An Algorithm for Incremental Mode Induction

Nicola Di Mauro, Floriana Esposito, Stefano Ferilli, and Teresa M.A. Basile

Dipartimento di Informatica, Università di Bari - Italy
{nicodimauro,esposito,ferilli,basile}@di.uniba.it

Abstract. Learning systems have been devised as a way of overcoming the knowledge acquisition bottleneck in the development of knowledge-based systems. They often cast learning to a search problem in a space of candidate solutions. Since such a space can grow exponentially, techniques for pruning it are needed in order to speed up the learning process. One of the biases used by Inductive Logic Programming (ILP) systems for this purpose is mode declaration. This paper presents an algorithm to incrementally learn this type of meta-knowledge from the available observations, without requiring the final user's intervention.

1 Introduction

The knowledge acquisition process is widely known to be a serious bottleneck in the development of intelligent systems, such as expert systems and decision support systems. For this reason, great interest has been devoted in the implementation of learning systems that are able to automatically induce the needed knowledge from observations. If, on the one hand, efficiency is a critical factor for the success of such systems, on the other hand reaching high performance is often strongly dependent on good parameter settings, which in turn require deep technical knowledge and can be hardly done by the final users. This is why it would be highly desirable to set automatically the parameters of the learning systems.

Inductive Logic Programming (ILP) is an established sub-field of Machine Learning aimed at inducing first-order clausal theories from examples. This objective can be cast to a search problem in a hypotheses space, that may lead to serious efficiency problems when the complexity of the learning task and/or of the description language used causes an explosion in the size of such a space. For this reason, many ILP systems are designed to use, if available, various forms of *meta-knowledge* about the hypotheses to be learned in order to make the search more efficient.

In a general ILP scenario, given a description language, unary predicates represent values of the objects properties, while n-ary predicates represent associations among objects, and hence raise the problem of understanding which of their arguments must be supplied as input and which ones they return as a result of their computation (*mode* of the predicate), according to specific use of the predicates in a given context. Such issue is well-known, for instance, in Prolog, where the procedural interpretation re quires predicates' arguments to be used properly. This holds not only for predicates having a unique use, but also for reversible predicates, in which different combinations of input and output arguments are possible. However, in Prolog the possible uses of built-in and library predicates are pre-defined, or known to the programmer in the case of user-defined ones. The situation is different in first-order machine learning, where the available information (examples, observations, background knowledge, etc.) could be provided by sources that are extraneous to the experimenter, and thus the mode of the predicates in the description language could be unknown.

A *mode declaration* [2] is a specification of the predicate modes. It plays a central role in the efficiency improvement of some systems (e.g., Progol [12], GOLEM [10], FOIL [11]), and it is required, in particular, by systems designed to learn recursive definitions (such as TIM [14] and MRI [13]). This paper presents an algorithm that induces *mode declarations* from examples of the concept to be learned. Its main novelty lays in its *incremental behaviour*, useful when the knowledge is not completely available at the beginning of the learning process: in fact, it works without requiring a previous theory for the target concept. It can be embedded in a natural way in ILP systems as a pre-processor.

In Section 2 some significant previous works on Automatic Mode Inference in an ILP problem are presented. Section 3 introduces the basic concepts needed to understand the proposed approach and presents the new algorithm. In Section 4 the main characteristics and the behavior of the algorithm are presented by means of examples. Finally, Section 5 draws some conclusions.

2 Related Work

Formally, a *mode* of an *n*-ary predicate symbol is an *n*-tuple that represents a possible instantiation of arguments of that predicate symbol in terms of some domain. Each element of such a domain corresponds to a degree of instantiation of an argument of the predicate symbol. In this work we set such a domain to {+,–}, where '+' means that the argument must be instantiated, and '–' expresses the fact that the argument is returned after the predicate computation.

Generally speaking, in a logic program there is not the concept of "input" and "output" variable: each variable might in principle be used either as an input or as an output argument, and programs might be executed in either a "forward" or a "backward" direction. Nevertheless, the need of mode declarations may arise in Logic Programming for different reasons. For instance, knowledge about which arguments in a predicate must be specified and which ones are returned after computation can help programmers in verifying program correctness. Moreover, compilers can profitably use it for optimization purposes (e.g., using efficient special-purpose unification routines instead of the general unification algorithm): indeed, it is often the case that, in a particular program, a predicate is executed in one direction only, i.e. it is always called with a particular set of its variables that are bounded (the "input" variables) and the remaining set unbounded (the "output" variables).

Traditionally, the task of supplying this kind of information has been in charge of the programmer. In such a case, however, problems might arise due to the errors made by the programmer in declaring modes that can lead to very strange program behavior, whose cause can be hard to find. A possible solution may be letting the compiler infer the modes, using them either to optimize a program without mode declarations, or to verify the declarations made by the programmer, in a similar fashion to type-checkers used in other languages to verify type declarations. The issue of automatic mode inference for these purposes has been considered in [3], [4], [5], [6] and [7]. All these works deal with a Prolog program (i.e., a set of definite Horn clauses) together with a negative clause (called *query*), where the mode of a predicate in the program indicates how its arguments will be instantiated when that predicate is called. Thus, the modes of a program represent statements about all computations that are possible from it [8].

Another use of mode declarations, as introduced in Section 1, comes from the ILP where the modes are exploited to build systems that are able to explore the space of hypotheses more efficiently. An algorithm to extract modes from data was presented in [1] and implemented in LIME [15]. In LIME the search space is restricted to *determinate clauses*; in the absence of any information, each time a literal is added to a clause the system must assume that unbounded variables will be uniquely bounded. As this process is essentially the same each time it is carried out, considerable improvement in performance can be achieved if mode information is available. LIME extracts such information from the data, this way being able to skip clauses that are not determinate. It considers predicates one at a time and assumes that any possible mode declaration for that predicate has functional dependency. Then this assumption is checked against the data: if it is false (i.e., the data show a counterexample), then the algorithm discards that possible functional dependency. Another system that faced these issues is MOBAL [9].

Example 1. Consider predicates `append(l1,l2,l3)` ("`l3` is the list obtained appending the list `l2` to the list `l1`") and `decomp(l,e,r)` ("`l` is the list with head `e` and tail `r`"). Mode declarations `{append(+,+,-),append(+,-,+),append(-,+,+),decomp(+,-,-),decomp(-,+,+)}` are not consistent with `append`$(X_1,X_2,X_3) \leftarrow$ `decomp`(X_1,X_2,X_3), and hence the search space can be pruned by eliminating such a clause.

3 MILE: Mode-Declarations Incremental Learner from Examples

In this section, after giving some preliminary definitions and a general idea of the proposed technique for inducing mode declarations from a set of examples for some concept (predicate), we present the detailed algorithm.

Examples are ground Horn clauses and the hypotheses space consists of *function free* clauses, i.e. terms in the head of the target clauses and terms in the literals of the body can be either variables or constants. This simplifies the structure of each clause without affecting expressiveness [16]. Each clause consists of a head literal followed by a list of body literals. Each literal in the body of a clause must use at least one term (variable or constant) that has been introduced by a previous literal in the body. Hence, the list of literals in the body can be ordered according to the following definition.

Definition 1. (Layers of a clause) *Given a linked clause $h \leftarrow b_1,b_2,\dots,b_n$: the set of* 1-layer literals *consists of all b_i's, $1 \leq i \leq n$, such that b_i shares some argument with h. The arguments of the literal h are called* 1-layer activated terms*; the set of* k-layer literals *consists of all b_i's such that b_i is not a t-layer literal for $1 \leq t < k$ and it shares some of its arguments with the set of (k–1)-layer activated terms. The set* of k-layer activated terms *(also known as* variable depth *[12]) is the union of all k-layer literals' terms that are not (k–1)-layer activated terms. This notion induces an equivalence relation on the set of literals in a given clause.*

Example 2. The literals in the body of the following Horn clause:

```
h(X) :- p(X,Y),p(X,Z),p(X,W),o(Y,Z),q(Z,T),s(T),l(W).
```

can be ordered in layers (equivalence classes). The set of *1-layer activated terms* is `{X}`, that contains the only argument that is present in the head of the clause; `p(X,Y)`, `p(X,Z)`, `p(X,W)` are the *1-layer literals*, since they contain the variable `X` as argument. Hence, the set of *2-layer activated terms* is `{Y,Z,W}`. Then, `o(Y,Z)`, `q(Z,T)`, `l(W)` are the *2-layer literals* and `{T}` is the set of *3-layer activated terms*. Finally, `s(T)` is the only *3-layer literal.*

According to the identified layers, the above clause can be rewritten as follows:

```
h(X) :- [  [ p(X,Y), p(X,Z), p(X,W) ],      % X
           [ o(Y,Z), q(Z,T), l(W) ],        % Y Z W
           [ s(T) ] ].                      % T
```

After introducing how to represent a clause by means of layers, we can give the general idea behind MILE. Such an algorithm can be used as a preprocessor of the examples given as input to an ILP system. It is incremental but obviously it can be exploited also by batch systems.

Given a Horn clause:

1. select a possible mode declaration[1] for the literal in its head and consider the corresponding input arguments as *1-layer activated terms*;
2. for $k \geq 1$ repeat the following:
   1.1. collect the set of *k-layer literals*
   1.2. consider as input arguments those that correspond to *k-layer activated terms* and as output arguments the remaining ones.

It is important to note that, at any step (corresponding to a layer), if an output term is introduced by more than one literal (i.e., there are many *k*-layer literals with the

[1] Given a literal $p(x_1,x_2, \dots ,x_n)$, there are 2^n-1 possible mode declarations, excluding the mode that considers all arguments as output ones (we assume that there must be always at least one input argument according to which computing the others). For instance, predicate *p/3* has $(2^3$-$1)$=7 possible modes, specifically: (+,+,+),(+,+,–),(+,–,+),(–,+,+),(+,–,–),(–,+,–),(–,–,+).

Algorithm 1. *MD(Bound,Lits,Modes):NewModes*

```
/* Bound: set of terms; Lits: list of literals;
Modes: set of set of mode declarations: Modes = {$m_1$, ...,$m_n$} where $m_i$ is a set of modes*/

if Lits ≠ []
  Bound' ← ∅; Lits' ← Lits
  forall $b_i$∈Lits that shares some of its arguments with Bound do
    if $b_i$ is not a linked-layer literal then
      $M_i$ is the mode of $b_i$ obtained by setting its arguments that are in Bound as
        input (+) and the others as output (–)
      if consistent($M_i$,Modes) then
        Remove $b_i$ from Lits' and add to Bound' the output terms of $b_i$
      else remove inconsistency from Modes
    else Link ← list of all literals linked-layer to $b_i$
      forall $b_j$ ∈ Link do
        $M_j$ is the mode of $b_j$ obtained by setting its arguments that are k-layer
          activated terms as input (+) and the others as output (–)
        if consistent($M_i$,Modes) then
          Remove $b_i$ from Lits' and add to Bound' the output terms of $b_j$
        else remove inconsistency from Modes and exit
        if Modes ≠ ∅
          NewModes ← MD(Bound',Lits',Modes)
        else fail
else NewModes ← Modes
return(NewModes)
```

same term marked as "output" by step 1.2), it is not possible to decide which of them actually introduces it. For example, if `p(X,Y)` and `q(X,Y)` are two *k*-layer literals and `X` is a *k*-layer activated term, then it is necessary to assess if `Y` is produced by `p` or by `q`. In these cases, the algorithm maintains all the possible alternatives, waiting for the next steps and/or examples to (hopefully) *disambiguate* the correct one.

Example 3. Given the following clause:

```
  h(a,b) :- p(a,b), p(b,c), o(a,b,c).
```

and hypothesizing mode (+,–) for the head literal, there are two *1*-layer literals, `p(a,b)` and `o(a,b,c)`, that have the same term `b` as a possible output. In this case, the available information was not sufficient to identify a single mode for predicates `p/2` and `o/3` given the mode (+,–) for the predicate `h/2`. Indeed, the list of modes induced by MILE for the above clause is

```
  md([[ h(-,+), [ [p(+,+),o(-,+,-)]]],
       [ h(+,-), [ [p(+,+),o(+,-,-)], [p(+,-),o(+,+,+)]]],
       [ h(+,+), [ [p(+,+),o(+,+,-)]]]]).
```

Algorithm 2. MILE(E)

```
$\boldsymbol{M} = \varnothing$
**forall** $e \in E$ ***do***
  $(h/n) \leftarrow$ predicate in the head of $e$
  **forall** possible modes $M_{h/n}$ of the predicate $h/n$ **do**
    *Bound* $\leftarrow$ input terms of $h$
    *Lits* $\leftarrow$ the body of $e$
    **if** $m(M_{h/n}, MD(Bound, Lits, \varnothing))$ is consistent with $\boldsymbol{M}$
      update $\boldsymbol{M}$ accordingly
    **else**
      **fail**
```

Note that a mode is related to the concept described by the predicate it belongs to (i.e., to its context[2]). The term *disambiguation* refers to the assumption that there is a unique mode for each predicate in a given context (i.e., each predicate is used always with the same meaning – it is a function). Thus, the algorithm is able to learn *many modes for the same predicate*, even if just one for each fixed context.

Definition 2. ***(linked-layer literal)*** *A k-layer literal p is called* linked-layer literal *if there exists a k-layer literal q that shares some argument with p and such arguments are not k-layer activated terms; in this case, we also say that p is* linked-layer to *q*.

Algorithm 1 describes how MILE learns mode declarations for the predicates in the body of an example '*h :- Body*'. Let us consider the list $Lits=(l_1,\ldots,l_n)$ of such body literals, the set *Bound* of 1-layer activated terms (input arguments of *h*), and an initial set of mode declarations *Modes* ($=\varnothing$ at beginning for a new example). For each literal $l_i \in Lits$ that shares some arguments with the set *Bound*: **a)** if l_i is not a linked-layer literal then its mode considers as input terms all those that are in *Bound*, and as output the others; **b)** if l_i is a linked-layer literal, then all the literals that are linked-layer to it are collected, and all possible modes for them are considered (see Example 3). Each new mode is added to the set *Modes*, and all elements of *Modes* that are inconsistent[3] with it are eliminated.

If many examples are available, they can be used to further refine the identified set of modes, as described in Algorithm 2. At any moment, MILE maintains a set $\boldsymbol{M}$ of all consistent modes learned thus far for all concepts and predicates encountered in the processed examples. For each new incoming example, MILE applies Algorithm 1 to find all possible modes for its predicates, and then combines the outcome with $\boldsymbol{M}$

[2] Specifically, the *context* of a predicate in the body of an example is the head of that example.

[3] A mode declaration M_1 is inconsistent respect to a mode M_2 iff $M_1 \neq M_2$ (i.e., M_1 has input/output terms different from M_2). The internal representation of *Modes* in Algorithm 1 is a set $M=\{m_1,\ldots,m_n\}$ where each m_i is the list of modes for the literals in the body of an example. A mode m_p for a predicate p is inconsistent with respect to the set M, if there is a list of modes $m_i \in M$ such that m_i contains a mode $m_{p'}$ for the predicate p inconsistent with respect to m_p.

in order to exclude inconsistent ones. Modes in ***M*** are grouped by concept, and represented as couples of the form "*m(h,b)*" where *h* is a concept along with one possible mode for its arguments (e.g., p(+,–)) and *b* is the list of all corresponding modes for the other predicates.

Note that, the restriction of having a single mode for each context can be relaxed in the algorithm, by just dropping the consistency test. This is useful when a predicate is used with different meanings. However, such cases may give rise to issues that need further discussion (outside the scope of this paper). Consider for instance the following predicate for reversing a list:

```
reverse([3,1,2],[2,1,3]).
```

represented by the example:

```
rev(a,b):- decomp(a,3,c),decomp(c,1,d),decomp(d,2,e),nil(e),
           decomp(b,2,f),decomp(f,1,g),decomp(g,3,e).
```

where `a` represents the list `[3,1,2]` and `b` the list `[2,1,3]`. The following modes are obtained by applying the algorithm:

```
{rev(+,-),decomp(+,-,-),decomp(-,+,-),decomp(-,+,+),
  decomp(+,-,+),nil(+)}
```

The *strange* mode is `decomp(-,+,-)` expressing that it is possible to obtain the tail of a generic list with a given head. The problem is due to one predicate (`decomp/3`) being used both to *decompose* (meaning represented by the mode `decomp(+,-,-)`) and to *recompose* (expressed by the mode `decomp(-,+,+)`) a list. One solution would be to exploit different predicate names for different uses. However, a more clever solution is based on the consideration that this mode is obtained from literal `decomp(g,3,e)`, which is applicable if and only if the third argument is a constant `e` such that `nil(e)` is true (i.e., the empty list).

4 Examples

In the outlined mode declarations learning task, it is possible to encounter problems due to the ambiguity of a predicate mode. In some cases, this ambiguity can be resolved by subsequent observations (if any) provided to the algorithm. In other cases, the ambiguity cannot be avoided since it depends on the context in which the predicate is used. The presented algorithm is able to recognize such cases and to manage them correctly. This section shows, by means of some examples, the capabilities of the algorithm in such different situations.

The internal representation of ***M*** is a structure of the following form:

```
md([ [cm_1, [mp_11, mp_12, ... ] ],
     [cm_2, [mp_21, mp_22, ... ] ],
         ... ,
     [cm_n, [mp_n1, mp_n2, ... ] ] ).
```

where `md` represents ***M***, `cm`$_i$ represents a possible concept with one of its possible modes, and the `mp_ij`'s represent a possible list of modes for body predicates in the context of `cm_i`.

The first example shows as the incremental feature of the algorithm turns out to be important in some contexts.

Example 4. Given the example:

```
h(a):-p(a,b),p(a,c),t(a,b),t(a,c),l(c).
```

MILE induces the following list of modes for the literals `h/1`, `p/2`, `t/2` and `l/1`:

```
md([[h(+),[[l(+),t(+,+),p(+,-)],[l(+),t(+,-),p(+,+)]]]]).
```

According to the algorithm presented in Section 3, the argument `a` in the head leads to the following 1-layer literals: `p(a,b)`, `p(a,c)`, `t(a,b)`, `t(a,c)`. The first argument (`a`) of the predicates `p/2` and `t/2` is certainly an "input argument", but there is an indeterminism on the second argument, because both literals of `p/2` and literals of `t/2` produce simultaneously the same set of constants {`b,c`}. Since it is impossible to determine which of these two predicates actually gives as output the terms {`b,c`}, the system keeps both possibilities, hoping that a new incoming example resolves this ambiguity. Indeed, supposing that the next example provided to the system is

```
h(a) :- p(a,b),p(a,c),t(a,b),l(c).
```

MILE is able to induce the following list of modes for this example

```
md([[h(+),[[l(+),t(+,+),p(+,-)]]]])
```

that causes the following updated version of the global one (in which ambiguity is removed):

```
md([[h(+),[[l(+),t(+,+),p(+,-)]]]])
```

Hence, after examining just two examples, MILE learned the correct mode declarations for the predicates `h/1`, `l/1`, `t/2` and `p/2`. All new incoming examples must satisfy these declarations, or else an error will be notified. Indeed, the semantics that can be associated to these predicates is that `p/2` introduces new objects in the description, while `t/2` checks a relation between two known objects and, finally, `l/1` represents a property of the objects. Hence, supposing that the system is provided with the new example

```
h(a):-p(a,c),t(a,b),l(b).
```

it notes that the semantics (i.e., the mode declaration) of its predicates is different from that learned so far: *the predicate `t/2` now introduces objects in the description.* Then, the algorithm notify at the user that his descriptions associate different (or inconsistent) semantics to predicates. Note that this behavior can be avoided relaxing the consistency check.

The following example aims at showing a case in which the algorithm learns many mode declarations for one concept.

Example 5. Given the following example regarding the domain of family relationships

```
father(a,b) :- parent(a,b),male(a).
```

MILE is able to induce the following list of modes for the concept `father/2`:

```
md([ [ father(-,+), [ [male(+),parent(-,+)] ]],
     [ father(+,-), [ [male(+),parent(+,-)] ]],
     [ father(+,+), [ [male(+),parent(+,+)] ]]])
```

Due to a poor example description (only two literals in the body of the example), all the possible modes for the predicate `father/2` are correct (no inconsistency are derived). More in general, it is impossible to associate the correct semantics to the predicates `father/2` and `parent/2`. Supposing that MILE is provided with this second example, whose description is more detailed than the first one:

```
father(a,b):-parent(a,b),male(a),parent(b,c),female(c).
```

Now, the algorithm is able to understand that the only correct mode for the predicate `father/2` is `(+,-)`:

```
md([[father(+,-),[[female(+),male(+),parent(+,-)]]]])
```

Indeed, since `female(c)` is only used to test a property, the only predicate that can provide 'c' is `parent(b,c)`. Thus, `parent/2` must have mode `(+,-)`, and hence this leads to mode `(+,-)` for the predicate `father/2`.

Finally, the following example presents an important characteristic of the algorithm: its capability to learn many mode declarations for the same predicate when it is involved in different contexts.

Example 6. Supposing the following examples are provided to the algorithm:

```
father(a,b) :-
  parent(a,b),male(a),parent(b,c),female(c).
tree(t) :-
  node(t,a),node(t,b),node(t,c),parent(a,b),parent(a,c).
```

The user exploited the same predicate `parent/2` to describe both the family relationship concept `father/2` and the data structure concept `tree/1`. In this case, it is important to distinguish the two uses by linking the mode to its context. MILE induced the following list of modes:

```
md([[tree(+),      [ [node(+,-),parent(+,+)] ]],
     [father(+,-), [ [female(+),male(+),parent(+,-)]]]])
```

The algorithm distinguished different semantics for the same predicate: in the concept (context) `tree/1` the predicate `parent/2` is used to check a property of the objects (*nodes*) introduced by the predicate `node/2`; while in the concept (context) `father/2` the predicate `parent/2` gives the second argument as output, thus allowing to 'retrieve' children of a given person.

The algorithm MILE is integrated as preprocessor in a system for the incremental learning of first-order logic theories from examples, called INTHELEX, that is included in the architecture of the EU project COLLATE[4], in order to learn rules for automated classification and understanding of paper documents [17]. A deeper analysis about the best way to exploit/integrate the meta-knowledge of mode declaration in the revision phases of INTHELEX is currently ongoing, in order to assess the gain obtained in terms of computational complexity.

[4] IST-1999-20882 project COLLATE: Collaboratory for Annotation, Indexing and Retrieval of Digitized Historical Archive Material (URL: http://www.collate.de).

5 Conclusions

One of the biases used by Inductive Logic Programming (ILP) systems for reducing the space of candidate solutions in knowledge acquisition and concept formation is mode declaration. An algorithm to incrementally learn mode declarations for predicates from examples, called MILE has been presented. Since such information can be useful for restricting the search space of ILP systems, implementations of the proposed algorithm could be profitably used as a preprocessor of input examples in ILP systems in order to provide them with this meta knowledge. At present, the algorithm is implemented in Prolog language. Important features of MILE are its ability to manage and resolve (whenever possible) ambiguity, and to capture different semantics for each predicate in relation with its use in different contexts. We plan to extend the algorithm in order to deal with predicates used with different meanings in the same context.

References

1. Eric McCreath and Arun Sharma. Extraction of Meta-Knowledge to Restrict the Hypothes Space for ILP Systems. Eight Australian Joint Conference on Artificial Intelligence,pp.75--82, Xin Yao, 1995
2. D.H.D. Warren. Implementing Prolog – Compiling Predicate Logic Programs. Research Reports 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.
3. C.S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Dept. of Artificial Intelligence, University of Edinburgh, Aug. 1981.
4. C.S. Mellish. Some Global Optimizations for a Prolog Compiler. J. Logic Programming 2, 1 (Apr. 1985), pp.43-66.
5. U.S. Reddy. Transformation of Logic Programs into Functional Programs. In Proc. 1984 Int.Symposium on Logic Programming, IEEE Computer Society, Atlantic City, New Jersey, Feb. 1984, pp.187-196.
6. M. Bruynooghe, B. Demoen, A. Callebaut and G. Janssens. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In Proc. Fourth IEEE Symposium on Logic Programming, San Francisco, CA, Sep. 1987.
7. H. Manilla and E. Ukkonen. Flow Analysis of Prolog Programs. In Proc. Fourth IEEE Symposium on Logic Programming, San Francisco, CA, Sep. 1987.
8. S.K. Debray and D.S. Warren. Automatic Mode Inference for Logic Programs. Journal of Logic Programming, vol.5, n.3, pp.207-229, 1988.
9. K. Morik, S. Wrobel, J. Kietz, and W. Emde. Knowledge Acquisition and Machine Learning: Theory Methods and Applications. Academic Press, 1993.
10. S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo, pp.368-381. Ohmsa Publishers, 1990.
11. R.M. Cameron-Jones and J.R. Quinlan. Efficient top-down induction of logic programs. SIGART Bulletin, 5(1):33-42, 1994.
12. S. Muggleton, Inverse Entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming, 13 (3-4), Ohmsha, pp.245-286, 1995.

13. M. Furusawa, N. Inuzuka, H. Seki and H. Itoh. Bottom-up induction of logic programs with more than one recursive clause. In Proceedings of IJCAI97 workshop Frontiers of ILP, Nagoya, 1997.
14. P. Idestam-Almquist. Efficient induction of recursive definitions by structural analysis of saturations. In L. De Raedt (Ed.), Advances in Inductive Logic Programming, pp.192-205. IOS Press, 1996.
15. E. McCreath and A. Sharma. LIME: A System for Learning Relations. Algorithmic Learning Theory, pp.336-374, 1998.
16. C. Rouveirol. Extensions of Inversion of Resolution Applied to Theory Completion. Inductive Logic Programming, pp.64--90, S. Muggleton, Academic Press, 1992.
17. F. Esposito, S. Ferilli, N. Fanizzi, T.M.A. Basile and N. Di Mauro. Incremental Multistrategy Learning for Document Processing. Applied Artificial Intelligence Journal, 17:859-883, Taylor & Francis, London, 2003.