UNIVERSIDADE DE SÃO PAULO

Limiting the background knowledge in inductive logic programming

MARIA DO CARMO NICOLETTI
MARIA CAROLINA MONARD

Nº 5

N O T A S

Instituto de Ciências Matemáticas de São Carlos

Instituto de Ciências Matemáticas de São Carlos

ISSN - 0103-2577

Limiting the background knowledge in inductive logic programming

MARIA DO CARMO NICOLETTI

MARIA CAROLINA MONARD

Nº 5

NOTAS DO ICMSC

Série Computação

São Carlos

jan./ 1994

Limiting the Background Knowledge in Inductive Logic Programming

Maria do Carmo Nicoletti[1]
Universidade Federal de São Carlos/ILTC/IFQSC-USP
Departamento de Computação
Via Washington Luiz, km 235
Caixa Postal 676, 13565-905 - São Carlos, SP
e-mail:carmo@icmsc.usp.br

Maria Carolina Monard[2]
Universidade de São Paulo/ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP
e-mail: mcmonard@icmsc.usp.br

Abstract

Inductive Logic Programming — ILP — is a new trend in the field of Machine Learning, which tries to integrate the techniques already available and established for logic programming in a framework of learning, aiming to induce first-order logic programs from examples, using background knowledge. ILP attempts to overcome some of the limitations of earlier machine learning methods by

- allowing the use of background knowledge (domain theory) and
- providing a more powerful concept description language, namely, the language of logic programs.

The adoption of the language of logic programs as instances, background and hypotheses description languages gives rise to many difficulties that need to be overcome — learning logical definitions requires the exploration of very large space of hypotheses descriptions and, consequentely, restrictions should be imposed on the hypotheses space so to make learning a feasible task.
This work focuses on ways of confining the hypotheses space by restricting the background knowledge description language. The notion of generative clauses, a restricted form of Horn clause program which allows the construction of a finite model associated with a program is discussed in details. Two algorithms are presented: the first verifies if a given background knowledge consists only of generative clauses and the second constructs the finite model associated with it.

[1] Work partially supported by State Research Council FAPESP Proc. Nº 92/2151-8.
[2] Work partially supported by National and State Research Council — CNPq and FAPESP Proc. Nº 92/2151-8.

Contents

1 Introduction **1**

2 The Learning Process in ILP **2**

2.1 Basic Concepts 2

2.2 Learning in ILP 4

2.3 Searching the Hypotheses Space 5

3 Using the Background Knowledge **5**

3.1 Basic Concepts 6

3.2 Restricting the Background Knowledge 6

3.2.1 Background Knowledge Given as a Set of Ground Facts 6

3.2.2 The H-easy Model 7

3.2.3 Generative Clauses 11

4 Transforming Intentional Background Knowledge into Extensional Background Knowledge **14**

4.1 Algorithm for Verifying if a Logical Program is Generative 15

4.2 Algorithm for Generating the Atomic Closure of a Generative Logical Program 15

5 Conclusions **16**

1 Introduction

One of the most widely adopted and studied paradigm for symbolic learning is known as *inductive learning from examples.* In this paradigm the learning task consists in building a general concept description — *hypothesis* — from a given set of examples — *positive examples* — and counterexamples — *negative examples* — of the concept.

Generally machine learning systems employ formal languages for describing instances and concepts referred to as *instance description language* and *concept description language* respectively.

In order to represent instances — training examples — many of the existing inductive learning algorithms use an *attribute-based* language. The concept description language used for expressing the induced hypotheses, which are typically production rules or decision trees, can be treated as variants of attribute-based languages. For the expression of instances and concepts, an attribute-based language can be considered as representative as the language of propositional calculus.

Among the many systems which employ attribute-based languages, two families of systems, namely TDIDT (Top Down Induction of Decision Trees) and AQ, based on ID3 [Quinlan 79] and AQ [Michalski 83] algorithms respectively, have been particularly sucessful.

Despite their success, attribute-based learning methods are constrainned by the language used to describe examples and concepts, as well as by the very limited and inexpressive role that background knowledge plays in the learning process. Only concepts expressible in propositional logic are candidates to be learned by a system which uses an attribute-based language. This strong restriction prevents representing structured objects as well as relations among objects or among its components. Thus relevant aspects of the training examples, which somehow could characterize the concept being learned, cannot be represented.

In order to overcome the representational limitations imposed by an attribute-based language, learning in representations which are more powerful, such as some variants of first-order logic has received more attention recently. It can be noticed an increasing amount of research on

- extending the representational power of machine learning systems by using restricted first-order logics as concept/instance description language and
- effectively incorporating background knowledge to induce more compact hypotheses.

It is well known that logic programming is a traditional and sound area of research which is based on the use of first-order logic to define computer programs. Logic programming can be defined broadly as the use of symbolic logic for the explicit representation of problems and their associated knowledge bases, together with the use of controlled logical inference for the effective solution of those problems. At present, logic programming

is generally understood in more specific terms: the problem-representation language is a particular subset (Horn-clause form) of classical first-order predicate logic, and the problem-solving mechanism is a particular form (resolution) of classical first-order inference [Kowalski 87]. Logic programming in its narrow sense is based on Horn clause and it is often identified with Prolog.

Inductive Logic Programming — ILP — is an attempt to integrate the techniques already available and established for logic programming in a framework of learning, aiming to induce first-order logic programs from examples, using background knowledge. In ILP the system's knowledge consists of examples and background knowledge expressed as a logic program — the expressiveness of logic programs and the use of background knowledge have promoted ILP as a powerful inductive learning paradigm.

The adoption of a more powerful language description gives rise to many difficulties that need to be overcome. Since learning logical definitions requires the exploration of very large space of hypotheses, restrictions should be imposed on the hypotheses space in order to make learning a feasible task. Ways of confining the hypotheses space can be implemented via restricting the hypotheses and background knowledge description languages. By reducing the representative power of the languages employed, the search carried out by a learning system can be both: better controlled and limited.

This work focuses on ways of restricting the use of background knowledge and is organized as follows: in Section 2 some basic concepts used in logic programming are introduced and the task of empirical, single predicate learning process using Inductive Logic Programming is formally presented. Section 3 highlights and discusses the main concepts and ideas used for limiting the background knowledge. The notion of generative clauses, a restricted form of Horn clause program which allows the construction of a finite model associated with a program is discussed in details. An algorithm to verify if the given background knowledge consists only of generative clauses as well as an algorithm for constructing the finite model associated with it are shown in Section 4. Section 5 presents our conclusions and guidelines for future work.

2 The Learning Process in ILP

Next are presented the notation and definitions that formalize the basic notions used in this work as well as a definition for learning in ILP. The learning process treated as a search process is also discussed. The concepts which follow are adapted from [Lloyd 84] and [Lavrač 94].

2.1 Basic Concepts

Definition 2.1 *A definite program clause or a definite clause is a clause which contains exactly one positive literal. It has the form*

$$A \leftarrow B_1, \ldots, B_n$$

where $A, B_1, \ldots, B_n$ *are atoms. The positive literal* A *is called the head and* $B_1, \ldots, B_n$ *is called the body of the program clause.*

Only atoms are allowed in the body of definite clauses. In Prolog, however, literals of the form *not* B, where B is an atom, are allowed, where *not* is interpreted under the *negation-as-failure* rule.

Definition 2.2 *A program clause is a clause of the form*

$$A \leftarrow L_1, \ldots, L_m$$

where A *is an atom (i.e. a positive literal) and each of* $L_1, \ldots, L_m$ *is of the form* B *or* $\neg B$*, where* B *is an atom.*

Definition 2.3 *A unit clause is a clause of the form*

$$A \leftarrow$$

that is, a program clause (or definite program clause) with an empty body.

In Prolog terminology such a clause is called a *fact* — and sometimes *unconditional definite clause* — and is denoted simply by

$$A.$$

Definition 2.4 *A goal clause is a clause of the form*

$$\leftarrow L_1, \ldots, L_m$$

that is, a clause which has no head. Each L_i $(i = 1, \ldots, m)$ is called a subgoal of the goal clause. It is the pure negation of $L_1, \ldots, L_m$.

Definition 2.5 *A clause is typed if each variable appearing in arguments of clause literals is associated with a set of values the variable can take.*

Definition 2.6 *A database clause is a typed program clause of the form*

$$A \leftarrow L_1, \ldots, L_m$$

where A *is an atom and* $L_1, \ldots, L_m$ *are literals.*

Definition 2.7 *Let* E *be a wff or term. Let vars(E) denote the set of variables in* E*.* E *is said to be ground if and only if vars(E)* $= \emptyset$*.*

Definition 2.8 *A finite set of program clauses is called a logic program.*

Definition 2.9 *In a logic program, the set of all program clauses with the same predicate p in the head (and same arity) is called the definition of p.*

Definition 2.10 *A Horn clause is a clause which is either a program clause or a goal clause. That means that a Horn clause contains at most one positive literal.*

2.2 Learning in ILP

This section presents a formal general setting for learning in ILP.

Definition 2.11 *The task of empirical, single predicate learning in ILP can be formulated as follows [Lavrač 92] [Muggleton 91]*

Given:

- a set of training examples $\mathcal{E}$ described in a language $\mathcal{L}_{\mathcal{E}}$ and consisting of:
  - positive examples, $\mathcal{E}^+$
  - negative examples, $\mathcal{E}^-$

  of an unknown predicate p — target relation,
- a concept description language $\mathcal{L}_C$, specifying syntatic restrictions on the definition of predicate p,
- background knowledge $\mathcal{K}$, described in language $\mathcal{L}_{\mathcal{K}}$, defining predicates q_i (other than p) which may be used in the definition of p and which provide additional information about the arguments of the examples of predicate p,
- a matching operator between $\mathcal{L}_{\mathcal{E}}$ and $\mathcal{L}_C$ wrt $\mathcal{L}_{\mathcal{K}}$ that determines whether an example is covered by a clause expressed in $\mathcal{L}_C$.

Find:

- a definition $\mathcal{H}$ for p, expressed in $\mathcal{L}_C$, such that:
  - $\mathcal{H}$ is complete, i.e., $\mathcal{K} \wedge \mathcal{H} \models^3 \mathcal{E}^+$,
  - $\mathcal{H}$ is consistent, i.e., $\mathcal{K} \wedge \mathcal{H} \not\models \mathcal{E}^-$

  with respect to the examples.

[3]The logical symbol $\models$ stands for semantic entailment [Lloyd 84].

In ILP the languages $\mathcal{L}_{\mathcal{E}}$, $\mathcal{L}_{\mathcal{K}}$ and $\mathcal{L}_{\mathcal{C}}$ used to represent examples, background knowledge and concept descriptions respectively are typically subsets of first order logic, namely logic programs — for a great deal of available ILP systems those languages are the language of Horn clauses.

In an ILP framework the learning of a single concept can be viewed as the learning of a predicate definition (predicate p in Definition 2.11). For some ILP systems the learning of a predicate definition is restricted to the learning of a single clause.

2.3 Searching the Hypotheses Space

Learning from examples can be approached as a process of search in a space of hypotheses which is ordered by the generality relationship between them. The size of the space and the properties of its generalization relation[4] determine the difficulty of finding the concept [Mitchell 82]. In order to deal with such complex hypotheses spaces, some representational aspects of the learning task should be analysed and choices should be made. According to [Kietz 92], choices can be made related to

1. *representation formalism:* it has to do mostly with the language employed for representing the examples, the hypotheses and the background knowledge. Horn clause logic (pure Prolog) and function-free Horn clause programs (Datalog programs) are among the more used formalisms
2. *restricted search subspaces:* frequently the restriction imposed on the representation formalism is not enough for avoiding very complex search problems. So, further restrictions are still necessary — this is accomplished by restricting once more the representational formalism or by using generalization/specialization operators that direct the search process
3. *restricted background subspaces:* the way background knowledge is used has an important influence on the complexity of the learning task. Generally this knowledge should be provided to the system as ground knowledge.
4. *search control:* is related to the use of heuristics to guide the search process in the hypotheses space. It can improve efficiency at the expense of completeness.

In this work we are concerned in ways of restricting background knowledge, so to profit from its use in a framework of inductive learning.

3 Using the Background Knowledge

The background knowledge used in ILP systems must always be restricted otherwise the learning task in first order logic, approached as a search process, inherits the

[4] Such as $\theta -$ ***subsumption*** [Plotkin 71].

undecidability[5] of the deduction process.

3.1 Basic Concepts

Definition 3.1 *A background knowledge $\mathcal{K}$ is ground if it consists of ground unit clauses only.*

Definition 3.2 *The background knowledge $\mathcal{K}$ is of bounded arity if the maximum arity of the predicates in $\mathcal{K}$ is bounded by some constant j.*

The bounded-arity restriction is used in [Page 92] to prove the PAC-learnability of a special kind of hypotheses language $\mathcal{L}_{\mathcal{H}}$ called *constrained clauses.*

Definition 3.3 *A clause is constrained if all variables in the body of the clause also occur in its head.*

A common restriction applied to both, $\mathcal{L}_{\mathcal{K}}$ and $\mathcal{L}_{\mathcal{H}}$, is the restriction to function-free clauses.

Definition 3.4 *A clause is function-free if it has no function symbols; i.e. all of its arguments are either variables or constants (function symbols of arity 0).*

The restriction to function-free ground background knowledge (as well as to function-free ground unit clauses as examples) enables the use of θ – *subsumption* [Plotkin 71] as a complete inference procedure. Function-free knowledge has another positive effect on deduction. It has been proven that inferring ground background knowledge can be done completey in time polynomial to the size of $\mathcal{K}$ (background knowledge), if $\mathcal{K}$ consists only of function-free generative Horn clauses (see Section 3.9, pg. 12) and there is a fixed maximum arity of predicates in the background knowledge [Kietz 93] [Wrobel 87].

3.2 Restricting the Background Knowledge

As stated earlier, a common restriction in ILP systems is the restriction to ground background knowledge and ground unit clauses as examples.

3.2.1 Background Knowledge Given as a Set of Ground Facts

If the logic program P which expresses the background knowledge $\mathcal{K}$ consists of only ground facts, P itself can be chosen as a model[6] M.

[5] A logical system is considered *decidable* if there exists a procedure which can be used to determine if any given formula is valid or not.

[6] An interpretation which turns a formula valid.

3.2.2 The H-easy Model

If the logic program P which expresses the background knowledge consists of arbitrary clauses, the model M for P can be infinite. In order to deal with situations like that and restrict infinite models to finite ones, the notion of *h-easiness* was established and is presented next. The definitions and theorems that follows were extracted from [Muggleton 90].

Definition 3.5 *Given a logic program P, an atom* **a** *is h-easy with respect to P, for a given natural number h, iff there exist a derivation of* **a** *from P involving at most h binary resolutions*[7].

Definition 3.6 *The Herbrand h-easy model of P — M_h — is the set of all Herbrand instantiations*[8] *of h-easy atoms of P.*

Theorem 3.1 *Given a logic program P, for any finite h, the number of h-easy atoms of P is finite.*

Proof: This follows from the fact that there are only a finite number of such derivations.

Example 3.1 *For a given non-ground program P, representing a certain background knowledge $\mathcal{K}$, the computation of M_h for $h = 5$ is shown next. Factual clauses of the program are pictorially shown in Figure 1.*

Let P be the following Prolog program:

(1)	$parent(beth, tim).$	
(2)	$parent(fred, tim).$	
(3)	$parent(jackie, vicky).$	
(4)	$parent(tim, vicky).$	
(5)	$parent(vicky, anna).$	
(6)	$parent(vicky, rachel).$	
(7)	$parent(martin, anna).$	
(8)	$parent(martin, rachel).$	
(9)	$parent(rachel, vera).$	
(10)	$parent(rachel, sean).$	
(11)	$parent(ed, vera).$	
(12)	$parent(ed, sean).$	
(13)	$predecessor(X, Z) :-$	$parent(X, Z).$
(14)	$predecessor(X, Z) :-$	$parent(X, Y),$ $predecessor(Y, Z).$

The predicate *predecessor/2* is written in clausal form as

(13) $predecessor(X, Z) \vee \neg parent(X, Z)$
(14) $predecessor(X, Z) \vee \neg parent(X, Y) \vee \neg predecessor(Y, Z)$

[7] Observe that this definition implies that facts in a logic program P are 1-easy wrt P since 1 binary resolution is required to derive each fact.

[8] See [Nicoletti 93] [Nicoletti 93a].

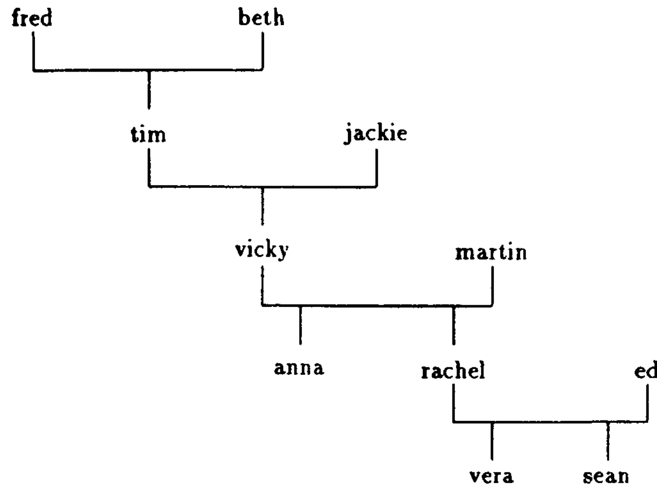

Figure 1: Family relation

The set of 1-easy atoms with respect to P coincides with the Herbrand 1-easy model of P — M_1. Since no Herbrand instantiations of 1-easy atoms of P were performed, all the 1-easy atoms obtained were ground atoms.

$$\begin{aligned} M_1 = \; & \{parent(beth, tim), parent(fred, tim), parent(jackie, vicky), \\ & parent(tim, vicky), parent(vicky, anna), parent(vicky, rachel), \\ & parent(martin, anna), parent(martin, rachel), parent(rachel, vera), \\ & parent(rachel, sean), parent(ed, vera), parent(ed, sean)\} \end{aligned}$$

M_2 is the set of all Herbrand instantiations of 2-easy atoms of P. Recall that by definition an atom a is 2-easy with respect to P iff there exist a derivation of a from P involving at most 2 binary resolutions. Next table shows all possible derivations of program P involving 2 binary resolutions.

$Clause_1$	$Clause_2$	Resolvent (2 binary resolution)	
(1)	(13)	*predecessor(beth, tim)*	(15)
(2)	(13)	*predecessor(fred, tim)*	(16)
(3)	(13)	*predecessor(jackie, vicky)*	(17)
(4)	(13)	*predecessor(tim, vicky)*	(18)
(5)	(13)	*predecessor(vicky, anna)*	(19)
(6)	(13)	*predecessor(vicky, rachel)*	(20)
(7)	(13)	*predecessor(martin, anna)*	(21)
(8)	(13)	*predecessor(martin, rachel)*	(22)
(9)	(13)	*predecessor(rachel, vera)*	(23)
(10)	(13)	*predecessor(rachel, sean)*	(24)
(11)	(13)	*predecessor(ed, vera)*	(25)
(12)	(13)	*predecessor(ed, sean)*	(26)
(1)	(14)	$predecessor(beth, Z) \vee \neg predecessor(tim, Z)$	(27)
(2)	(14)	$predecessor(fred, Z) \vee \neg predecessor(tim, Z)$	(28)
(3)	(14)	$predecessor(jackie, Z) \vee \neg predecessor(vicky, Z)$	(29)
(4)	(14)	$predecessor(tim, Z) \vee \neg predecessor(vicky, Z)$	(30)
(5)	(14)	$predecessor(vicky, Z) \vee \neg predecessor(anna, Z)$	(31)
(6)	(14)	$predecessor(vicky, Z) \vee \neg predecessor(rachel, Z)$	(32)
(7)	(14)	$predecessor(martin, Z) \vee \neg predecessor(anna, Z)$	(33)
(8)	(14)	$predecessor(martin, Z) \vee \neg predecessor(rachel, Z)$	(34)
(9)	(14)	$predecessor(rachel, Z) \vee \neg predecessor(vera, Z)$	(35)
(10)	(14)	$predecessor(rachel, Z) \vee \neg predecessor(sean, Z)$	(36)
(11)	(14)	$predecessor(ed, Z) \vee \neg predecessor(vera, Z)$	(37)
(12)	(14)	$predecessor(ed, Z) \vee \neg predecessor(sean, Z)$	(38)

It can be seen that only the 12 first derivations involving 2 binary resolutions (numbered 15 - 26) derive an atom (all of them are ground atoms in this example), so the set of 2-easy atoms with respect to P (which includes 1-easy atoms as well) coincides with the Herbrand 2-easy model of P — M_2.

$$M_2 = M_1 \bigcup \{predecessor(beth, tim), predecessor(fred, tim), predecessor(jackie, vicky), predecessor(tim, vicky), predecessor(vicky, anna), predecessor(vicky, rachel), predecessor(martin, anna), predecessor(martin, rachel), predecessor(rachel, vera), predecessor(rachel, sean), predecessor(ed, vera), predecessor(ed, sean)\}$$

Next tables present the construction of 3-easy, 4-easy and 5-easy set of atoms with respect to P, which coincide with its respectives h-easy models, namely M_3, M_4 and M_5 (the coincidence is due to the fact that the h-easy sets of atoms are ground, so, no Herbrand instantiation takes place in order to generate M_h).

$Clause_1$	$Clause_2$	Resolvent(3 binary resolutions)	
(18)	(27)	*predecessor(beth, vicky)*	(39)
(18)	(28)	*predecessor(fred, vicky)*	(40)
(19)	(29)	*predecessor(jackie, anna)*	(41)
(20)	(29)	*predecessor(jackie, rachel)*	(42)
(19)	(30)	*predecessor(tim, anna)*	(43)
(20)	(30)	*predecessor(tim, rachel)*	(44)
(23)	(32)	*predecessor(vicky, vera)*	(45)
(24)	(32)	*predecessor(vicky, sean)*	(46)
(23)	(27)	*predecessor(martin, vera)*	(47)
(24)	(27)	*predecessor(martin, sean)*	(48)

$$M_3 = M_2 \bigcup \{predecessor(beth, vicky), predecessor(fred, vicky), predecessor(jackie, anna), predecessor(jackie, rachel), predecessor(tim, anna), predecessor(tim, rachel), predecessor(vicky, vera), predecessor(vicky, sean), predecessor(martin, vera), predecessor(martin, sean)\}$$

$Clause_1$	$Clause_2$	Resolvent(4 binary resolutions)	
(27)	(43)	$predecessor(beth, anna)$	(49)
(27)	(44)	$predecessor(beth, rachel)$	(50)
(28)	(43)	$predecessor(fred, anna)$	(51)
(28)	(44)	$predecessor(fred, rachel)$	(52)
(29)	(45)	$predecessor(jackie, vera)$	(53)
(29)	(46)	$predecessor(jackie, sean)$	(54)
(30)	(45)	$predecessor(tim, vera)$	(55)
(30)	(45)	$predecessor(tim, sean)$	(56)

$$M_4 = M_3 \bigcup \{predecessor(beth, anna), predecessor(beth, rachel), predecessor(fred, anna), predecessor(fred, rachel), predecessor(jackie, vera), predecessor(jackie, sean), predecessor(tim, vera), predecessor(tim, sean)\}$$

$Clause_1$	$Clause_2$	Resolvent (5 binary resolutions)	
(27)	(55)	$predecessor(beth, vera)$	(57)
(27)	(56)	$predecessor(beth, sean)$	(58)
(28)	(41)	$predecessor(fred, vera)$	(59)
(28)	(42)	$predecessor(fred, sean)$	(60)

$$M_5 = M_4 \bigcup \{predecessor(beth, vera), predecessor(beth, sean), predecessor(fred, vera), predecessor(fred, sean)\}$$

Example 3.2 *Let P be the following Prolog program*

$$\begin{array}{ll} p(X) :- & q(X). \\ q(Z). & \end{array}$$

This program has as Herbrand Universe and Base respectively the sets

$$\begin{array}{lll} H & = & \{a\} \\ H_B & = & \{p(a), q(a)\} \end{array}$$

and the following sets

- set of 1-easy atoms of $P : \{q(Z)\}$
- set of all Herbrand instantiations of 1-easy atoms of P

$$M_1 = \{q(a)\}$$

- set of 2-easy atoms of $P : \{p(Z), q(Z)\}$
- set of all Herbrand instantiations of 2-easy atoms of P

$$M_2 = \{p(a), q(a)\}$$

Example 3.3 *Let P be the following Prolog program*

$$p(X) :- \quad q(f(X)).$$
$$q(Z).$$

This program has as Herbrand Universe and Base respectively the sets

$$\begin{array}{rcl} H & = & \{a, f(a), f(f(a)), f(f(f(a))), \ldots\} \\ H_B & = & \{a, p(a), q(a), p(f(a)), q(f(a)), p(f(f(a))), q(f(f(a))), \ldots\} \end{array}$$

and the following sets

- set of 1-easy atoms of $P : \{q(Z)\}$
- set of all Herbrand instantiations of 1-easy atoms of P

$$M_1 = \{q(a), q(f(a)), q(f(f(a))), \ldots\}$$

- set of 2-easy atoms of $P : \{p(Z), q(Z)\}$
- set of all Herbrand instantiations of 2-easy atoms of P

$$M_2 = \{p(a), q(a), p(f(a)), q(f(a)), \ldots\}$$

Example 3.3 illustrates the fact that a program P which has a finite set of h-easy atoms can have its correspondent Herbrand Model M_h infinite.

Since there is a particular interest in generating finite models of programs, next it will be introduced the notion of generative clauses [Muggleton 90], a restricted form of Horn clause program, which will allow the construction of a finite model associated with a program. In fact, if the background knowledge is expressed as generative clauses, it is possible to construct a finite model associated with it.

3.2.3 Generative Clauses

Next it will be defined a restricted form of Horn clause program P for which the Herbrand h-easy model is finite.

Definition 3.7 *The n-atomic-derivation set of program P, $D^n(P)$ is defined recursively as follows*[9]

- $D^0(P) =$ set of unit clauses in P
- $D^n(P) = D^{n-1}(P) \bigcup \{A\theta_1 \ldots \theta_n : A \leftarrow B_1, \ldots, B_m \in P$ *and for each* B_i $\exists\, B_i' \in D^{n-1}(P)$ *such that* θ_i *is the mgu*[10] *of* B_i *and* $B_i'\}$

[9]Observe that with the definitions presented here, in fact $M_h = D^{h-1}(P)$.

The atomic-derivation-closure is defined as:

$$D^*(P) = D^0(P) \bigcup D^1(P) \bigcup \dots$$

Definition 3.8 *The logic program P is said to be semantically generative iff every element of $D^*(P)$ is ground.*

The following syntactic constraint on logic programs corresponds to this semantic definition.

Definition 3.9 *The clause $A \leftarrow B_1, \dots, B_m$ is syntactically generative*[11] *whenever the variables in the head A are a subset of the variables in the body $B_1, \dots, B_m$. The logic program P is syntactically generative*[11] *iff every clause in P is syntactically generative.*

Theorem 3.2 *Every syntactically generative logic program P is semantically generative.*

Proof:

- For $n = 0$, $D^0(P)$ contains the unit clauses of P. Those unit clauses are ground since the logic program P is syntactically generative
- Suppose every element of $D^{n-1}(P)$ is ground
- Let $A \leftarrow B_1, \dots, B_m$ be a clause in P and $A\theta_1\theta_2 \dots \theta_m$ be an element of $D^n(P) - D^{n-1}(P)$ such that for each B_i there exists $B_i' \in D^{n-1}(P)$ for which θ_i is the mgu of B_i and B_i'.

  By the inductive assumption each $B_i'\ in D^{n-1}(P)$ is ground thus each $B_i\theta_i$ is ground.

  Since each θ_i is a ground substitution and every variable in A is found in the domain of $\theta_1\theta_2 \dots \theta_m$ it follows that $A\theta_1\theta_2 \dots \theta_m$ is ground, which completes the proof.

It is important to notice that the Least Herbrand Model and the atomic-derivation-closure of a semantically generative logic program are the same. This is illustrated by this following example [Hooger 90]

Example 3.4 *Let a clause set P be*

$likes(chris, X)$ **if** $likes(X, logic)$
$likes(bob, logic)$

[10] most general unifier.
[11] Or range-restricted.

The Herbrand Universe for P is the set $H = \{chris, bob, logic\}$ and the ground instantiation of P, namely G_P is

$$likes(chris, chris) \textbf{ if } likes(chris, logic)$$
$$likes(chris, bob) \textbf{ if } likes(bob, logic)$$
$$likes(chris, logic) \textbf{ if } likes(logic, logic)$$
$$likes(bob, logic)$$

which can be rewritten, for facility, as

$$CC \textbf{ if } CL$$
$$CB \textbf{ if } BL$$
$$CL \textbf{ if } LL$$
$$BL$$

The Herbrand Base is

$$H_B = \{CC, CL, CB, LC, LL, LB, BC, BL, BB\}$$

The set $\{BL, CB\} \subset B_P$ is thus a Herbrand interpretation. For this interpretation the ground instantiation G_P is reproduced below with the associated truth-values written in place of its atoms.

$$false \textbf{ if } false$$
$$true \textbf{ if } true$$
$$false \textbf{ if } false$$
$$true$$

With this assignment, all the clauses of G_P are true. This interpretation is therefore an Herbrand Model for G_P and hence an Herbrand Model for P, and is the Least Herbrand Model for P.

This last program is a semantically generative logic program because it is a syntactically generative logic program. Its atomic-derivation-closure is

$$D^*(P) = D^0(P) \bigcup D^1(P) = \{BL\} \bigcup \{CB\} = \{BL, CB\}$$

Theorem 3.3 *For every syntactically generative logic program P and every h, $M_h(P)$ is finite.*

Corollary 3.1 *For every syntactically generative logic program P and set of examples $e_1, \ldots, e_n$, the rlgg*[12] *of $e_1, \ldots, e_n$ with respect to $M_h(P)$ is finite and has length at most $| M_h(P) |^n + 1$.*

[12] Relative Least General Generalization [Muggleton 90].

4 Transforming Intentional Background Knowledge into Extensional Background Knowledge

As stated in Definition 2.9, pg. 4, a predicate definition p is the set of all program clauses with the same predicate p in the head (and same arity). A predicate can be defined *extensionally* as a set of ground facts or *intensionally* as a set of database clauses [Lavrač 94]. Some ILP systems, such as *GOLEM* and *FOIL* require all background knowledge to be given extensionally.

The only way to use an intensionally specified background knowledge P in *GOLEM* or *FOIL* is by using an extensional version of this knowledge. This version can be obtained by deriving background atoms that are relevant to the examples, as a pre-processing step of the learning task.

After having verified that the given background knowledge consists only of generative clauses, it is possible to construct the finite model associated with it, by constructing its *atomic-derivation-closure*. For this, it is necessary an algorithm to find all possible conclusions of program P. This can be done starting with facts and reasoning to goals, which is known as *forward chaining* or *modus ponens* reasoning. For example, given the following program P

$$p(X,Y) : -q(X), p(Y).$$
$$q(a).$$
$$p(b).$$

by modus ponens it can be concluded that *p(a,b)* is also a fact.

In general, to use modus ponens as the basis of a control structure, it is necessary to consider the facts in order and, for each fact F, it is necessary to find all clauses

$$A_i \leftarrow L_{i_1}, L_{i_2}, \ldots, L_{i_{n_i}}.$$

such that some of the literals L_{i_j} can be matched with the fact F. For those clauses, a new clause is generated from the old one but without this fact. Whenever a fact matches the unique literal in the body of a clause, the head of the clause has been proven a new fact. One important point to consider is the order in which new facts are pursued. New facts can be added in front of those not yet considered, such that the fact just proven will be the next fact to be considered.

Although this idea is good when it is necessary to reach some particular conclusion as fast as possible, it is not appropriated if it is necessary to systematically find every possible conclusion from the facts. For this latter case, it is better to put new proven facts at the end of the set of facts, i.e., the fact list is much like a queue (last in last out).

Another point which should be considered are negated atoms in the body of a clause. The algorithm presented next follows the *closed-world assumption*, so $\neg L$ (negation

of L) means L *cannot be proven*. For this, the forward chaining algorithm must first assume that all negations are false, prove all possible facts and only consider as true those negations which still are not facts. It should be observed that those negations being true, can possibly prove new facts with new consequences.

4.1 Algorithm for Verifying if a Logical Program is Generative

Let a logical program P be defined as the set of clauses $\{C_1, C_2, \ldots, C_m\}$ of the form

$$C_i : A_i \leftarrow L_{i_1}, L_{i_2}, \ldots, L_{i_{n_i}} \quad \forall i = 1, \ldots, m \text{ and } n_i \geq 0$$

Let $variables(L_{ij})$ be the set of variables occuring in literal L_{ij}.

```
function generative(P) : boolean
    generative ← true
    begin
        for k = 1 to m do
        begin
            B = ∅
            H = variables(A_k)
            for p = 1 to n_k do
                B = B ⋃ variables(L_kp)
            if H ⊄ B then
              begin
                generative = false
                exit
              end
        end
    end
```

on exit, if *generative(P)* is *true* then the logical program P consists only of generative clauses.

4.2 Algorithm for Generating the Atomic Closure of a Generative Logical Program

Next an algorithm for generating the atomic closure (Definition 3.7, pg. 11) of a generative logical program P, is presented. Let a logical program P be defined as the set of generative clauses $\{C_1, C_2, \ldots, C_m\}$ of the form

$$C_i : A_i \leftarrow L_{i_1}, L_{i_2}, \ldots, L_{i_{n_i}} \quad \forall i = 1, \ldots, m \text{ and } n_i \geq 0$$

```
unused_queue_fact ← queue of all facts
used_queue_fact ← ∅
queue_clauses ← queue of all clauses C (which are not unit clauses)
   repeat
      while unused_queue_fact ≠ ∅ do
        begin
          let F be the first literal in unused_queue_fact
          for each C in queue_clause which can match F with a literal[13] in its body do
            begin
              create a new clause C' from C removing the literal F after the necessary
              bindings for the variables in C have been done to make the match possible
                if C' is a unit clause
                  then add C' as the last element of unused_queue_fact
                  else add C' as the first element of queue_clause
            end
        end
      for each negated atom ¬B of clauses in queue_clauses such that B does not match any
      atoms in used_queue_fact add ¬B as the first element of unused_queue_fact
   until unused_queue_fact = ∅
```

5 Conclusions

In this work we have discussed ways of incorporating background knowledge into the learning task as well as restricting the background knowledge description language such that the search space is limited, thus making the learning task feasible.

A restricted form of Horn clause program — generative program — is discussed in details since, if the background knowledge $\mathcal{K}$ can be expressed as a generative program, there is a finite model associated with $\mathcal{K}$

An algorithm to verify if a given background knowledge $\mathcal{K}$ consists of generative clauses as well as the algorithm for constructing the finite model associated with it are presented. Both algorithms are currently being implemented in Prolog.

Acknowledgments: to Dr. Doris Ferraz de Aragon who read the preliminar version of this work and provided significant comments and advices.

References

[Cohen 93] Cohen, W.W. *Learnability of Restricted Logic Programs.* Proceedings of the Third International Workshop on Inductive Logic Programming — ILP'93, Bled, Slovenia, April, 1993, pp 1-3.

[13] If there are negated atoms in the body of the clauses, after the first iteration *unused_queue_fact* may contain the negation of a fact proved under the *closed-world assumption*.

[Džeroski 92] Džeroski, S.; Muggleton, H.S.; Russell, S. *PAC - Learnability of Determinate Logic Programs.* In Proceedings of ISSEK Workshop'92, Slovenia, September, 1992.

[Hooger 90] Hooger, C.J. *Essentials of Logic Programming.* Clarendon Press, Oxford, 1990.

[Kietz 93] Kietz, J-U. *Some Lower Bounds for the Computational Complexity of Inductive Logic Programming.* Lecture Notes in Artificial Intelligence 667, Pavel B. Bradzil (ed.), Springer-Verlag, 1993, pp 115-123.

[Kietz 92] Kietz, J-U.; Wrobel S. *Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models.* Inductive Logic Programming, S. Muggleton (ed.), Academic Press, 1992.

[Kowalski 87] Kowalski, R.A.; Hogger, C.J. *Logic Programming.* Encyclopedia of Artificial Intelligence. S.C.Shapiro; D. Eckroth; G.A. Vallasi (eds.), John Wiley & Sons, New York, 1987, pp 544-558.

[Lavrač 92] Lavrač, N.; Džeroski, S. *Background Knowledge and Declarative Bias in Inductive Concept Learning.* Lectures Notes on Artificial Intelligence 462, 1992, pp. 51-71.

[Lavrač 94] Lavrač, N.; Džeroski, S. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, London, 1994.

[Lloyd 84] Lloyd, J.W. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Michalski 83] Michalski, R.S.; Larson, J. *Selection of Most Representative Training Examples and Incremental Generation of VL1 Hypotheses: the Underlying Methodology and the Description of Programs ESEL and AQ11.* Report ISG 83-5, Dept. of Computer Science, University of Illinois, Urbana, USA, 1983.

[Mitchell 82] Mitchell, T.M. *Generalization as Search.* Artificial Intelligence 18(2), 1982, pp 203-226.

[Muggleton 90] Muggleton, S.H.; Feng, C. *Efficient Induction of Logic Programs.* TIRM-90-044, Turing Institute Press, Glasgow, October 1990.

[Muggleton 91] Muggleton, S.H. *Inductive Logic Programming.* New Generation Computing, Vol. 8, 1991, pp 295-318.

[Nicoletti 93] Nicoletti, M.C.; Monard, M.C. *Herbrand Interpretation, Model and Least Model within the Framework of Logic Programming.* Notas do ICMSC-USP, Série Computação Nº 2, junho 1993, 30 pg.

[Nicoletti 93a] Nicoletti, M.C.; Monard, M.C. *Learning Horn Clauses Using the ILP System GOLEM.* Notas do ICMSC-USP, Série Computação, Nº 4, novembro 1993, 35 pg.

[Page 92] Page, C.D.; Frish A.M. *Generalisation and Learnability: a Study in Constrained Atoms.* In S. Muggleton (ed.), Inductive Logic Programming, Academic Press, 1992, pp 29-61.

[Plotkin 71] Plotkin G.D. *Automatic Methods of Inductive Inference.* Ph. D. thesis, Edinburgh University, August 1971.

[Quinlan 79] Quinlan, J. R. *Discovering Rules by Induction from Large Collections of Examples.* In D. Michie (ed.), Expert systems in the Microeletronic Age, Edinburgh University Press, 1979.

[Wrobel 87] Wrobel, S. *Higher-order Concepts in a Tractable Knowledge Representation.* In K. Morik, (ed.), GWAI-87 11th German Workshop on Artificial Intelligence, Informatik-Fachberichte Nr. 152, Springer-Verlag, October 1987, pp 129-138.

N O T A S D O I C M S C

Serie : Computacao

Nº 004/93 - NICOLETTI, M.N.; MONARD, M.C. - Learning horn clauses using the ILP system GOLEM

Nº 003/93 - ARENALES, M.N.; MORABITO, R.N. - An and/or-graph approach to the solution of two-dimensional non-guillotine cutting problems

Nº 002/93 - NICOLETTI, M.C.; MONARD, M.C. - Herbrand interpretation model and least within the framework of logic programming

Nº 001/93 - MORABITO, R.; ARENALES, M.N. - An and/or graph approach to the container loading problem