended dealer's rule was: "Completed color sequences must be odd and a male card may not appear next to a female card". Neither FOIL nor LINUS could discover the intended rule, because no information on the sex of cards was encoded in the background relations. LINUS using ASSISTANT with post-processing induced the clauses given below.

can_follow(A,B,C,D,E,F) ← same_color(B,D).
can_follow(A,B,C,D,E,F) ← odd_num(F), not precedes_suit(D,B).
can_follow(A,B,C,D,E,F) ← odd_num(F), not precedes_rank(C,A).

While the clauses induced by LINUS using both ASSISTANT and NEWGEM cover all positive examples, the ones induced by FOIL are not complete: they do not cover

main line	A♡	7♣	6♣	9♠	10♡	7♡	10♢	J♣	A♢	4♡	8♢	7♣	9♠
side lines			K♢			5♠		Q♢		3♠			9♡
			J♡										6♡
main (ctd)	10♣	K♠	2♣	10♠	J♠								
side (ctd)		Q♡											
		A♢											
main line	J♣	4♣	Q♡	3♠	Q♢	9♡	Q♣	7♡	Q♢	9♢	Q♣	3♡	K♡
side lines	K♣	5♠				4♠		10♢					
		7♠											
main (ctd)	4♣	K♢	6♣	J♢	8♢	J♡	7♣	J♢	7♡	J♡	6♡	K♢	
mainline	4♡	5♢	8♣	J♠	2♣	5♠	A♣	5♠	10♡				
side line	7♣	6♠	K♣	A♡		6♣		A♠					
	J♡	7♡	3♡	K♢									
	4♣	2♣		Q♠									
	10♠	7♠											
	8♡	6♢											
	A♢	6♡											
	2♢	4♣											

Figure 3: Three Eleusis layouts, from Dietterich and Michalski (1986) and Quinlan (1990)

the positive example can_follow(10,heart,9,spade,1,3). This is due to the encoding restriction, which prevents further search for clauses. Although this might be useful when dealing with noisy data, it is unsuitable for exact domains. The application of the encoding restriction could be controlled by a parameter: it should be applied if the domain is noisy or inexact, but should not be applied to exact domains with no noise, such as the domains presented in this paper. Here are the clauses induced by LINUS using NEWGEM, and by FOIL:

```
% layout 1
% negative examples: explicitly
% LINUS using NEWGEM
can_follow(A,B,C,D,E,F) ←
  same_color(B,D).
can_follow(A,B,C,D,E,F) ←
  odd_num(F),
  odd_rank(A).
can_follow(A,B,C,D,E,F) ←
  not face(A),
  lower_rank(C,A).
```

```
% negative examples: explicitly
% FOIL
can_follow(A,B,C,D,E,F) ←
  same_color(B,D).
can_follow(A,B,C,D,E,F) ←
  odd_num(F),
  odd_rank(A).
```

In layout 2 both LINUS using NEWGEM and FOIL correctly induced the intended rule: "Play alternate face and non-face cards". ASSISTANT's rule set contains a superfluous clause which was not removed in post-processing.

```
% layout 2
% LINUS using NEWGEM
can_follow(A,B,C,D,E,F) ←
  face(A),
  not face(C).
can_follow(A,B,C,D,E,F) ←
  face(C),
  not face(A).
```

```
% FOIL
can_follow(A,B,C,D,E,F) ←
  face(A),
  not face(C).
can_follow(A,B,C,D,E,F) ←
  face(C),
  not face(A).
```

```
% LINUS using ASSISTANT (after post-processing)
can_follow(A,B,C,D,E,F) ← face(C), not face(A).
can_follow(A,B,C,D,E,F) ← face(A), not face(C).
can_follow(A,B,C,D,E,F) ← not odd_num(E).
```

In layout 3 the intended rule was: "Play a higher card in the suit preceding that of the last card; or, play a lower card in the suit following that of the last card". FOIL discovered only one clause, approximately describing the first part of the rule. LINUS using ASSISTANT discovered an approximation of the whole rule: "Play a higher or equal card in the suit preceding that of the last card; or, play a lower card in the suit following that of the last card". The DHDB clauses are given below.

```
can_follow(A,B,C,D,E,F) ← lower_rank(A,C), precedes_suit(D,B).
can_follow(A,B,C,D,E,F) ← precedes_suit(B,D), not lower_rank(A,C).
```

Again, the descriptions induced by LINUS are complete, while FOIL's is not. Using NEWGEM, LINUS generated exactly the intended dealer's rule.

```
% layout 3
% LINUS using NEWGEM
can_follow(A,B,C,D,E,F) ←
  lower_rank(A,C),
  precedes_suit(D,B).
can_follow(A,B,C,D,E,F) ←
  lower_rank(C,A),
  precedes_suit(B,D).
```

```
% FOIL
can_follow(A,B,C,D,E,F) ←
  precedes_suit(B,D),
  not lower_rank(A,C).
```

3.3 Learning illegal positions in a chess endgame

The domain of this learning task, described in Muggleton et al. (1989) and Quinlan (1990), is the chess endgame White King and Rook versus Black King. The target

relation illegal(A,B,C,D,E,F) states whether the position in which the White King is at (A,B), the White Rook at (C,D) and the Black King at (E,F) is not a legal White-to-move position. In FOIL, the domain knowledge is represented by the two relations adjacent(X,Y) and less_than(X,Y) indicating that rank/file X is adjacent to rank/file Y and rank/file X is less than rank/file Y, respectively.

LINUS uses the following utility predicates: adjacent_rank(X,Y), adjacent_file(X,Y), less_rank(X,Y), less_file(X,Y) and equality X=Y. Their arguments are of type rank (with values 1 to 8) and file (with values a to h), respectively.

The training and testing sets used in our experiments were the ones used by Muggleton et al. (1989). There are altogether ten sets of positions (examples), five of 100 examples each and five of 1000 examples each. Each of the sets was used as a training set for the three systems FOIL, LINUS using ASSISTANT, and LINUS using NEWGEM. The sets of clauses were then tested as described in Muggleton et al. (1989). The clauses obtained from a small set were tested on the 5000 examples from the large sets and the clauses obtained from each large set were tested on the remaining 4500 examples.

System	100 training instances		1000 training instances	
	Accuracy	Time	Accuracy	Time
CIGOL	77.2%	21.5 hr	N/A	N/A
DUCE	33.7%	2 hr	37.7%	10 hr
FOIL on different sets	92.5% sd 3.6%	1.5 sec	99.4% sd 0.1%	20.8 sec
FOIL	90.8% sd 1.7%	31.6 sec	99.7% sd 0.1%	4.0 min
LINUS using ASSISTANT	98.1% sd 1.1%	55.0 sec	99.7% sd 0.1%	9.6 min
LINUS using NEWGEM	88.4% sd 4.0%	30.0 sec	99.7% sd 0.1%	4.3 min

Table 1: Results on the chess endgame tasks

Table 1 gives the results in the chess endgame task. The classification accuracy is given by the percentage of correctly classified testing instances and by the standard deviation (sd), averaged over 5 experiments. The first two rows are taken from Muggleton et al. (1989), the third is from Quinlan (1990) and the last three rows present the results of our experiments. Note that the results reported by Quinlan (1990) were not obtained from the same training and testing sets. The times in the first two rows are for a Sun 3/60, in the third for a DECStation 3100, in the fourth for a Sun 3/50 and in the last two rows CPU times are given for a VAX-8650 mainframe. The times given for LINUS include transformation to attribute-value form, learning and transformation into DHDB clauses.

In brief, on the small training sets LINUS using ASSISTANT with post-processing outperformed FOIL. According to the T-test for dependent samples, this result is significant at the 0.5% level. Although LINUS using NEWGEM was sligtly worse than FOIL, this result is not significant (even at the 20% level). The clauses obtained with LINUS

are as short and understandable (transparent) as FOIL's. On the large training sets both systems performed equally well. Although LINUS is slower than FOIL, it is much faster than DUCE and CIGOL. LINUS is slowed down mainly by the parts implemented in Prolog, that is the DHDB interface and especially the post-processor. More efficient implementations would significantly improve LINUS' speed. For illustration, for the small training sets, the average time spent on transforming to attribute-value form, learning and transforming to DHDB form was 16, 6 and 36 seconds for ASSISTANT, and 16, 11 and 3 seconds for NEWGEM, respectively.

Our latest measurements on noisy data indicate that for large and noisy training sets FOIL is much slower than LINUS. Namely, on the training set consisting of the 5000 examples with artificially added noise (30%), it took LINUS less than 20 minutes of VAX 8650 CPU time to generate the hypothesis while FOIL on Sun 3/50 did not complete the induction in 24 hours.

As an example, the clauses induced by LINUS using ASSISTANT (with post-processing) from one of the sets of 100 examples are:

```
illegal(A,B,C,D,E,F) ← C=E.
illegal(A,B,C,D,E,F) ← D=F.

illegal(A,B,C,D,E,F) ← adjacent_file(A,E), B=F.
illegal(A,B,C,D,E,F) ← adjacent_file(A,E), adjacent_rank(B,F).
illegal(A,B,C,D,E,F) ← A=E, adjacent_rank(B,F).

illegal(A,B,C,D,E,F) ← A=E, B=F.
```

These may be paraphrased as: a position is illegal if the Black King is on the same rank or file as (i.e., is attacked by) the Rook, or the White King and the Black King are next to each other, or the White King and the Black King are on the same square. Although these clauses are neither consistent nor complete, they correctly classify 98.5% of the 5000 unseen cases.

4 Summary and discussion

Compared to attribute-value learning, our approach has a number of advantages. It allows for relational descriptions; use of compound terms; compact description of concepts; use of utility predicate definitions and utility functions (background knowledge) in concept descriptions; and inclusion of existing successful attribute-value learning programs into the logic programming environment. LINUS can be used both for learning definitions of relations and for inducing descriptions of individual classes (possibly considering more than one decision variable). In LINUS, we use attribute-value learning programs that embody years of research work, that are known to perform well and that were tested and evaluated on a number of real-life domains. We add to their advantageous features (e.g., mechanisms for handling noisy data in ASSISTANT) the ability of learning logical definitions of relations in a more expressive first-order representational formalism.

For the sake of efficiency, all systems that learn in first-order logic restrict the hypotheses language. For example, FOIL uses function-free and CIGOL negation-free Horn clause logic. LINUS uses an even more restricted language, i.e., the deductive hierarchical database formalism. All variables in the body of a DHDB clause must appear in its head. This is the main reason why LINUS can not learn recursive definitions of relations, while FOIL and CIGOL can. On the other hand, the efficiency of attribute-value learning algorithms is preserved which is extremely important in large real-life domains.

In this paper we have shown that LINUS can induce concept descriptions similar to the ones obtained with other systems that learn definitions of relations in first-order logic. The results of the experiments in learning nonrecursive definitions are equal to or better than the ones obtained with FOIL. The descriptions generated by LINUS are more general and more accurately represent the intended relations. In the 'arches' domain less training examples were needed and the generation of negative examples in the closed-world assumption mode was unnecessary. In the task of learning rules that govern card sequences, in layout 3 LINUS, unlike FOIL, induced the intended (correct) definition, and in the first layout it induced a more general definition. In the chess endgame, LINUS using ASSISTANT achieved better classification accuracy than FOIL. LINUS using NEWGEM achieved comparable results. LINUS and FOIL were both much better than CIGOL.

To summarize, in our experiments, LINUS performed slightly better than FOIL. Having parts implemented in Prolog, LINUS was slightly slower than FOIL, implemented in C. However, both LINUS and FOIL were substantially more efficient than DUCE and CIGOL. Our latest measurements on noisy data indicate that for large and noisy training sets FOIL is much slower than LINUS.

Acknowledgements

This research was supported by the Slovene Research Council and is part of the ESPRIT II Basic Research Action No. 3059, Project ECOLES. We are grateful to Ross Quinlan for making available his system FOIL and the data used in the experiments, and Michael Bain for the chess endgame data. We wish to thank Igor Mozetič for his contribution in the development of LINUS, Dunja Mladenič for the implementation of ASSISTANT in the VAX/VMS environment, and Ivan Bratko and Igor Kononenko for their comments on an earlier draft of the paper.

References

Bratko, I., Mozetič, I. & Lavrač, N. (1989) KARDIO: A study in deep and qualitative knowledge for expert systems. Boston, MA: The MIT Press.

Cestnik, B. & Bratko, I. (1988) Learning redundant rules in noisy domains. Proc. European Conference on Artificial Intelligence, ECAI-88, Muenchen, Germany.

Cestnik, B., Kononenko, I. & Bratko, I. (1987) ASSISTANT 86: A knowledge-elicitation tool for sophisticated users. In: Bratko, I. & Lavrač, N. (eds.) Progress in machine learning. Wilmslow: Sigma Press.

Clark, P. & Niblett, T. (1989) The CN2 induction algorithm. Machine Learning 1 (3), 261-284. Kluwer Academic Publishers.

Dietterich, T.G. & Michalski, R.S. (1986) Learning to predict sequences. In: Michalski, R.S., Carbonell, J.G. & Mitchell, T.M. (eds.) Machine learning: An artificial intelligence approach (Volume 2). Los Altos: Morgan Kaufmann.

Gams, M. (1989) New measurements highlight the importance of redundant knowledge. Proc. European Working Session on Learning, EWSL 89. Montpellier, France: Pitman.

Lavrač, N. (1990) Principles of knowledge acquisition in expert systems. PhD Thesis, Maribor University, Yugoslavia.

Lavrač, N., Džeroski, S. & Grobelnik, M. (1990) Experiments in learning nonrecursive definitions of relations with LINUS. Report IJS-DP-5863, Jožef Stefan Institute, Ljubljana, Yugoslavia.

Lavrač, N. & Mozetič, I. (1988) Experiments with inductive learning programs NEWGEM and ASSISTANT in the DHDB environment. Report IJS-DP-5029, Jožef Stefan Institute, Ljubljana, Yugoslavia.

Lloyd, J.W. (1987) Foundations of logic programming (Second edition). Springer-Verlag.

Michalski, R.S., Mozetič, I., Hong, J. & Lavrač, N. (1986) The multi-purpose incremental learning system AQ15 and its testing application on three medical domains. Proc. National Conference on Artificial Intelligence, AAAI-86. Philadelphia, PA: Morgan Kaufmann.

Mozetič, I. (1985) NEWGEM: Program for learning from examples - Technical documentation and user's guide. Report, University of Illinois at Urbana-Champaign, Department of Computer Science. Also: Report IJS-DP-4390, Jožef Stefan Institute, Ljubljana, Yugoslavia.

Mozetič, I. (1987) Learning of qualitative models. In: Bratko, I. & Lavrač, N. (eds.) Progress in machine learning. Wilmslow: Sigma Press.

Mozetič, I. & Lavrač, N. (1988) Incremental learning from examples in a logic-based formalism. Proc. Int. Workshop on Machine Learning, Meta-reasoning and Logic, Sesimbra, Portugal.

Muggleton, S.H. & Buntine, W. (1988) Machine invention of first-order predicates by inverting resolution. Proc. Machine Learning Conference, Ann Arbor, Michigan.

Muggleton, S.H., Bain, M., Hayes-Michie, J. & Michie, D. (1989) An experimental comparison of human and machine learning formalisms. Proc. Sixth International Workshop on Machine Learning. Ithaca, NY: Morgan Kaufmann.

Muggleton, S.H. & Feng, C. (1990) Efficient induction of logic programs. Proc. First Conference on Algorithmic Learning Theory. Tokyo: Ohmsha.

Quinlan, J.R. (1986) Induction of decision trees. Machine Learning 1 (1), 81-106. Kluwer Academic Publishers.

Quinlan, J.R. (1987) Generating production rules from decision trees. Proc. Int. Joint Conference on Artificial Intelligence, IJCAI 87. Milano, Italy: Morgan Kaufman.

Quinlan, J.R. (1989) Learning relations: Comparison of a symbolic and a connectionist approach. Technical Report 346, Basser Dept. Comp. Sc., University of Sydney, Sydney, Australia.

Quinlan, J.R. (1990) Learning logical definitions from relations. To appear in Machine Learning. Kluwer Academic Publishers.

Winston, P.H. (1975) Learning structural descriptions from examples. In: Winston, P.H. (ed.) The psychology of computer vision. New York: McGraw-Hill.