MULT-ICN: an empirical multiple predicate learner
Lionel MARTIN  Christel VRAIN 1,
L.I.F.O.
Rue Léonard de Vinci
B.P. 6759
45067 Orléans Cedex 2
France
email: martin@chambord.univ-orleans.fr  cv@chambord.univ-orleans.fr
tel: (33) 38 41 72 89
fax: (33) 38 41 71 37

Abstract
In this paper, we are interested in empirical multiple predicate learning. The first solution to this problem that consists in putting together the definitions obtained by a single predicate learning system is rarely interesting. We explain why and we show how a single predicate learning system has been extended to a multiple predicate learning system called MULT-ICN, which learns definite logic programs. Our system is based on the notion of extensional coverage but during the construction of the program, it builds a set of recursive dependencies that gives information about the mutually recursive calls. It has therefore two main advantages. First, it ensures that the learned program is globally consistent and complete, i.e., the learned program does not only extensionally cover the positive examples and reject the negative ones but it does prove that positive examples are true and negative ones are false in the semantics of the learned program. Secondly it uses only the knowledge given by the user to induce definitions and it is based on an acceptability rate; both enable to reduce the influence of the order the predicates are learned.

1 Introduction
Learning a logical definition of a single predicate from a set of positive and negative examples is a classical problem in Inductive Logic Programming (ILP) [14, 3, 9] referred to as single predicate learning (spla). In this paper, we consider the more general problem, called multiple predicate learning (mpl), that consists in finding a definition of a set of predicates from positive and negative examples. We show why putting together the definitions of each predicate learned by a spla system is rarely interesting. We present a mpl system, MULT-ICN, which extends to multiple

1 corresponding member of the Inference and Learning group, L.R.L., university of Paris South
predicate learning the ideas we have developed in [19]. The program learned by MULTICN always satisfies the properties of completeness and consistency.

In the context of spl, these ideas have also been extended to a three-valued framework, to deal with incomplete specifications of basic predicates and general logic programs [8]. In this paper, we consider only definite programs and their classical two-valued semantics.

The spl task can be formulated as follows. As inputs, we have a set BASE of basic predicates \( \{ p_1, \ldots, p_k \} \) and a predicate \( q \) to learn. The predicates of BASE are specified by a definite logic program, \( \text{BK} \), and the predicate to learn is specified by two sets of ground atoms, a set of positive instances \( E^+_q \) and a set of negative instances \( E^-_q \), that form the intended interpretation of \( q \). The spl of \( q \) consists in finding a definition for \( q \), i.e., a set \( C_q \) of clauses such that:

- the head of a clause is an atom built with the predicate \( q \),
- the body is a set of atoms built with some predicates belonging to \( \text{BASE} \cup \{ q \} \),
- \( C_q \) is locally complete, i.e., for each \( e^+ \) in \( E^+_q \), \( \text{BK} \cup C_q \models e^+ \),
- \( C_q \) is locally consistent, i.e., for each \( e^- \) in \( E^-_q \), \( \text{BK} \cup C_q \not\models e^- \).

Most systems in ILP are based on the notion of extensional coverage and extensional rejection w.r.t. an interpretation\(^2\) \( E \): let \( \mathcal{M}_{\text{BK}} \) be the semantics of the program \( \text{BK} \), i.e., the set of ground atoms true for \( \text{BK} \); an example \( e \) is extensionally covered by a clause if there exists a ground instance of this clause \( e \leftarrow l_1, \ldots, l_n \) where each \( l_i \) belongs to \( E^+_q \cup \mathcal{M}_{\text{BK}} \); an example \( e \) is extensionally rejected by a clause if for all ground instances of this clause \( e \leftarrow l_1, \ldots, l_n \), there exists an atom \( l_j \) in \( E^-_q \cup \mathcal{M}_{\text{BK}} \). Most spl systems based on the notion of extensional coverage [9, 14] work as follows: while there exists an uncovered positive example, find a clause which covers at least an uncovered positive example and rejects each negative one. Such systems have a well-known drawback when recursive clauses are learned: they have to prevent the creation of trivial clauses such as \( p(X) \leftarrow p(X) \), which cover all the positive examples and reject all the negative ones. This problem can be solved by learning only theories that are not recursive or by using biases using a well-founded ordering of the domain [14, 3, 1]: for example, in the case of an unary predicate \( q \), the restrictions induced by such an ordering can be stated as follows: if \( q(X) \leftarrow l_1, \ldots, l_n \) is the clause under construction, the literal \( q(Y) \) is added to the body of the clause only if there exists \( l_i = p(Z_1, \ldots, Z_k) \) with \( p \in \text{BASE} \), \( X = Z_0 \), \( Y = Z_j \), and a partial order of the domain, \( \leq \), such that for each ground instance \( p(c_1, \ldots, c_k) \in \mathcal{M}_{\text{BK}} \), \( c_j \leq c_i \). Such a bias generates strong restrictions, and in the worst case, it can prevent from finding a definition, even if there exists one. In [8], we avoid this problem by studying the recursive

\(^2\)The framework of TNC is more general since it allows partial definitions for basic predicates and it is based on a 3-valued logic.
dependencies between ground recursive calls. We present the notion of positive recursive dependencies in section 2 in the more general context of mpl.

In the mpl problem, we have still a set base of basic predicates \( \{p_1, \ldots, p_k\} \) defined by a definite logic program \( \text{bk} \) and we have a set \( \text{targ} \) of predicates to learn \( \{q_1, \ldots, q_m\} \). Each predicate \( q_i \) is specified by two sets of ground atoms \( E^+_q \) (the positive examples of \( q_i \)) and \( E^-_q \) (the negative examples of \( q_i \); if \( E^+_q \cup E^-_q \) is the set of all the ground atoms built with the predicate \( q_i \), we say that \( q_i \) is completely (or totally) specified (or defined), otherwise it is partially specified. We note \( E^+_\text{targ} = \bigcup_{q \in \text{targ}} E^+_q \) and \( E^-_\text{targ} = \bigcup_{q \in \text{targ}} E^-_q \). The mpl task has to produce a predicate definition \( C_q \) for each predicate \( q \) of \( \text{targ} \) such that:

\[
\begin{align*}
&\triangleright \text{the head of a clause of } C_q \text{ is an atom built with the predicate } q_i, \\
&\triangleright \text{the body of a clause is built with predicates from base or tarG,} \\
&\triangleright \{C_{q_1}, \ldots, C_{q_m}\} \text{ is globally complete, i.e., for each } e^+ \in E^+_q \cup \ldots \cup E^+_q, \\
&\text{bk} \cup C_{q_1} \cup \ldots \cup C_{q_m} \models e^+, \\
&\triangleright \{C_{q_1}, \ldots, C_{q_m}\} \text{ is globally consistent, i.e., for each } e^- \in E^-_q \cup \ldots \cup E^-_q, \\
&\text{bk} \cup C_{q_1} \cup \ldots \cup C_{q_m} \not\models e^-.
\end{align*}
\]

It is not possible to solve this mpl problem by putting together the definitions learned for each predicate \( q_i \) by a spl system for two reasons: if we try to learn successively a definition for each \( q_i \) with a spl process, it means that when learning \( q_i \), each predicate \( q_j, j \neq i \) has to be considered as a basic predicate; then the definite program \( \text{bk}_i \) used when learning \( q_i \) must be such that an atom \( e \) built with the predicate \( q_j, j \neq i \) is true for \( \text{bk}_i \) if \( e \in E^+_q \), false for \( \text{bk}_i \) if \( e \in E^-_q \), unknown for \( \text{bk}_i \) otherwise. Since \( \text{bk}_i \) is a definite program, it cannot support unknown information. On the other hand, if the predicates to learn are completely specified, the definitions obtained by successive spl satisfy, in the general case local but not global properties of completeness and consistency: let \( P_{q_i} = \{e \leftarrow \} \) \( \models e \in E^+_q \cup \ldots \cup E^+_q \); \( \models e^- \in E^-_q \cup \ldots \cup E^-_q \); \( \text{bk} \cup (\bigcup_{q \neq q_i} P_{q_i}) \cup C_{q_i} \not\models e^+ \), and for each \( e^- \) in \( E^-_q \cup \ldots \cup E^-_q \); \( \text{bk} \cup (\bigcup_{q \neq q_i} P_{q_i}) \cup C_{q_i} \not\models e^- \). This solution leads to a “global problem” [16], when some definitions are mutually recursive: the union of locally complete and consistent definitions does not generally give a set of globally consistent and complete definitions. For example, the two clauses \( \text{uncle}(X, Y) \leftarrow \text{nephew}(Y, X) \) and \( \text{nephew}(X, Y) \leftarrow \text{uncle}(Y, X) \) can be locally consistent and complete, but their union is globally useless. Again, biases have to be added to ensure global properties: first, if some definitions are already learned, when learning a predicate \( p \), we can prevent from using a predicate which depends on \( p \). With such a restriction, learning even and odd with the basic predicates zero and succ fails on...

---

A predicate \( q \) depends directly on a predicate \( p \) if there exists a clause having \( q \) in its head and \( p \) in its body; a predicate \( q \) depends on \( p \) if \( q \) depends directly on \( p \) or if \( q \) depends on a predicate which depends on \( p \).
the domain $[0...n]$, although there exists a globally satisfactory solution:

$$
\text{even}(0).
$$

$$
\text{even}(X) \leftarrow \text{succ}(Y, X), \text{odd}(Y).
$$

$$
\text{odd}(X) \leftarrow \text{succ}(Y, X), \text{even}(Y).
$$

An other bias [1] is based on a well-founded ordering of the elements of the domain. In this case, if the previous example is viewed in $\mathbb{Z}/6\mathbb{Z}$, this definition that is globally satisfactory cannot yet be learned.

When solving mpl problems by iterating spl techniques, an important problem pointed out in [16] is how the order the predicates are learned has an influence on the solution provided by the system. We present, in this paper, an uniform way to treat the drawbacks previously mentioned and which are due to recursive or mutually recursive definitions that lead to infinite loops. Moreover, our approach enables to reduce the consequences of the order the predicates are learned. It is based on the study of the ground recursive and mutually recursive calls between positive examples: the set of such calls is built during the construction of the program. This set and the way it is built is precisely defined in section 2. Section 3 presents the system MULT-ICN. We finish with a discussion about the problems linked to the order predicates are learned.

2 Positive recursive dependencies

Before presenting the notion of positive recursive dependencies, let us first recall a few points about definite logic programs and their semantics.

2.1 Definite programs

A definite program is a set of clauses $A \leftarrow A_1, \ldots, A_n$ where $A, A_1, \ldots, A_n$ are atoms; if $C$ is the clause $A \leftarrow A_1, \ldots, A_n$, $\text{head}(C)$ is the atom $A$ and $\text{body}(C)$ is the set of atoms $\{A_1, \ldots, A_n\}$.

If $P$ is a definite program, $\mathcal{H}_P$ denotes the Herbrand base of $P$, i.e., the set of ground atoms built with the predicates and constants appearing in $P$. We note $\text{Inst}_P$ the set of ground instances of clauses of $P$.

A subset $I$ of $\mathcal{H}_P$ is a model of $P$ iff for each ground instance $C$ of a clause of $P$ such that $\text{body}(C) \subseteq I$, then $\text{head}(C) \in I$. The semantics $M_P$ of $P$ is the least model of $P$, i.e., the intersection of each model of $P$; $M_P$ denotes the set of ground atoms true for $P$; the ground atoms false for $P$ are $\mathcal{H}_P - M_P$.

$M_P$ is usually defined as the least fixed point of the immediate consequence operator $T_P$ defined by:

$$
T_P(I) = \{ p \in \mathcal{H}_P \mid \text{there exists } C \in \text{Inst}_P \text{ with } p = \text{head}(C) \text{ and } \text{body}(C) \subseteq I \}
$$

4We consider in this paper, only definite programs in which appears no function symbols of arity $\geq 1$. 

2.2 Motivation and definition

Let us suppose that we want to learn the predicates even and odd on the set of integers $D = \{0, \ldots, 5\}$ and that we know the predicates zero and succ, i.e.,

$$\text{BK} = \{ \text{zero}(0) \leftarrow, \text{succ}(0,1) \leftarrow, \text{succ}(1,2) \leftarrow, \text{succ}(2,3) \leftarrow, \text{succ}(3,4) \leftarrow, \text{succ}(4,5) \leftarrow \}$$

The following program $P$ extensionally covers all the positive examples and rejects the negative ones.

$C_1 : \text{even}(X) \leftarrow \text{succ}(X,Y), \text{odd}(Y)$.

$C_2 : \text{odd}(X) \leftarrow \text{succ}(Y,X), \text{even}(Y)$.

But if, for instance, we call this program with the goal even(4), it successively calls the subgoal odd(5), even(4),... and therefore it generates a loop even(4) is not in the semantics of the previous program $P$. even(4) is false whereas it was expected to be true.

To deal with this problem we build the set of positive recursive dependencies that gives the links that exist between positive examples.

In our example, the whole set of positive recursive dependencies is:

$$\begin{align*}
\text{even}(4) & \leftarrow \text{odd}(5) & \text{odd}(5) & \leftarrow \text{even}(4) \\
\text{even}(2) & \leftarrow \text{odd}(3) & \text{odd}(3) & \leftarrow \text{even}(2) \\
\text{even}(0) & \leftarrow \text{odd}(1) & \text{odd}(1) & \leftarrow \text{even}(0)
\end{align*}$$

Figure 1

The positive recursive dependency even(4) $\leftarrow$ odd(5) comes from the ground instance of $P$, even(4) $\leftarrow$ succ(4,5), odd(5) which extensionally covers even(4). It means that to prove that even(4) is true, it is sufficient to prove that odd(5) is true.

The clause even(4) $\leftarrow$ succ(4,1), odd(1) is a ground instance of $P$ but it does not extensionally cover even(4) and therefore no recursive dependencies is built from it.

Construction: Let $P$ be a definite logic program defining predicates of TARG that extensionally covers all the positive examples and rejects the negative ones. We build the set $P_{rec}$ of positive recursive dependencies of $P$ w.r.t. $E_{TARG}$ as follows:

- for each ground instance of $P$, $L \leftarrow L_1, \ldots, L_n$ (where $L \in E^{+}_{TARG}$) which extensionally covers $L_n$, we add to $P_{rec}$ the clause $L \leftarrow L_n$,
- $L_{i_1}, \ldots, L_{i_k}$ are the atoms of $L_1, \ldots, L_n$ built with predicates of TARG.

Note: The clauses of $P_{rec}$ are ground and they are built only with predicates belonging to TARG.
2.3 Property

MULTICON builds a program \( \mathcal{P} \) such that each clause of \( \mathcal{P} \) extensionally covers some positive examples, and rejects the negative ones. Since such a program is not satisfying in the general case (as shown in the introduction), for each clause MULTICON tests if this clause is acceptable or not, i.e., if it allows to prove a sufficient number of positive examples. This test is realised by a computation of the semantics of the whole program \( \mathcal{P} \cup \text{bk} \) since the semantics of \( \mathcal{P} \) alone is empty (\( \mathcal{P} \) depends on predicates which are not defined in \( \mathcal{P} \)). To reduce the cost of the computation of the semantics of \( \mathcal{P} \cup \text{bk} \), we use the following property, which ensures that it is sufficient to compute the semantics of \( \mathcal{P} \cup \text{bk} \) if the semantics \( \mathcal{M}_{\text{bk}} \) of \( \text{bk} \) is then computed once for all at the beginning of the learning process and is used to build \( \mathcal{P} \).

\textbf{Theorem 1:} If \( E^+_{\text{TARG}} \subseteq \mathcal{M}_{\mathcal{P}_{\text{rec}}} \), then \( E^+_{\text{TARG}} \subseteq \mathcal{M}_{\mathcal{P} \cup \text{bk}} \).

\textbf{Theorem 2:} If each clause of the learned program extensionally rejects each negative example, then \( E^-_{\text{TARG}} \subseteq \mathcal{M}_{\mathcal{P} \cup \text{bk}} \).

These theorems are proved in annex; they give a way to ensure that the learned program is consistent and complete.

2.4 An acceptability criterion for a clause

Let us consider again the example of section 2.2 and let us suppose that the first clause built is \( C_1 \). The set of recursive dependencies is given in the first column of figure 1. The semantics of this set, restricted to the predicate even is empty and we could at first refuse this clause. But, if we examine more precisely the program that was given, both clauses \( C_1 \) and \( C_2 \) alone sound good, it is the conjunction of the two clauses that behaves incorrectly. The first clause could be accepted if afterwards we build “good” clauses that enable to prove \( \text{odd}(1), \text{odd}(3) \) and \( \text{odd}(5) \). A better idea is to consider that the positive examples that are not yet extensionally covered will be covered later and can be for the moment considered as true.

\textbf{Definition.} Let \( J \) be a set of ground atoms and \( \mathcal{P} \) a definite logic program, we note \( \mathcal{M}_{\mathcal{P}}(J) \) the semantics of \( \mathcal{P} \) w.r.t \( J \), i.e., the semantics of the program \( \mathcal{P} \cup \mathcal{P}_J \) where \( \mathcal{P}_J = \{ e \leftarrow . \mid e \in J \} \).

\footnote{\( \mathcal{P} \) generally depends on basic predicates which are defined in \( \text{bk} \), therefore its semantics is empty.}

\footnote{It is easy to show that \( \mathcal{M}_{\mathcal{P}}(J) \) is the least fixed point of the operator \( T_{\mathcal{P} \cup J} \) where \( T_{\mathcal{P} \cup J}(J) = T_{\mathcal{P}}(J \cup J) \). This operator gives a simple and efficient way to compute \( \mathcal{M}_{\mathcal{P}}(J) \).}
Therefore, the semantics of \( \{C_1\}_{rec} \) with respect to the interpretation \( \{odd(1), odd(3), odd(5)\} \) restricted to the predicate even is \( \{even(0), even(2), even(4)\} \). All the positive examples are true and the negative ones are false and therefore the clause can be accepted.

Let us now consider the second clause \( C_2 \). The set of recursive dependencies of the program is given in figure 1. All the positive examples are extensionally covered and we can no more consider that some uncovered positive examples will be later covered by “good” clauses. Since the semantics of \( P_{rec} \) is empty, \( C_2 \) is not acceptable, we must backtrack on \( C_2 \) and find an other clause. In this case, the clauses \( odd(X) \leftarrow succ(Y, X), zero(Y) \) and \( odd(X) \leftarrow succ(Y, X) \), \( succ(Z, Y), odd(Z) \) could be built.

In this example, a clause \( C \) is accepted only if all the positive examples covered by \( C \) are also true in the semantics of the learned program w.r.t. the set of uncovered positive examples. We introduce an acceptability rate in order to weaken this condition: a clause is accepted if a sufficient number of positive examples are true in the semantics of the learned program w.r.t. the set of uncovered positive examples.

**Definition:** Let \( \epsilon \) be an acceptability rate, \( 0 \leq \epsilon \leq 1 \). Let us call \( P \) the set of clauses that has already been built, \( C \) a new clause and \( P' = P \cup C \). Let \( Cov \) denotes the set of positive examples extensionally covered by \( P' \), and \( UnCov \) the set of positive examples which are not covered by \( P' \). The clause \( C \) is acceptable if \( \frac{\text{card}(\mathcal{M}_{P_{rec}}(UnCov) \cap Cov)}{\text{card}(Cov)} \geq \epsilon \).

We discuss, in section 4, the consequence of a low or an high acceptability rate. It is clear that, in the general case, a program composed of acceptable clauses (with an acceptability rate \( \epsilon < 1 \)) does not satisfy \( E_{TARG}^{+} \subseteq \mathcal{M}_{P_{rec}} \); such a program must then be completed with clauses (non recursive clauses for example) which ensure that the learned program is complete. However, an acceptability rate \( \epsilon = 1 \) can prevent from learning a complete program, as shown in [8]: it may exist a complete program \( P \) such that, whatever the order its clauses are learned, the first clause built is acceptable only for an acceptability rate \( \epsilon < 1 \).

3 **Presentation of MULTICN**

The system MULTICN takes as inputs a knowledge base on a set BASE of basic predicates and positive and negative examples of a set TARG of predicates, as specified in the introduction.

The predicate definitions learned by MULTICN are function-free general logic programs, i.e., sets of clauses: \( A \leftarrow L_1, \ldots, L_n \) where:

\[ \text{It behaves as if we momentarily add to the set of recursive dependencies the 3 clauses: } odd(1) \leftarrow odd(3) \leftarrow \text{ and } odd(5) \leftarrow. \]
no function symbol appears,
• A is an atom built with one of the target predicate,
• \( L_i \) is an atom or an expression \( X = Y, X \neq Y \) where \( X \) and \( Y \) are variables
  previously introduced in the clause.

3.1 Algorithm

We give in figure 2 the algorithm of MULT_ICN. We call \( \text{NEG} \) the set of negative
examples and \( \text{POS} \) the set of positive examples that have not yet been extensionally
covered. First, (step 1.1), it chooses the predicate \( Q \), among the predicates of \( \text{TARG} \)
that have still uncovered positive examples, that will compose the head \( Q(X_1, \ldots , X_n) \leftarrow \).
of the clause (this point is detailed in section 3.2). Then (step 1.2) it
refines the clause by iteratively adding the most promising literal to the body of
the clause until all the negative examples of the predicate \( Q \) are rejected. The gain
of a literal is computed from the number of ground instances of the clause which
extensionally cover (resp. reject) positive examples (resp. negative examples)
with and without the new literal; literals with a good gain are memorized for an
eventual backtrack. It evaluates the clause according to the acceptability criterion
given in section 2.4; if the clause that has been built is not acceptable then the
system backtracks to find a new one.

1. **While** \( \text{POS} \neq \emptyset \) **do**
   1.1 Choose a predicate to learn
   1.2 Create a new acceptable clause which extensionally covers some
       elements of \( \text{POS} \) and rejects each element of \( \text{NEG} \)
   1.3 Remove the covered examples from \( \text{POS} \).
2. Evaluate \( M_{\text{POS}} \) (\( P \) is the learned program)
3. **While** \( E_{\text{TARG}}^+ - M_{\text{POS}} \neq \emptyset \) **do**
   3.1 Choose a predicate to learn
   3.2 Create a new acceptable clause containing no target predicate
       in the body which extensionally covers some elements of
       \( E_{\text{TARG}}^+ - M_{\text{POS}} \)
       and rejects each element of \( \text{NEG} \)
   3.3 Evaluate \( M_{\text{POS}} \)

Figure 2

When all the positive examples have been extensionally covered, the system
computes (step 2) the semantics of the set of recursive dependencies. If it remains
some positive examples that are covered but not proved the system attempts to
find some new clauses that cover them and that uses only knowledge base in order
to prevent from loops (step 3).

In this last step we can build a clause that covers a lot of examples that were
covered by a clause previously built. We could improve MULT_ICN by pruning the
set of clauses. In the same way, we have noticed that sometimes when building a
3.2 Choice of the predicate to learn

An important point in this algorithm is the choice of the predicate to learn. We will discuss in section 4 how it has an effect on learning.

In MULTICN, the choice can either be automatic or it can be left to the user.

In the former case, the underlying idea is to learn, at first, predicates that precede\(^8\) other ones. At the beginning, the system has no information and therefore chooses at random the first predicate to learn. Let us call it \(Q_1\). After the construction of the first clause, if a predicate \(Q_2\) appears in the body of the clause, then the system focusses on \(Q_2\) and learns a clause for \(Q_2\) and so on. We search down the precedence graph between the predicates of TARG (this graph is built from the learned definitions) so as to reach, if possible, a predicate that depends only on the knowledge base. When the system has not enough information to decide, as for instance when a cycle appears, it chooses at random a predicate that has already been chosen but is not completely learned yet (uncovered positive examples remain) otherwise, it chooses at random a new predicate to learn. The advantages of this way of choosing the predicates to learn are discussed in section 4.

3.3 Practical computation of \(M_{\mathcal{P}_{\text{rec}}}\)

First, let us note that when a clause \(C\) is added to the program \(\mathcal{P}\), the set of recursive dependencies of \(\mathcal{P} \cup C\) is \(\mathcal{P}_{\text{rec}} \cup \{C\}_{\text{rec}}\).

The semantics of a definite program \(\mathcal{P}\) is computed with the immediate consequence operator \(T_{\mathcal{P}}\) defined in section 2.1. Since \(T_{\mathcal{P} \cup C}(I) = T_{\mathcal{P}}(I) \cup T_{\mathcal{P}}(I)\), when a clause \(C\) is added to a program \(\mathcal{P}\), the computation of \(M_{\mathcal{P}_{\text{rec}}}\) can be simplified by using information about \(M_{\mathcal{P}}\). In our framework, when a learned clause \(C\) is built without predicates of TARG, then \(\{C\}_{\text{rec}}\) is a set of ground facts \(e \leftarrow \cdot\) for each example \(e\) covered by \(C\); as soon as \(C\) is added to the learned program \(\mathcal{P}\), the semantics of \(\mathcal{P} \cup C \cup \text{BK}\) contains each atom \(e\) such that \(e \leftarrow \cdot\) belongs to \(\{C\}_{\text{rec}}\). Then, we can perform the following transformations on the set of recursive dependencies \(\mathcal{P}_{\text{rec}}\).

Let \(E\) be a set of ground atoms, initially \(E = \{e \mid e \leftarrow \cdot, \in \{C\}_{\text{rec}}\}\).
- remove from \(\mathcal{P}_{\text{rec}}\) each clause \(c\) such that \(\text{body}(c) \neq \emptyset\) and \(\text{head}(c) \in E\),
- delete from the body of the clause of \(\mathcal{P}_{\text{rec}}\) atoms belonging to \(E\),
- for each new clause \(e \leftarrow \cdot\) of \(\mathcal{P}_{\text{rec}}\), add \(e\) to \(E\).

until no more atoms are added to \(E\).

---

\(^8\)a predicate \(p\) precedes a predicate \(q\) if \(q\) depends on \(p\)
Example: Consider the example of section 2.2 and assume that the first built clause is: \( \text{odd}(X) \leftarrow \text{succ}(Y, X), \text{even}(Y) \); its set of recursive dependencies is:
\[
\{ \text{odd}(1) \leftarrow \text{even}(0), \ \text{odd}(3) \leftarrow \text{even}(2), \ \text{odd}(5) \leftarrow \text{even}(4) \}
\]
If the clause \( \text{even}(X) \leftarrow \text{zero}(X) \) is added, then the new set of recursive dependencies is:
\[
\{ \text{even}(0) \leftarrow, \ \text{odd}(1) \leftarrow, \ \text{odd}(3) \leftarrow \text{even}(2), \ \text{odd}(5) \leftarrow \text{even}(4) \}
\]

3.4 An example session

We have run \textsc{MultInc} on an example given in [16], where the knowledge base \( \mathcal{B} \) is defined by the following set of ground facts:

\[
\begin{align*}
\text{male}(\text{prudent}) &\quad & \text{female}(\text{laurn}) &\quad & \text{father}(\text{bar}, \text{stijn}) &\quad & \text{mother}(\text{kateen}, \text{stijn}) \\
\text{male}(\text{willem}) &\quad & \text{female}(\text{ester}) &\quad & \text{father}(\text{bar}, \text{stijn}) &\quad & \text{mother}(\text{kateen}, \text{pieter}) \\
\text{male}(\text{etienne}) &\quad & \text{female}(\text{rose}) &\quad & \text{father}(\text{luc}, \text{kateen}) &\quad & \text{mother}(\text{laurn}, \text{etienne}) \\
\text{male}(\text{rew}) &\quad & \text{female}(\text{alice}) &\quad & \text{father}(\text{willem}, \text{kateen}) &\quad & \text{mother}(\text{etienne}, \text{kateen}) \\
\text{male}(\text{burt}) &\quad & \text{female}(\text{gromme}) &\quad & \text{father}(\text{rew}, \text{willem}) &\quad & \text{mother}(\text{gromme}, \text{willem}) \\
\text{male}(\text{luc}) &\quad & \text{female}(\text{kateen}) &\quad & \text{father}(\text{burt}, \text{lucy}) &\quad & \text{mother}(\text{gromme}, \text{lucy}) \\
\text{male}(\text{pieter}) &\quad & \text{female}(\text{lieve}) &\quad & \text{father}(\text{leon}, \text{rose}) &\quad & \text{mother}(\text{alice}, \text{rose}) \\
\text{male}(\text{stijn}) &\quad & \text{female}(\text{am}) &\quad & \text{father}(\text{etienne}, \text{burt}) &\quad & \text{mother}(\text{rose}, \text{lucy}) \\
\text{female}(\text{lucy}) &\quad & \text{female}(\text{lucy}) &\quad & \text{father}(\text{etienne}, \text{am}) &\quad & \text{mother}(\text{rose}, \text{am})
\end{align*}
\]

\textsc{MultInc} learns the following program:

\[
\begin{align*}
\text{female.ancestor}(X, Y) &\leftarrow \text{mother}(X, Y). \\
\text{female.ancestor}(X, Y) &\leftarrow \text{male.ancestor}(Z, Y), \text{female.ancestor}(X, Z). \\
\text{female.ancestor}(X, Y) &\leftarrow \text{male.ancestor}(Z, Y), \text{mother}(X, T), \text{female.ancestor}(T, Y). \\
\text{male.ancestor}(X, Y) &\leftarrow \text{father}(X, Y). \\
\text{male.ancestor}(X, Y) &\leftarrow \text{female.ancestor}(Z, Y), \text{male.ancestor}(X, Z). \\
\text{male.ancestor}(X, Y) &\leftarrow \text{father}(X, Z), \text{father}(Z, Y).
\end{align*}
\]

4 Discussion

4.1 Choice of the predicate to learn

Let us recall that, when it is possible, \textsc{MultInc} tries to find a definition for a predicate which is the deepest leaf of the precedence graph between the target predicates, induced by the previously learned definitions.

This way of choosing the predicates to learn has some advantages:

- We can reach a predicate that depends only on knowledge base and therefore computing the actual semantics of the program defining this predicate is easy and it will be easier to compute the semantics of predicates that depend on it (the set \( P_{\text{rec}} \) will be simplified as shown in section 3.3).
- When a predicate is chosen, we learn all the predicates that are strongly connected to it. If we succeed we can then forget the whole set of recursive dependencies between these predicates and focus on a new predicate that has not yet been learned.
- When all the positive examples have been covered, we test whether it remains some positive examples that are not proved. In this case, for the time being, we try to cover them with new clauses using only background knowledge. But we could extend the definition by allowing predicates that are not linked to this predicate.

4.2 Influence of the order of learning the predicates

We discuss, in this section, the influence of the order the predicates are learned. When predicates are only partially defined, a first problem can appear in systems that compute, directly or indirectly, the semantics of the program obtained with the new learned clause and use these new facts to learn other definitions [16]. In our framework, **MULTICN** does not take into account learned information and keeps always the same knowledge base for learning new definitions, therefore this problem does not occur.

The second problem can be stated as follows: the choice of a clause for a predicate \( p_1 \) can prevent from building a clause for a predicate \( p_2 \). Let us illustrate this point by the following example in which we have six persons \( yves, pierre, paul, jean, yann \) and \( leon \), 3 basic predicates defined as follows:

\[
\text{BK} = \{ \text{hate(\text{jean}, \text{pierre})} \iff \text{hate(\text{leons}, \text{yann})} \}, \\
\cup \{ \text{indebted(\text{pierre}, \text{paul})} \iff \text{indebted(\text{pierre}, \text{jean})} \}
\cup \{ \text{indebted(\text{yves}, \text{jean})} \iff \text{indebted(\text{yann}, \text{leons})} \}
\cup \{ \text{melancholic(\text{pierre})} \iff \}
\]

and two predicates to learn, \( \text{happy} \) and \( \text{sad} \) defined by \( \text{sad(\text{pierre})}, \text{sad(\text{yves})} \) and \( \text{happy(\text{jean})} \) are true and all the other facts about \( \text{happy} \) and \( \text{sad} \) are false.

Let us assume that we start by learning a clause for the predicate \( \text{sad} \), and suppose that the clause is \( C_1: \text{sad}(X) \iff \text{indebted}(X,Y), \text{happy}(Y) \). It means that to prove that \( \text{sad(\text{pierre})} \) is true, we must prove \( \text{happy(\text{jean})} \). We must then learn \( \text{happy} \), the only possible clause is \( C_2: \text{happy}(X) \iff \text{hate}(X,Y), \text{sad}(Y) \). In this clause, we must prove that \( \text{pierre} \) is sad to prove that \( \text{jean} \) is happy. Then, when adding \( C_2 \) to \( C_1 \), although all the positive examples are covered, no one is proved, and then the clause is accepted by **MULTICN**, only if the acceptability rate \( \epsilon \) is equal to 0. If \( \epsilon = 0 \), since no example is proved, **MULTICN** searches for a clause built with predicates different from \( \text{sad} \) and \( \text{happy} \) which covers either \( \text{happy(\text{jean})} \) or \( \text{sad(\text{pierre})} \) or \( \text{sad(\text{yves})} \) and rejects the negative examples; there is no possible clause which covers \( \text{happy(\text{jean})} \), but \( \text{sad(\text{pierre})} \) is covered by the clause \( C_3: \text{sad}(X) \iff \text{melancholic}(X) \). Finally, \( C_1 \cup C_2 \cup C_3 \) forms an acceptable program.

This example shows that when the acceptability rate is high, the learning task
is sensitive to the order the predicates are learned: in this case, it is not possible
to learn a definition for happy and then the learning system fails. This drawback
disappears when the acceptability rate is low. Therefore,
- on one hand, the more this rate is low, the less the mpl task is sensitive to
the order the clauses are built. When this rate is equal to 0, any clause which
extensionally covers a positive example and rejects all the negative ones is accep-
table, and when every positive example is covered, multi-icn tries to complete the
definition of predicates for which it remains positive examples that are not proved.
- on the other hand, if we add clauses for which the number of covered but
not proved examples is high, it will probably be necessary to build new definitions
in order to cover some examples that were previously covered, and some previous
clauses may become redundant. This is the case in the previous example: if we
add to the knowledge base the fact melancholic(yves), then C3 covers the two
positive examples of sad and C1 is useless.

5 Conclusion

This paper addresses the problem of multiple predicate learning, introduced in [16].
multi-icn has been implemented in Sicstus Prolog on a Sun. It has been runned on
the examples given in this paper and has given the expected results. Biases have
been introduced in multi-icn to reduce the search space. As already mentioned,
multi-icn uses only the knowledge given by the user to learn. This point and the
study of the dependency set enable to ensure that the learned program will prove
that the positive examples are true and that the negative ones are false. But, if
the specification of the predicates are too partial, then it can be difficult to find a
consistent and complete definition.

We are now extending multi-icn in two directions, as done in icn [8]: on one
hand we allow the use of the negation in the body of a clause, by replacing the
set of recursive and mutually recursive dependencies between positive examples,
by the set of recursive and mutually recursive dependencies between positive and
negative examples, and extending the notion of acceptable clause; on the other
hand, we allow partial definitions for basic predicates. Both extension are studied
in the case of 3-valued semantics for general logic programs.

This work is currently applied to revise deductive databases [18].

References

Proc. of the Fourth Int. Workshop on Inductive Learning Programming
(IIP-94), pp. 11-29.


A Proof of theorem 1

Theorem 1. If $E_{\text{TARG}} \subseteq M_{\text{rec}}$ then $E_{\text{TARG}} \subseteq M_{\text{UBK}}$.

Let us recall that $T_p(I) = \{ p \in \mathcal{H}_p \mid \text{there exists } C \in \text{Inst}_p \text{ with } p = \text{head}(C) \text{ and } \text{body}(C) \subseteq I \}$.

and let us define:

$$
T_p^0 = \emptyset \quad T_p^0 = \emptyset
$$

Before proving theorem 1, we must prove the two following lemmas.

Lemma 1. For all $i$, $T_{\text{BK}}^i \subseteq T_{\text{UBK}}^i$

Proof: by induction

The case $i = 0$ is obvious.

If the result of lemma 1 holds for $i$, let $a \in T_{\text{BK}}^{i+1}$, there exists a ground clause $C$ of $\text{Inst}_{\text{BK}}$ such that $\text{head}(C) = a$ and $\text{body}(C) \subseteq T_{\text{BK}}^i$. $C$ belongs to $\text{Inst}_{\text{UBK}}$ and by hypothesis of induction, $\text{body}(C) \subseteq T_{\text{UBK}}^i$. Therefore, $a \in T_{\text{UBK}}^{i+1}$.

Lemma 2. Let $\alpha$ be the smallest integer such that $T_{\text{BK}}^\alpha = T_{\text{BK}}^{\alpha+1}$ (i.e., $M_{\text{BK}} = T_{\text{BK}}^\alpha$). Then,$$
For all n , E_{\text{TARG}} \cap T_{\text{rec}}^n \subseteq T_{\text{UBK}}^{n+\alpha}.$$

Remark: The integer α given in lemma 2 exists, since the set of ground atoms is finite.

Proof: by induction
- n = 0: obvious
- Let us suppose that the result holds for n. Let \( e \in E^+_{\text{TARG}} \cap T^n_{\text{rec}} \). There exists a clause \( C_{\text{rec}} \) of \( P_{\text{rec}} \) with \( e = \text{head}(C_{\text{rec}}) \) and \( \text{body}(C_{\text{rec}}) \subseteq T^n_{\text{rec}} \).

\[ C_{\text{rec}} \text{ comes from a ground instance } C \text{ of a clause of } P. \text{ Since } e \in E^+_{\text{TARG}}, C \text{ extensionally covers } e \text{ and therefore for all } l_i \in \text{body}(C), l_i \in E^+_{\text{TARG}} \cup M_{\text{BK}}. \]

\[ \begin{itemize}
  \item If \( l_i \) is built with a predicate of \( \text{TARG} \), \( l_i \in E^+_{\text{TARG}} \). Moreover, \( l_i \) appears in \( C_{\text{rec}} \) (construction of \( P_{\text{rec}} \)). Therefore, \( l_i \in E^+_{\text{TARG}} \cap T^n_{\text{rec}} \), and by hypothesis of induction, \( l_i \in T^{n+\alpha}_{\text{rec}} \).
  \item If \( l_i \) is built with a predicate of \( \text{BASE} \), \( l_i \in M_{\text{BK}} \) (2), i.e., \( l_i \in T^0_{\text{BK}} \).
  Therefore, \( l_i \in T^{n+\alpha}_{\text{rec}} \) (lemma 2), and \( l_i \in T^{n+\alpha}_{\text{rec}} \) (\( T \) is monotonous).
\end{itemize} \]

For all \( l_i \in \text{body}(C) \), \( l_i \in T^{n+\alpha}_{\text{rec}} \) and therefore \( e \in T^{n+\alpha+1}_{\text{rec}} \).

Proof of theorem 1: When such a result holds, let us consider \( \beta \) the smallest integer satisfying \( T^\beta_{\text{rec}} = T^{\beta+1}_{\text{rec}} \), i.e., \( M_{\text{rec}} = T^\beta_{\text{rec}} \).

If \( E^+_{\text{TARG}} \subseteq M_{\text{rec}} \) then on one hand, \( E^+_{\text{TARG}} \cap M_{\text{rec}} = E^+_{\text{TARG}} \) and on the other hand, \( E^+_{\text{TARG}} \cap M_{\text{rec}} \subseteq E^+_{\text{TARG}} \cap T^\beta_{\text{rec}} \). Consequently, \( E^+_{\text{TARG}} \subseteq E^+_{\text{TARG}} \cap T^\beta_{\text{rec}} \subseteq M_{\text{rec}} \).

B Proof of theorem 2

Theorem 2. If each clause of the learned program extensionally rejects each negative example, then \( E^{-}_{\text{TARG}} \subseteq M_{\text{rec}} \).

Before proving this result, we prove the two following lemmas, lemma 3 is used in the proof of lemma 4.

Lemma 3. For all \( i \), \( T^i_{\text{rec}} \cap H_{\text{BK}} \subseteq M_{\text{BK}} \).

Proof: by induction
- Case \( i = 0 \): obvious.
- Let us suppose that the result holds for \( i \). If there exists \( l_i \in T^{i+1}_{\text{rec}} \cap H_{\text{BK}} \cap M_{\text{BK}} \), then there exists a ground instance of a clause of \( \text{BK} \) (no clause of \( P \) can be unified with \( l_i \)) with \( \text{head}(C) = l_i \) and \( \text{body}(C) \subseteq T^i_{\text{rec}} \). By hypothesis of induction, \( \text{body}(C) \subseteq M_{\text{BK}} \) (all the clauses of \( \text{BK} \) are built with predicates of \( \text{BASE} \)) and therefore \( l_i \in M_{\text{BK}} \), which contradicts \( l_i \in M_{\text{rec}} \).
Lemma 4. For all $i$, $E_{\text{TARG}}^- \cap T_{\text{PUBK}}^i = \emptyset$.

Proof: by induction
- $i = 0$: obvious
- Let us suppose that the result holds for $i$, i.e., $E_{\text{TARG}}^- \cap T_{\text{PUBK}}^i = \emptyset$. (3)
  If there exists $e \in E_{\text{TARG}}^- \cap T_{\text{PUBK}}^{i+1}$, then there exists a ground instance $C$ of a clause of $P$ (no clause of BK can be unified with $e$) such that head$(C) = e$ and body$(C) \subseteq T_{\text{PUBK}}^i$.
  But, $e \in E_{\text{TARG}}^-$, therefore this clause extensionally rejects $e$ and there exists a literal $l_i$ of body$(C)$ with $l_i \in E_{\text{TARG}}^- \cup \overline{M_{\text{BK}}}$.
  - If $l_i$ belongs to $E_{\text{TARG}}^-$, then $l_i$ belongs to $E_{\text{TARG}}^- \cap T_{\text{PUBK}}^i$, which contradicts the hypothesis (3).
  - If $l_i$ belongs to $\overline{M_{\text{BK}}}$ then $l_i \in T_{\text{PUBK}}^i$ (lemma 3) which contradicts the hypothesis (3).

Proof of theorem 2: Let us call $\gamma$ the least integer such that $T_{\text{PUBK}}^\gamma = T_{\text{PUBK}}^{\gamma+1}$, i.e.,
$T_{\text{PUBK}}^\gamma = M_{\text{PUBK}}$.
Lemma 4 applied to $\gamma$ gives $E_{\text{TARG}}^- \cap T_{\text{PUBK}}^\gamma = \emptyset$, i.e., $E_{\text{TARG}}^- \subseteq M_{\text{PUBK}}$. 