Integrating Induction and Abduction in Logic Programming

Evelina Lamma [a], Paola Mello [b], Michela Milano [a] and Fabrizio Riguzzi [a]

[a] *DEIS, Università di Bologna Viale Risorgimento 2, 40136 Bologna, Italy*
{elamma,mmilano,friguzzi}@deis.unibo.it

[b] *Dip. di Ingegneria, Università di Ferrara Via Saragat 1, 41100 Ferrara, Italy*
pmello@ing.unife.it

We propose an approach for the integration of abduction and induction in Logic Programming. We define an Abductive Learning Problem as an extended Inductive Logic Programming problem where both the background and target theories are abductive theories and where abductive derivability is used as the coverage relation instead of deductive derivability. The two main benefits of this integration are the possibility of learning in presence of incomplete knowledge and the increased expressive power of the background and target theories. We present the system LAP (Learning Abductive Programs) that is able to solve this extended learning problem and we describe, by means of examples, four different learning tasks that can be performed by the system: learning from incomplete knowledge, learning rules with exceptions, learning from integrity constraints and learning recursive predicates.

Key words: Abductive Logic Programming. Inductive Logic Programming. Machine Learning. Nonmonotonic Reasoning.

1 Introduction

Both abduction and induction have been recognized as powerful mechanisms for hypothetical reasoning in presence of incomplete knowledge [9,20,25,31,32]. Abduction is generally understood as reasoning from effects to causes or explanations. Given a theory T and a formula G, the goal of abduction is to find a (possibly minimal) set of atoms Δ which, together with T, entails G. Induction is generally understood as inferring general rules from specific data. Given a theory T and a formula (observation) G, the goal of induction is to find a set of general rules Δ (of the type $\alpha \rightarrow \beta$) which, together with T, entails G.

The relationship between abduction and induction has been studied recently by several authors, mainly in the field of Artificial Intelligence (see, for instance, [8,34]). In general, the question of how abduction and induction could be integrated and how they would cooperate, complement and affect each other is emerging as an important problem and has been the subject of two international workshops (see [15,22]).

Our work is an attempt to bring closer the fields of abductive and inductive reasoning in the Logic Programming setting, as done also in [1,4,17]. An Inductive Logic Programming (ILP, for short) problem can be defined as follows [5]: given a set $\mathcal{P}$ of possible programs, sets E^+ and E^- of positive and negative examples and a consistent logic program B (*background knowledge*), find a logic program $P \in \mathcal{P}$ (*target program*) such that $B \cup P$ entails (or *covers*) each $e^+ \in E^+$ and does not entail any $e^- \in E^-$. In Abductive Logic Programming (ALP, for short) ([20,23]) a program is a triple $\langle P, A, IC \rangle$ where P is a logic program possibly with abducible atoms in clause bodies, A is a set of *abducible predicates* and IC is a set of *integrity constraints* (denials, for simplicity). Abducibles are predicates which can be assumed true (or false) during the computation provided that they are consistent with integrity constraints. Therefore, they can be used to deal with incomplete information.

Our approach for the integration of abduction and induction consists in defining a new learning problem, called Abductive Learning Problem, where both the background and target programs are abductive logic programs and abductive derivability is used as the coverage relation in substitution of deductive derivability. The main advantages of the integration concern the possibility of learning in presence of incomplete knowledge and the increased expressive power of learned programs.

We propose the system LAP (Learning Abductive Programs) that is able to solve the new learning problem. The algorithm extends the basic top-down algorithm adopted in ILP [5] by substituting the deductive proof procedure of logic programming with the abductive proof procedure defined in [26] for testing the coverage of examples. The algorithm is sound but incomplete with respect to the new problem definition. We identify and discuss four different tasks that can be accomplished by such a learning system: learning from incomplete data, learning rules with exceptions, learning from integrity constraints and learning recursive predicates.

The paper is organized as follows: in section 2, we recall the main concepts of ALP and ILP. In section 3, we present the algorithm, and we discuss its properties of soundness and completeness in section 4. In section 5, we illustrate the above mentioned tasks by means of examples. Related works are presented in section 6. Conclusions and future works follow in section 7.

2 Abductive and Inductive Logic Programming

In this section, we first briefly recall some notations of Logic Programming and the basic concepts of Abductive Logic Programming and Inductive Logic Programming. Then we define the Abductive Learning Problem.

2.1 Preliminaries on Logic Programming

Let us first set up some basic notations and terminologies we will adopt throughout the paper. We shall use the basic concepts of logic programming (e.g. as in [3]). We consider normal logic programs, where negation is denoted by *not*, and can occur in clause bodies.

A *normal logic program* is a set of rules of the form:

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_{m+n}.$$

where $m, n \geq 0$ and each A_i $(i = 0, \ldots, m+n)$ is an atom.

2.2 Abductive Logic Programming

Abduction plays an important role in everyday human problem solving, and it has been recognized as a powerful mechanism for hypothetical reasoning in presence of incomplete knowledge. In Artificial Intelligence, abduction has been applied successfully to a number of fields [24]: fault diagnosis, high level vision, natural language understanding, planning, knowledge assimilation, default reasoning.

In a logic programming setting, abduction is modelled through *abductive theories* or *abductive logic programs* [24]. An abductive theory is a triple $\langle P, A, IC\rangle$ where:

- P is a definite logic program;
- A is a set of *abducible predicates* (or *abducibles*);
- IC is a set of *integrity constraints* in the form of denials, i.e.:
  $\leftarrow A_1, \ldots, A_m.$

Abducibles represent the predicates about which we can make assumptions because they are incompletely specified. They "carry" all the incompleteness of the domain. We indicate with $\mathcal{L}^{\mathcal{A}}$ the set of all atoms built over the predicates of A and we call them *abducible atoms*.

Given an abductive logic program $T = \langle P, A, IC \rangle$ and a formula G, the goal of abduction is to find a (possibly minimal) set of ground atoms Δ (*abductive explanation*) of predicates in A such that $P \cup \Delta \models G$. It is also required that the program $P \cup \Delta$ is consistent with respect to IC, i.e., $P \cup \Delta \models IC$.

Example 1 *Let us consider the following abductive logic program T, inspired to [35], where P is:*

$$\begin{array}{l} shoes\_are\_wet \leftarrow grass\_is\_wet. \\ grass\_is\_wet \leftarrow rained\_last\_night. \\ grass\_is\_wet \leftarrow sprinkler\_was\_on. \\ electrical\_black\_out. \end{array}$$

The integrity constraints IC are:

$$\leftarrow electrical\_black\_out, sprinkler\_was\_on.$$

(stating that electrical_black_out and sprinkler_was_on cannot be both true in a consistent model) and let predicates rained_last_night and sprinkler_was_on be abducible. The observation shoes_are_wet has the abductive explanation:

$$\Delta_1 = \{rained\_last\_night\}.$$

The set $\Delta_2 = \{sprinkler\_was\_on\}$, even if it explains the observation, is not considered as a valid abductive explanation because it violates the constraint.

ALP considers as well programs $T = \langle P, A, IC \rangle$ where P is a normal logic program and constraints in IC contain negative literals. In this case, abduction is used also for modelling negation, obtaining a type of negation called *negation by default* [6]. Negation is modelled by transforming the program T into its *positive version* $T^* = \langle P^*, A \cup A^*, IC \cup IC^* \rangle$. T^* does not contain negative literals and is obtained in this way: for each predicate symbol p, a new predicate symbol not_p is included in A^* and the integrity constraint:

$$\leftarrow p(\mathbf{\overline{x}}), not\_p(\mathbf{\overline{x}})$$

is included in IC^*, where $\mathbf{\overline{x}}$ is a tuple of variables. We define the *complement* $\bar{l}$ of a literal l as

$$\bar{l} = \begin{cases} not\_p(\mathbf{\overline{x}}) & \text{if } l = p(\mathbf{\overline{x}}) \\ p(\mathbf{\overline{x}}) & \text{if } l = not\_p(\mathbf{\overline{x}}) \end{cases}$$

We indicate with $\mathcal{L}^{\mathcal{D}}$ the set of all atoms built over the predicates of A^* and we call them *default atoms*. In the following we will always consider the positive version of programs.

As a model-theoretic semantics for ALP, we adopt the *abductive model* semantics defined in [6]. We do not want to enter into the details of the definition, we will just give the following proposition which will be useful in the following.

Proposition 2 *Given an abductive model M for the abductive program $AT = \langle P, A, IC \rangle$, there exists a set of atoms $H \subseteq (\mathcal{L}^{\mathcal{A}} \cup \mathcal{L}^{\mathcal{D}})$ such that M is the least Herbrand model of $P \cup H$.*

PROOF. Straightforward from the definition of abductive model (definition 5.7 in [6]. $\square$

In [26] a proof procedure for the positive version of abductive logic programs has been defined. This procedure (reported in the Appendix) starts from a goal and a set of initial assumptions Δ_i and results in a set of consistent hypotheses (abduced literals) Δ_o such that $\Delta_o \supseteq \Delta_i$ and Δ_o, together with the program P, allows to derive the goal. The proof procedure uses the notion of *abductive* and *consistency derivations*. Intuitively, an abductive derivation is the standard Logic Programming derivation suitably extended in order to consider abducibles. As soon as an abducible atom δ is encountered, it is added to the current set of hypotheses, and it must be proved that every integrity constraint containing δ is satisfied. To this purpose, a consistency derivation for δ is started. Each integrity constraint containing δ is considered and δ is removed from it. We have to verify that all the resulting constraints are satisfied. Being the constraints denials only (i.e. goals), this corresponds to prove that every resulting goal fails. In the consistency derivation, when an abducible is encountered, an abductive derivation for its complement is started in order to prove its failure, so that the constraint is satisfied.

When the procedure succeeds for the goal G and the initial set of assumptions Δ_i, producing as output the set of assumptions Δ_o, we say that T *abductively derives* G or that G is *abductively derivable* from T and we write $T \vdash_{\Delta_i}^{\Delta_o} G$.

In [6] it has been proved that the proof procedure is *sound* and *weakly complete* with respect to the abductive model semantics of [6] under a number of restriction. We will present these results in details in section 4.

2.3 Inductive Logic Programming

Now we recall some basics on ILP. We first give a definition of the ILP problem [5]:

Given:

– a set $\mathcal{P}$ of possible programs
– a set E^+ of positive examples
– a set E^- of negative examples
– a consistent logic program B such that $B \nvdash e^+$ for at least one $e^+ \in E^+$.

Find:

– a logic program $P \in \mathcal{P}$ such that
  · $\forall e^+ \in E^+$, $B \cup P \vdash e^+$ (*completeness*)
  · $\forall e^- \in E^-$, $B \cup P \nvdash e^-$ (*consistency*).

Let us introduce some terminology. The sets E^+ and E^- form the *training set*. The program P that we want to learn is the *target program* and the predicates which are defined in it are *target predicates*. The program B is called *background knowledge* and contains the definitions of the predicates that are already known. We say that program P *covers* example e if $P \cup B \vdash e$. Therefore the conditions that the program P must satisfy in order to be a solution for the ILP problem can be expressed as "P must cover all positive examples (P is *complete*) and must not cover any negative example (P is *consistent*)". The set $\mathcal{P}$ is called the *hypothesis space*. The importance of this set lies in the fact that it defines the search space of the ILP system. In order to be able to effectively learn a program, this space must be restricted as much as possible. If the space is not restricted, the search could result infeasible. The *language bias* (or simply *bias* in this paper) is a description of the hypothesis space. Many formalisms have been introduced in order to describe this space [5], we will consider only a very simple bias in the form of a set of literals which are allowed in the body of the clauses for the target predicates.

Let us now consider a simple example.

Example 3 *Suppose we want to learn the concept $grandfather$ from the background knowledge:*

$father(X, Y) \leftarrow parent(X, Y), male(X).$
$parent(john, mary).$
$parent(ann, mary).$
$parent(mary, steve).$
$male(john).$
$female(mary).$

and the training sets:

$E^+ = \{grandfather(john, steve)\}$
$E^- = \{grandfather(ann, steve), grandfather(john, mary)\}$

```
Initialize $H := \emptyset$
**repeat** (*Covering loop*)
    Generate one clause $c$
    Remove from $E^+$ the $e^+$ covered by $c$
    Add $c$ to $H$
**until** $E^+ = \emptyset$ (*Sufficiency stopping criterion*)

Generate one clause $c$:
Select a predicate $p$ that must be learned
Initialize $c$ to be $p(\overline{X}) \leftarrow .$
**repeat** (*Specialization loop*)
    Select a literal $L$ from the language bias
    Add $L$ to the body of $c$
    **if** $c$ does not cover any positive example
        **then** backtrack to different choices for $L$
**until** $c$ does not cover any negative example (*Necessity stopping criterion*)
**return** $c$
(or fail if backtracking exhausts all choices for $L$)
```

Fig. 1. Basic top-down ILP algorithm

Suppose also that the hypothesis space $\mathcal{P}$ is described in this way:
$\mathcal{P}$ is the set of clauses of the type $grandfather(X,Y) \leftarrow \alpha$ where α is a conjunction of literals chosen among the following:

$$\begin{aligned}&father(X,Y), father(X,Z), father(Z,Y),\\ &parent(X,Y), parent(X,Z), parent(Z,Y),\\ &male(X), male(Y), male(Z),\\ &female(X), female(Y), female(Z)\end{aligned}$$

The following program P is a solution to this ILP problem because it covers the positive example and does not cover any of the negative ones:

$$grandfather(X,Y) \leftarrow father(X,Z), parent(Z,Y).$$

There are two broad categories of ILP learning methods: *bottom-up* methods and *top-down* methods. In bottom-up methods clauses are generated by starting with a clause that covers one or more positive examples and no negative example and by generalizing it as much as possible without covering any negative example. In top-down methods clauses are constructed starting with a general clause that covers all positive and negative examples and by specializing it until it does no longer cover any negative example while still covering at least one positive. In this paper, we concentrate on top-down methods. A basic top-down inductive algorithm [5,30] learns programs by generating clauses one after the other. A clause is generated by starting with an empty body and iteratively adding literals to the body. The basic inductive algorithm, adapted from [5] and [30], is sketched in figure 1.

2.4 The Abductive Learning Problem

We consider an extended definition of the ILP learning problem where both the background and target theory are abductive logic programs and abductive derivability is used as the coverage relation instead of deductive derivability.

Let us first define the *correctness* of an abductive logic program T with respect to the training set E^+, E^-. This notion replaces those of completeness and consistency for logic programs.

Definition 4 (Correctness) *An abductive logic program T is correct, with respect to E^+ and E^-, if and only if there exists a Δ such that*

$$T \vdash_{\emptyset}^{\Delta} E^+, not\_E^-$$

where $not\_E^- = \{not\_e^- | e^- \in E^-\}$ and E^+, not_E^- stands for the conjunction of each atom in E^+ and not_E^-

Definition 5 (Abductive Learning Problem) Given

- *a set of positive examples E^+,*
- *a set of negative examples E^-,*
- *an abductive theory $T = \langle P, A, IC \rangle$ as background theory.*
- *a set $\mathcal{T}$ of possible abductive theories*

Find

A new abductive theory $T' = \langle P \cup P', A, IC \rangle$ such that $T' \in \mathcal{T}$ and T' is correct wrt E^+ and E^-.

We say that a conjunction of positive or (negated) negative examples L is *covered* if $T \vdash_{\emptyset}^{\Delta} L$. We say that a negative example e^- is *not covered* (or *ruled out*) if $T \vdash_{\emptyset}^{\Delta} not\_e^-$

Differently from the ILP problem, we require the conjunction of examples to be covered instead of each example singularly. This is required to ensure that the abductive explanations for different examples are consistent with each other.

The abductive program that is learned can contain new rules (eventually containing abducibles in the body), but not new abducible predicates and new integrity constraints.

```
procedure LAP(
    inputs : $E^+, E^-$ : training sets,
        $T = \langle P, A, IC \rangle$ : background abductive theory,
    outputs : $H$ : learned theory, $\Delta$ : abduced literals)

$H := \emptyset$
$\Delta := \emptyset$
repeat
    GenerateRule(input: $T, H, E^+, E^-, \Delta$; output: $Rule, E^+_{Rule}, \Delta_{Rule}$)
    Add to $E^+$ the positive literals of target predicates in $\Delta_{Rule}$
    Add to $E^-$ the complement of negative literals
        of target predicates in $\Delta_{Rule}$
    $E^+ := E^+ - E^+_{Rule}$
    $H := H \cup \{Rule\}$
    $\Delta := \Delta \cup \Delta_{Rule}$
until $E^+ = \emptyset$ (Sufficiency stopping criterion)
output $H, \Delta$
```

Fig. 2. The covering loop

3 The Algorithm

The system LAP is able to solve the Abductive Learning Problem. The algorithm is obtained from the basic top-down ILP algorithm shown in section 2 by substituting the deductive proof procedure of logic programming with the abductive proof procedure defined in [26] for testing the coverage of examples. The system has been implemented in Sicstus Prolog version 3#5 [38].

The covering loop of the algorithm for learning abductive rules is presented in figure 2, the specializing loop in figure 3 and the procedure for testing the coverage of a rule in figure 4. The basic top-down inductive algorithm is extended in the following respects.

First, in order to determine the set of positive examples E^+_{Rule} covered by the generated rule *Rule* (procedure TestCoverage in figure 4), an abductive derivation is started for each positive example. This derivation results in a (possibly empty) set of consistent hypotheses (abduced literals) which, together with the current theory, allows to derive the examples. The abductive procedure takes as input also the set of literals abduced in the derivation of previous examples. In this way, we ensure that the conjunction of examples will be covered because the procedure is allowed only to make assumptions that are consistent with those previously made for covering other examples, as will be proved in section 4.

```
procedure GenerateRule(
    inputs : $T$ : background abductive theory, $H$ : current hypothesis,
        $E^+, E^-$ : training sets, $\Delta$ : current set of abduced literals
    outputs : $Rule$ : rule, $E^+_{Rule}$ : positive examples covered by $Rule$,
        $\Delta_{Rule}$ : set of literals abduced during the derivation of examples)

Select a predicate to be learned $p$
Let $Rule := p(X) \leftarrow true.$
repeat
    select a literal $L$ from the language bias
    add $L$ to the body of $Rule$
    TestCoverage(input: $Rule, T, H, E^+, E^-, \Delta$,
    output: $E^+_{Rule}, consistent_{Rule}, \Delta_{Rule}$)
    if $E^+_{Rule} = \emptyset$
        backtrack to a different choice for $L$
until $consistent_{Rule}$ is true (Necessity stopping criterion)
output $Rule, E^+_{Rule}, \Delta_{Rule}$
```

Fig. 3. The specialization loop

Second, in order to check that no negative example is covered (necessity stopping criterion in figure 3) by the generated rule *Rule*, an abductive derivation is started for the conjunction of each negated negative example (procedure TestCoverage). We test the conjunction of negative examples, instead of testing them singularly as we do for positive examples, because we are interested in clauses that cover not_E^- and not a subset of it, while we are interested in clauses that possibly cover only a subset of E^+. Also in this case, the derivation does not start from an empty set of abducibles, but it starts from the set of abducibles previously assumed. Therefore the set of abducibles is passed on from derivation to derivation and gradually extended. This is done across different clauses as well by enlarging the Δ set at each iteration in procedure LAP (figure 2).

Third, in the covering loop, after the generation of a new clause, the abduced literals regarding target predicates are added to the training set. In particular, positive literals of target predicates are added to E^+, while default literals of target predicates (in their complemented form) are added to E^-. In this way, we ensure that the assumptions made about target predicates will be respected by the rules that will be learned in the future: no new rule will cover an atom assumed false, and all the atoms assumed true will be covered, as will be proved in section 4.

We have increased the ways in which a positive example can be covered and a negative example ruled out. A positive example can be covered by abducing nothing, thus expressing the fact that the example is surely positive, or it can

```
procedure TestCoverage(
    inputs : $Rule$ : rule, $T$ : background abductive theory,
        $H$ : current hypothesis, $E^+, E^-$ : training sets,
        $\Delta$ : current set of abduced literals
    outputs: $E^+_{Rule}$: positive examples covered by $Rule$,
        $consistent_{Rule}$: true if $not\_E^-$ is covered,
        $\Delta_{Rule}$ : set of literals abduced during the derivation of examples)

$E^+_{Rule} := \emptyset$
$\Delta_{in} := \Delta$
for each $e^+ \in E^+$ do
    if AbductiveDerivation($e^+, \langle P \cup H \cup \{Rule\}, A, IC\rangle, \Delta_{in}, \Delta_{out}$)
        succeeds then
            Add $e^+$ to $E^+_{Rule}$
            $\Delta_{in} := \Delta_{out}$
    endif
endfor
if AbductiveDerivation($not\_E^-, \langle P \cup H \cup \{Rule\}, A, IC\rangle, \Delta_{in}, \Delta_{Rule}$)
    succeeds then
    $consistent_{Rule} := true$
else
    $consistent_{Rule} := false$
endif
output $E^+_{Rule}, consistent_{Rule}, \Delta_{Rule}$
```

Fig. 4. Coverage testing

be covered by making some assumptions, thus expressing the fact that we do not have complete confidence in its coverage, but that it is consistent with our current representation of the domain. Similarly, a negative example can be ruled out with certainty, when its negation is derived by abducing nothing, or it can be ruled out under certain assumptions. The learning power of the algorithm is therefore greater than that of an ILP system because it is able to learn even when the knowledge about the domain is not completely specified, as it is often the case for real learning problems, by making hypotheses about the unknown parts of the domain, provided that these are consistent with known integrity constraints.

The system is able to learn as well from information on target predicates expressed in the form of integrity constraints. Before starting the learning process, additional examples are extracted from the constraints using the abductive proof procedure. The process will be described in full details in section 5.3.

4 Properties of the algorithm

The algorithm is sound, under some restriction, but not complete. In this section we give a proof of its soundness and we point out the reasons why it is incomplete.

Let us first adapt the definitions of soundness and completeness for an inductive inference machine, as given by [5], to the new problem definition. We will call Abductive Inductive Inference Machine (AIIM) an algorithm that solves the Abductive Learning Problem. If M is an AIIM, then we shall write $M(\mathcal{T}, E^+, E^-, B) = T$ to indicate that, given the hypothesis space $\mathcal{T}$, positive and negative examples E^+ and E^-, and a background knowledge B, the machine outputs a program T. We write $M(\mathcal{T}, E^+, E^-, B) = \perp$ when M does not produce any output.

With respect to the problem definition of section 2.4, the definition of soundness and completeness are

Definition 6 (Soundness) *An AIIM M is sound if and only if if $M(\mathcal{T}, E^+, E^-, B) = T$, then $T \in \mathcal{T}$ and T is correct with respect to E^+ and E^-.*

Definition 7 (Completeness) *An AIIM M is complete if and only if if $M(\mathcal{T}, E^+, E^-, B) = \perp$, then there is no $T \in \mathcal{T}$ that is correct with respect to E^+ and E^-.*

The proof of soundness of the algorithm is based on the theorems of soundness and weak completeness of the abductive proof procedure given in [6]. We will first present the results of soundness and completeness for the proof procedure and then we will prove the soundness of our algorithm.

The theorems of soundness and weak completeness (theorems 7.3 and 7.4 in [6]) have been extended by considering the goal to be proved as a conjunction of abducible and non-abducible atoms instead of a single non-abducible atom and by considering an initial set of assumptions Δ_i. The proofs of these extensions are straightforward, given the original theorems.

Theorem 8 (Soundness) *Let us consider an abductive logic program. Let L be a conjunction of atoms. If there exists an abductive derivation from $(\leftarrow L\ \Delta_i)$ to $(\leftarrow\ \Delta_o)$ then there exists an abductive model M such that $M \models L$ and $\Delta_o \subseteq M^{\mathcal{A}} \cup M^{\mathcal{D}}$, where $M^{\mathcal{A}} = M \cap \mathcal{L}^{\mathcal{A}}$ and $M^{\mathcal{D}} = M \cap \mathcal{L}^{\mathcal{D}}$.*

Theorem 9 (Weak completeness) *Let us consider an abductive logic program. Let L be a conjunction of atoms. Suppose that every selection of rules in the proof procedure for L terminates with either success or failure. If there exists an abductive model M such that $M \models L$, then there exists a selection of* *rules such that the procedure succeeds for L returning Δ, where $\Delta \subseteq M^{\mathcal{A}} \cup M^{\mathcal{D}}$.*

We need as well the following lemma.

Lemma 10 *Let us consider an abductive logic program. Let L be a conjunction of atoms. If there exists an abductive derivation from $(\leftarrow L\ \{\})$ to $(\leftarrow \Delta)$ then $lhm(P \cup \Delta) \models L$*[1]*, where $lhm(P \cup \Delta)$ is the least Herbrand model of $P \cup \Delta$.*

PROOF. Derives directly from theorem 5 in [18]. □

The theorems of soundness and weak completeness are true under a number of assumptions:

– the abductive logic program must be ground,
– the abducibles must not have a definition in the program,
– the integrity constraints are denials with at least one abducible in each constraint.

Moreover, the completeness theorem is limited by the assumption that the proof procedure for L always terminates.

The soundness of LAP is limited as well by these assumptions. In the following, we discuss the restrictions imposed by these assumptions.

The requirement that the program is ground is not restrictive in the case in which there are no function symbols in the program and therefore the Herbrand universe is finite. In this case, in fact, we can obtain a finite ground program from a non-ground one by grounding in all possible ways the rules and constraints in the program.

The restriction on the absence of a (partial) definition for the abducible does not reduce the generality of the results since, in the case in which abducible predicates have definitions in T, we can apply a transformation to T so that the resulting program T' has no definition for abducible predicates. This is done by introducing an auxiliary predicate δ_a for each abducible predicate a and by adding the clause

$$a(\overline{\mathbf{x}}) \leftarrow \delta_a(\overline{\mathbf{x}}).$$

The predicate a is no longer abducible, whereas δ_a is now abducible. In this way, we are able to deal as well with partial definitions for abducible predicates, and this is particularly important when learning from incomplete data,

[1] i.e., if $L = l_1 \wedge l_2 \wedge \ldots \wedge l_n$, then $M \models L \Leftrightarrow \forall i : i = 1 \ldots n,\ l_i \in lhm(P \cup \Delta)$.

because the typical situation is exactly to have some predicates whose definition is incomplete, as will be shown in section 5.1.

The requirement that each integrity constraint contains an abducible predicate is not restrictive because we use constraints only for limiting assumptions and therefore a constraint without abducible predicates would be useless.

The most restrictive requirement is the one on the termination of the proof procedure. However, it can be shown that the procedure always terminate for *call-consistent* programs [26], i.e., if no predicate depends on itself through an odd number of negative recursive calls (e.g., $p \leftarrow not\_p$).

We need as well the following two theorems.

Theorem 11 *If* $T = \langle P, A, IC\rangle \vdash_{\emptyset}^{\Delta_1} L_1$ *and* $T \vdash_{\Delta_1}^{\Delta_2} L_2$*, where* L_1 *and* L_2 *are two conjunctions of atoms, then* $T \vdash_{\emptyset}^{\Delta_2} L_1 \wedge L_2$*.*

PROOF. From $T \vdash_{\emptyset}^{\Delta_1} L_1$ and lemma 10 we have that $lhm(P \cup \Delta_1) \models L_1$.

From the definition of abductive proof procedure we have that $\Delta_1 \subseteq \Delta_2$. Since we consider the positive version of programs, $P \cup \Delta_1$ and $P \cup \Delta_2$ are definite logic programs. From the monotonicity of definite logic programs $lhm(P \cup \Delta_1) \subseteq lhm(P \cup \Delta_2)$ therefore $lhm(P \cup \Delta_2) \models L_1$.

From $T \vdash_{\Delta_1}^{\Delta_2} L_2$, by the soundness of the abductive proof procedure, we have that there exists an abductive model M_2 such that $M_2 \models L_2$ and $\Delta_2 \subseteq M_2^{\mathcal{A}} \cup M_2^{\mathcal{D}}$. From proposition 2, there exists a set H_2 such that $M_2 = lhm(P \cup H_2)$. Since abducible and default predicates have no definition in P, we have that $M_2 \cap (\mathcal{L}^{\mathcal{A}} \cup \mathcal{L}^{\mathcal{D}}) = H_2$ and $\Delta_2 \subseteq H_2$. Therefore $M_2 \models L_1$.

From $M_2 \models L_2$ and from the weak completeness of the abductive proof procedure, we have that $T \vdash_{\Delta_1}^{\Delta_2} L_1 \wedge L_2$ $\square$.

Theorem 12 *If* $T = \langle P, A, IC\rangle \vdash_{\emptyset}^{\Delta_1} L_1$ *and* $T' = \langle P \cup P', A, IC\rangle \vdash_{\Delta_1}^{\Delta_2} L_2$*, then* $T' \vdash_{\emptyset}^{\Delta_2} L_1 \wedge L_2$*.*

PROOF. Very similar to the proof of theorem 11. $\square$

We can now give the soundness theorem for our algorithm.

Theorem 13 (Soundness) *The AIIM LAP is sound.*

PROOF. Let us consider first the case in which the target predicates are not abducible and therefore no assumption is added to the training set. In order to prove that the algorithm is sound, we have to prove that, for any given sets E^+ and E^-, the program $T' = \langle B \cup H, A, IC \rangle$ that is output by the algorithm is such that

$$T' \vdash_{\emptyset}^{\Delta} E^+, not\_E^-$$

LAP learns the program T' by iteratively adding a new clause to the current hypothesis, initially empty. Each clause is tested by trying an abductive derivation for each positive and for the conjunction of each (negated) negative example. Let $E_c^+ = \{e_1^+ \dots e_{n_c}^+\}$ be the set of positive examples whose conjunction is covered by clause c.

Clause c is added to the current hypothesis H when:

$$\exists E_c^+ \subseteq E^+ : \quad \forall i \in \{1 \dots n_c\} : \quad \langle P \cup H \cup \{c\}, A, IC \rangle \vdash_{\Delta_{i-1}^+}^{\Delta_i^+} e_i^+$$

$$\langle P \cup H \cup \{c\}, A, IC \rangle \vdash_{\Delta_{n_c}^+}^{\Delta^-} not\_E^-$$

where $\Delta_0^+ = \Delta_H$, $\Delta_{i-1}^+ \subseteq \Delta_i^+$ and $\Delta_{n_c}^+ \subseteq \Delta^-$.

By induction on the examples and using theorem 11, we can prove that

$$\langle P \cup H \cup \{c\}, A, IC \rangle \vdash_{\Delta_H}^{\Delta_{H \cup \{c\}}} E_c^+, not\_E^-$$

where $\Delta_{H \cup \{c\}} = \Delta^-$.

At this point, it is possible to prove that

$$T' \vdash_{\emptyset}^{\Delta} E_{c_1}^+ \cup \dots \cup E_{c_k}^+, not\_E^-$$

by induction on the clauses and using theorem 12. From this and from the sufficiency stopping criterion (see figure 2) we have that $E_{c_1}^+ \cup \dots \cup E_{c_k}^+ = E^+$.

We now have to prove soundness when the target predicates are abducible as well and the training set is enlarged during the computation. In this case, if the final training sets are E_F^+ and E_F^-, we have to prove that

$$T' \vdash_{\emptyset}^{\Delta} E_F^+, not\_E_F^-$$

If a positive assumption is added to E^+, then the resulting program will contain a clause that will cover it because of the sufficiency stopping criterion. If a negative assumption not_e^- is added to E^- obtaining E'^-, clauses that will be generated afterwards will surely derive $not\_E'^-$. We have to prove that also clauses generated before allow to derive $not\_E'^-$. Consider a situation where not_e^- has been assumed during the testing of the last clause added to H. We have to prove that

$$\langle P \cup H, A, IC \rangle \vdash_{\emptyset}^{\Delta} E_H^+, not\_E^- \Rightarrow \langle P \cup H, A, IC \rangle \vdash_{\emptyset}^{\Delta} E_H^+, not\_E'^-$$

where $not\_e^- \in \Delta$ and $e^- \in E'^-$. From the left part of the implication and for the soundness of the abductive proof procedure, we have that there exists an abductive model M such that $\Delta \subseteq M^{\mathcal{A}} \cup M^{\mathcal{D}}$. From $not\_e^- \in \Delta$, we have that $not\_e^- \in M$ and therefore by weak completeness

$$\langle P \cup H, A, IC \rangle \vdash_{\emptyset}^{\Delta} not\_e^-$$

By theorem 11, we have the right part of the implication. $\square$

We turn now to the incompleteness of the algorithm. LAP is incomplete because a number of choice points have been overlooked in order to reduce the computational complexity. The first source of incompleteness comes from the fact that, after a clause is added to the theory, it is never retracted. Thus, it can be the case that a clause not in the solution is learned and the restrictions imposed on the rest of the learning process by the clause (through the covered positive examples and the respective assumptions) prevent the system from finding a solution even if there is one. In fact, the algorithm performs only a greedy search in the space of possible programs, while fully exploring only the smaller space of possible clauses. However, this source of incompleteness is not specific to LAP because most ILP systems perform such a greedy search in the programs space.

The following source of incompleteness, instead, is specific to LAP. For each example, there may be more than one explanation and, depending on the one we choose, the coverage of other examples can be influenced. An explanation for an example may prevent the coverage of other examples, because they do not have any explanation consistent with it, while a different explanation would have allowed such a coverage. Thus, in case of a failure in finding a solution, we should backtrack on example explanations.

We decided to overlook these choice points because they have an high computational cost and we estimated that the cases in which they can prevent the system from finding a solution are relatively infrequent.

5 Applications of LAP

In this section, we describe, by means of examples, four tasks that can be accomplished by LAP.

In this section, we consider the task of learning from incomplete data in the background knowledge and we show how LAP is able to complete the available knowledge.

Let us consider the case of an abductive background theory containing the following clauses, abducibles and constraints:

$$\begin{aligned} P = \quad & \{flat\_tyre(bike_1). \\ & circular(bike_1). \\ & tyre\_holds\_air(bike_3). \\ & circular(bike_4). \\ & tyre\_holds\_air(bike_4).\} \end{aligned}$$

$$A = \quad \{flat\_tyre, broken\_spokes\}$$

$$\begin{aligned} IC = \quad & \{\leftarrow flat\_tyre(X), tyre\_holds\_air(X). \\ & \leftarrow circular(X), broken\_spokes(X).\} \end{aligned}$$

$$\begin{aligned} E^+ &= \{wobbly\_wheel(bike_1), wobbly\_wheel(bike_2), wobbly\_wheel(bike_3)\} \\ E^- &= \{wobbly\_wheel(bike_4)\} \end{aligned}$$

The program must first be transformed in its positive version and then into a program where abducibles have no definition, as shown in section 2.2. For simplicity, we omit the two transformations (apart from the use of the symbol *not_* instead of *not*), and we suppose to apply to the learned program the inverse transformations.

LAP generates the following clause in the specializing loop:

$$wobbly\_wheel(X) \leftarrow flat\_tyre(X).$$

Then the clause is tested. This clause covers $wobbly\_wheel(bike_1)$ because $flat\_tyre(bike_1)$ is specified in the background knowledge and covers $wobbly\_wheel(bike_2)$ by assuming

$$\{flat\_tyre(bike_2), not\_tyre\_holds\_air(bike_2)\}.$$

The assumption $not\_tyre\_holds\_air(bike_2)$ is produced by the abductive proof procedure in order to satisfy the constraint $\leftarrow flat\_tyre(X), tyre\_holds\_air(X)$.

The example $wobbly\_wheel(bike_3)$, however, can not be covered: in fact, we can not assume $flat\_tyre(bike_3)$ since it is inconsistent with the integrity constraint $\leftarrow flat\_tyre(X), tyre\_holds\_air(X)$. and the fact $tyre\_holds\_air(bike_3)$. Then, we check that $not\_wobbly\_wheel(bike_4)$ is derivable in the hypotheses set. This derivation succeeds by abducing $not\_flat\_tyre(bike_4)$.

The system has now verified that the generated clause covers at least one positive example and the conjunction of (the default negation of) negative examples, therefore it can add the clause to the current theory and remove from E^+ the examples covered by it, i.e., $\{wobbly\_wheel(bike_1), wobbly\_wheel(bike_2)\}$. A new iteration of the covering loop is then started with:

$$E^+ = \{wobbly\_wheel(bike_3)\},\ E^- = \{wobbly\_wheel(bike_4)\}$$
$$\Delta = \{flat\_tyre(bike_2), not\_tyre\_holds\_air(bike_2), not\_flat\_tyre(bike_4)\}.$$

In order to cover the remaining positive example $wobbly\_wheel(bike_3)$, the system generates the clause:

$$wobbly\_wheel(X) \leftarrow broken\_spokes(X).$$

which covers the example by abducing

$$\{broken\_spokes(bike_3), not\_circular(bike_3)\}$$

In fact, these assumptions are consistent with the integrity constraint:

$$\leftarrow circular(X), broken\_spokes(X).$$

As for the previous case, the negative example is ruled out by assuming $not\_broken\_spokes(bike_4)$. At this point the algorithm terminates because E^+ becomes empty. The resulting set of assumptions allows the generated program to cover all the initial positive examples and not to cover the negative one.

To the best of our knowledge, only the system RUTH [2] would be able to generate the above theory from such a small amount of information (see section 6 for details on RUTH and on the differences with LAP). Even systems that embody special techniques for handling imperfect data, like FOIL [36], Progol [33] and mFOIL [19], would not be able to learn the above theory because they can not generate clauses that do not cover any positive example, like (without abduction) the clause $wobbly\_wheel(X) \leftarrow broken\_spokes(X)$.

5.2 Learning Rules with Exceptions

The task of learning exceptions to rules is a difficult one because exceptions limit the generality of the rules since they represent specific cases. In the following we show how LAP performs the task of learning exceptions to rules, provided that a number of auxiliary abducible predicates are available. These abducible predicates are used in order to represent exceptions to induced rules.

The bias must contain a number of abducible literals of the form $not\_abnorm_i(X)$ and, for each of them, the positive version of the program will contain an integrity constraint of the form $\leftarrow abnorm_i(X), not\_abnorm_i(X)$. and the predicate $abnorm_i(X)$ in the abducibles. The number of the literals $not\_abnorm_i(X)$ must be estimated so that it is sufficient to take care of all the exceptions that can be encountered by the system.

The example which follows is inspired to [16]. Let us consider the following background abductive theory $T = \langle P, A, IC \rangle$ and training sets E^+ and E^-:

$$
\begin{aligned}
P = \; & \{bird(X) \leftarrow penguin(X). \\
& penguin(X) \leftarrow superpenguin(X). \\
& bird(a). \\
& bird(b). \\
& penguin(c). \\
& penguin(d). \\
& superpenguin(e). \\
& superpenguin(f).\} \\
A = \; & \{abnorm_1, abnorm_2\} \\
IC = \; & \{\leftarrow abnorm_1(X), not\_abnorm_1(X). \\
& \leftarrow abnorm_2(X), not\_abnorm_2(X).\}
\end{aligned}
$$

$$
\begin{aligned}
E^+ &= \{flies(a), flies(b), flies(e), flies(f)\} \\
E^- &= \{flies(c), flies(d)\}
\end{aligned}
$$

Moreover, let the bias be:

$$
\begin{aligned}
& flies(X) \leftarrow \alpha \text{ where} \\
& \alpha \subseteq \; \{superpenguin(X), penguin(X), bird(X), \\
& \qquad not\_abnorm_1(X)\} \\
& abnorm_1(X) \leftarrow \beta \text{ where} \\
& \beta \subseteq \; \{superpenguin(X), penguin(X), bird(X), \\
& \qquad not\_abnorm_2(X)\} \\
& abnorm_2(X) \leftarrow \gamma \text{ where} \\
& \gamma \subseteq \; \{superpenguin(X), penguin(X), bird(X)\}
\end{aligned}
$$

The algorithm first generates the following rule (R_1):

$$flies(X) \leftarrow superpenguin(X).$$

and removes $flies(e)$ and $flies(f)$ from E^+. Then, in the specialization loop, the following rule (R_2) is generated:

$$\mathit{flies}(X) \leftarrow \mathit{bird}(X).$$

which covers all the remaining positive examples, but also the negative ones. In fact, the abductive derivation for $\mathit{not\_flies}(c), \mathit{not\_flies}(d)$ fails. In order to rule out negative examples, the abducible literal $\mathit{not\_abnorm}_1$ is added to the body of R_2 obtaining R_3:

$$\mathit{flies}(X) \leftarrow \mathit{bird}(X), \mathit{not\_abnorm}_1(X).$$

Now, the abductive derivation for $\mathit{not\_flies}(c), \mathit{not\_flies}(d)$ succeeds, provided that abducibles $\{\mathit{abnorm}_1(c), \mathit{abnorm}_1(d)\}$ are assumed true. Moreover, the derivations of the positive examples $\mathit{flies}(a)$ and $\mathit{flies}(b)$ succeed by assuming:

$$\{\mathit{not\_abnorm}_1(a), \mathit{not\_abnorm}_1(b)\}.$$

At this point, we start a new learning phase in which we try to generate a rule for the exceptions, i.e., for the predicate abnorm_1, by considering abduced literals as new training examples for this predicate. Positive literals form the E^+ set, while negative literals form the E^- set:

$$E^+ = \{\mathit{abnorm}_1(c), \mathit{abnorm}_1(d)\}$$
$$E^- = \{\mathit{abnorm}_1(a), \mathit{abnorm}_1(b)\}$$

The resulting induced rule is (R_4):

$$\mathit{abnorm}_1(X) \leftarrow \mathit{penguin}(X).$$

All positive examples for abnorm_1 are covered by assuming nothing, and the conjunction of negative examples is covered as well, therefore we can stop learning. The algorithm ends by producing the following abductive rules:

$$\mathit{flies}(X) \leftarrow \mathit{superpenguin}(X).$$
$$\mathit{flies}(X) \leftarrow \mathit{bird}(X), \mathit{not\_abnorm}_1(X).$$
$$\mathit{abnorm}_1(X) \leftarrow \mathit{penguin}(X).$$

Note that the new abducible abnorm_2 has not been used by the learning process since we have only exceptions for one concept: flies. An equivalent program for the same example can be obtained by using only negation by default in this way

$$\mathit{flies}(X) \leftarrow \mathit{bird}(X), \mathit{not}(\mathit{penguin}(X)).$$
$$\mathit{flies}(X) \leftarrow \mathit{superpenguin}(X).$$

However, our approach allows to learn as well a definition for the class of exceptions, which can provide useful information on the domain at hand. Moreover, in the case in which the exceptions have a complex definition, by learning a different concept for them we can achieve a more understandable theory. For example, suppose we have two exceptions for the concept fly: each bird flies except for red penguins and blue ostriches. In this case, through negation, we can learn:

$$\mathit{flies}(X) \leftarrow \mathit{bird}(X), \mathit{not}(\mathit{penguin}(X), \mathit{red}(X)), \mathit{not}(\mathit{ostrich}(X), \mathit{blue}(X)).$$

Conversely, through abduction we can learn a new concept for the exceptions:

$$\begin{array}{l} \mathit{flies}(X) \leftarrow \mathit{bird}(X), \mathit{not\_abnorm}_1(X). \\ \mathit{abnorm}_1(X) \leftarrow \mathit{penguin}(X), \mathit{red}(X). \\ \mathit{abnorm}_1(X) \leftarrow \mathit{ostrich}(X), \mathit{blue}(X). \end{array}$$

A result similar to ours is obtained in [16], but exploiting "classical" (or, better, syntactic) negation and priority relations between rules rather than default negation as we do. The advantage of our approach, is that we do not build an "ad hoc" algorithm to perform the task but we exploit a general system that can be used as well for other forms of nonmontonic reasoning.

RUTH would be able to perform such a task because it also considers assumptions as new training examples (see section 6).

5.3 Learning from Integrity Constraints on Target Predicates

As a further example, let us consider the abductive program T' generated in the previous section, and add to it the following user-defined constraint, I, on target predicates:

$$\leftarrow \mathit{rests}(X), \mathit{plays}(X).$$

Consider now the new training sets:

$$E^+ = \{\mathit{plays}(a), \mathit{plays}(b), \mathit{rests}(e), \mathit{rests}(f)\} \qquad E^- = \{\}$$

In this case, the information about the target predicates comes not only from the training set, but also from integrity constraints. This kind of integrity constraints differs from the ones that are usually given in the background knowledge because they contain target predicates, while constraints in the background knowledge usually contain only background predicates, either abducible or non-abducible. The generalization process is not limited by negative examples but by integrity constraints. Suppose we generalize the two positive examples for plays in $\mathit{plays}(X)$. This means that for all X, $\mathit{plays}(X)$ is true. However, this is inconsistent with the integrity constraint I because $\mathit{plays}(X)$

cannot be true for e and f.

The information contained in these integrity constraints must be made available in a form that is exploitable by our learning algorithm. In particular, our algorithm learns a definition for the target predicates employing only positive and negative examples of the predicates. Therefore, we must transform the information conveyed by the integrity constraints into new examples for the target concepts, as it is done in the theory revision systems [11,2]. When the knowledge base violates a newly supplied integrity constraint, these systems extract one example from the constraint and revise the theory on the basis of it: in [11] the example is extracted by querying the user on the truth value of the literals in the constraint, while in [2] the example is automatically selected by the system.

In our approach, we generate one or more examples from constraints on target predicates using the abductive proof procedure. This is done by checking the consistency of available examples with the constraints and by making assumptions in order to ensure consistency. Assumptions about target predicates are considered as new negative or positive examples.

In the previous case, the approach for generating examples is as follows. We start an abductive derivation for $\leftarrow plays(a), plays(b), rests(e), rests(f)$. A consistency derivation is then started for each literal. Suppose the selection rule selects first the literal $plays(a)$, in order to have the consistency with the constraint $\leftarrow plays(X), rests(X).$, the literal $not\_rests(a)$ is abduced. The same is done for the other literals in the goal obtaining the set of assumptions

$$\{not\_rests(a), not\_rests(b), not\_plays(e), not\_plays(f)\}$$

that is then transformed in the set of negative examples

$$E^- = \{rests(a), rests(b), plays(e), plays(f)\}$$

Now the learning process applied to the new training sets generates the following correct rules:

$$plays(X) \leftarrow bird(X), not\_abnorm_1(X).$$
$$rests(X) \leftarrow superpenguin(X).$$

In this way, we have increased the power of the learning process. We can learn not only from (positive and negative) examples but also from integrity constraints, as in [11,2,33].

In this section, we show how abduction can be useful when learning recursive predicates.

One possible application of LAP to learning recursive predicates is in the case when no examples for the base case of the recursive definition are contained in the training set. For example:

$$
\begin{aligned}
B = \ & \{prec2(3,1). \\
& prec2(5,3). \\
& prec2(7,5).\} \\
E^+ = & \{odd(7), odd(5)\} \\
E^- = & \{odd(2), odd(4)\}
\end{aligned}
$$

In this case, LAP generates the following theory:

$$
\begin{aligned}
& odd(X) \leftarrow X = 1. \\
& odd(X) \leftarrow prec2(X,Y), odd(Y).
\end{aligned}
$$

LAP first generates the recursive clause and abduces the base case $odd(1)$. Then the assumption is used as a positive example for learning the base clause[2].

To the best of our knowledge, among ILP systems only RUTH would be able to learn such a theory, because no system is able to complete the training set. Even systems that employ both the background knowledge and the training set in the testing of examples, such as MIS [37] with the adaptive strategy, are not able to learn the above theory unless the base case is in the training set.

6 Related Work

We will first mention our previous work in the field, and then we will describe related work by other authors.

In [21] we have presented a preliminary version of the algorithm for learning abductive rules.

In [29] we have proposed an algorithm for learning abductive rules obtained

[2] Supposing to have predicates of the type $X = constant$ in the bias.

modifying the extensional ILP system FOIL [36]. Extensional systems differ from intensional ones (as that presented in this paper) because they employ a different notion of coverage, namely *extensional coverage*. We say that the program P *extensionally covers* example e if there exists a clause of P, $l \leftarrow l_1, \ldots, l_n$ such that $l = e$ and for all i, $l_i \in E^+ \cup lhm(B)$. Thus examples can be used also for the coverage of other examples. This has the advantage of allowing the system to learn clauses independently from each other, eliminating the need of considering different orders in learning the clauses and the need for backtracking. However, it has also a number of disadvantages [13]. In [29] we have shown how the integration of abduction and induction can solve some of the problems of extensional systems when dealing with recursive predicates and programs with negation.

As concerns the integration of abduction and induction, a notable work is that by Dimopoulos and Kakas [17]. In this paper, the authors suggest two methodologies for the integration of abduction in learning. The first consist in using abduction to explain the training data of a learning problem in order to generate suitable or relevant background data on which to base the inductive generalization. The second consists in using abduction in order to cover positive examples and to avoid the coverage of negative ones, as in our approach. The main advantage of the integration in [17] is, as in our framework, that it allows to learn in presence of missing information, and later classify new examples that may be incompletely described. Differently from us, the authors allow the use of integrity constraints for rule specialization, while we rely only on the addition of a literal to the body of the clause. Adding integrity constraints for specializing rules means that each atom derived by using the rules must be checked against the constraints, which can be computationally expensive. Moreover, the results of soundness and weak completeness no longer hold for the extended proof procedure.

In [1] an integrated abductive and inductive framework is proposed in which abductive explanations that may include general rules can be generated by incorporating an inductive learning method into abduction. In particular, the authors first present a general parametric framework based on Bry's work for intensional knowledge base updating [7] which can describe either abduction or induction in logic programming according to the instantiation of the parameters. This framework is used in order to transform a proof procedure for abduction, namely SLDNFA, in a proof procedure for induction, called SLDNFAI. Informally, SLDNFA is extended in order to be able to abduce not only ground facts but also rules. However, the authors obtain a learning framework which is equivalent to the ILP one, they are not able to learn a rule and, at the same time, make assumptions about missing data for the coverage of examples.

The integration of induction and abduction for knowledge base updating has been studied in [11] and [2]. Both systems proposed in these papers perform incremental theory revision: they automatically modify a knowledge base when it violates a newly supplied integrity constraint. When a constraint is violated, they first extract an uncovered positive example or a covered negative example from the constraint (as we do) and then they revise the theory in order to make it coherent with the example, using techniques from incremental concept learning. The system in [11] differs from the system in [2] (called RUTH) because it relies on an oracle for the extraction of examples from constraints, while RUTH works non interactively. In [11] the user is asked about the truth value of each literal in the constraint in order to identify the one whose value should be changed for restoring consistency. The identified literal corresponds to an uncovered positive example or to a covered negative one depending on whether it is in the head or in the body of the constraint. Instead, RUTH avoids asking questions to the user by automatically selecting an erroneous literal from the constraint.

Once the example has been extracted from the constraint, the systems in [11] and [2] call similar inductive operators in order to update the knowledge base. In [11] the authors use the inductive operators of Shapiro's MIS system [37]. For handling an uncovered positive example, they call Shapiro's generalization procedure that first computes, in an abductive manner, the predicate(s) that is responsible for the true fact not being entailed. Then, it generates a clause (if it is an intensional predicate) or a fact (if it is an extensional predicate) so that the positive example is covered. For handling a negative example, they call Shapiro's specialization procedure that identifies an incorrect clause by asking questions to the user and then retracts it. RUTH's operator for handling negative examples is the same as Shapiro's, except for the fact that no query is asked to the user and the incorrect clause is hypothesized. In order to handle positive examples, RUTH has three operators: (*i*) adding an example as a fact in the database, or (*ii*) building a maximally general clause that covers the example using Shapiro's generalization operator, or (*iii*) abducing one or more new facts, that are considered as new training examples.

Let us highlight now the differences between our system and the systems in [11] and [2]. As regards the way in which examples are generated from constraints, we exploit the abductive proof procedure in order to extract new examples from a constraint on target predicates. The new examples are generated as assumptions that allow for the satisfaction of the constraint. The abductive proof procedure exploits all the information available in the knowledge base thus avoiding to ask questions to the users, as in [11]. The procedure ensures the consistency of the generated examples with respect to other integrity constraints, while both systems in [11,2] can generate examples that violate other integrity constraints and new inconsistencies have to be recovered at the next iteration of the learning loop. Instead, we are able to select the examples that allow the minimum revision of the theory. Moreover, it must be observed that, even if we consider only integrity constraints in the form of denials, we are able to generate not only negative examples but also positive ones because we can have negative literals in the constraints.

Another difference is that our system is a batch learner while the systems in [11,2] are incremental learners: since we have all the examples available at the beginning of the learning process, we generate only clauses that do not cover negative examples and therefore we do not have to revise the theory to handle covered negative examples, i.e., to retract clauses. As regards the operators that are used in order to handle uncovered positive examples, we are able to generate a clause that covers a positive example by also making some assumptions, while in [11] they can cover an example either by generating a clause or by assuming a fact for covering it, but not the two things at the same time. RUTH, instead, is able to do this, and therefore would be able to solve the problem presented in sections 5.1. Moreover, RUTH considers abduced literals as new examples, therefore it would be able to solve as well the problems in sections 5.2 and 5.4. However, we differ from both systems because, by using the abductive proof procedure, we search for the set of assumptions that are consistent with the available information and the integrity constraints, and therefore we do not have to revise again the theory as a consequence of these assumptions. Instead, the systems in [11,2] can abduce facts that, even if they allow to cover the current example, can violate other constraints and the systems have to deal with the arisen inconsistency at the next iteration of the learning loop. Both approaches have advantages and disadvantages. By making only consistent assumptions, as we do, we avoid the need for further revision of the theory, but this may result in a failure of the learning process when no assumption is possible. Instead, in [11,2], the systems are able to update the theory in any case, but the revisions may introduce further inconsistency and therefore further iterations are necessary to find a solution. Our approach reflects the fact that we are more concerned with inductive learning rather than with theory revision, and we make assumptions such that the available theory is consistent and must not be updated.

Both our system and [2] are able to generalize the assumptions, as it is shown in section 5.2, because assumptions about target predicates are considered as new training examples. Instead, in [11], when an assumption is made, it is added only to the knowledge base, regardless of the predicate, and no generalization is made on it. Therefore they would not be able to learn exceptions as we have shown in section 5.2.

The ILP system Progol [33] is able to learn from integrity constraints in the form of denials as we do. In fact, in Progol negative examples and integrity constraints are represented in the same way using headless Horn clauses and they are stored internally as clauses with head 'false'. The testing of negative examples and constraints is uniform as well: it is done by seeing whether 'false' is provable. If ‘false’ is provable, it means that either a negative example is covered or a constraint is violated, and therefore the clause under test must be refined.

As concerns the treatment of exceptions to induced rules, in [4] the authors have shown that is not possible, in general, to preserve correct information when incrementally specializing within a classical logic framework. They avoid this empass by using learning algorithms which employ a nonmonotonic knowledge representation. Several other authors have also addressed this problem, in the context of Logic Programming, by allowing for exceptions to (possibly induced) rules [16,10]. In these frameworks, nonmonotonicity and exceptions are dealt with by learning logic programs with negation. Our approach in the treatment of exceptions is very related to [16]. They rely on a language which uses a limited form of “classical” (or, better, syntactic) negation together with a priority relation among the sentences of the program [27]. Nonetheless, a program written in this language can be easily transformed into a normal logic program, by introducing new predicates and capturing the priority between rules into the notion of negation by default. In this respect, when the starting background knowledge is a (definite) logic program, their and our approach coincide. On the other hand, in [20] the authors have argued that negation by default can be seen as a special case of abduction. Thus, in our framework, relying on ALP [23], we can achieve greater generality than [16]. In [16], the authors also proved that their algorithm terminates under proper conditions. In particular, they consider a *hierarchy language bias* (which corresponds to impose that more general rules have lower priority) plus a *single clause bias* (which imposes to generate a single clause covering all the given examples). In our case, we do not have this restriction, we can generate more than one clause to cover positive examples. We avoid the problem of non-termination by considering only a finite number of predicates $abnorm_i$.

As concerns learning from incomplete information, many ILP systems include facilities in order to handle this problem, for example FOIL [36], Progol [33], mFOIL [19]. The approach that is followed by all these systems is fundamentally different with respect to ours: they are all based on the use of heuristic necessity and sufficiency stopping criteria and of special heuristic functions for guiding the search. The heuristic stopping criteria relax the requirements of consistency and completeness of the learned theory, allowing the system to deal with imperfect data in general, comprehending noisy data (data with random errors in training examples and in the background knowledge) and incomplete data. In this sense, they are more general than our approach, because we are not able to deal with noisy data. Their approach is equivalent to discarding some examples, considering them as noisy or insufficiently specified, while in our approach no example is discarded, the theory must be complete and consistent (in the abductive sense) with each example. We relax the completeness and consistency requirements by substituting it with correctness. The effect is that we do not have the requirements of having "most" examples covered and we are able to learn rules that would cover no example if abduction were not used, as the rule $wobbly\_wheel(X) \leftarrow broken\_spokes(X)$. in section 5.1, that would be impossible for the above mentioned ILP systems.

7 Conclusions and Future Work

We have presented an approach for the integration of abduction and induction in logic programming. We have defined a new learning problem, called Abductive Learning Problem, that extends the ILP problem by considering both the background and target theory as abductive theories and by using abductive derivability as the coverage relation in substitution of deductive derivability. By integrating abduction and induction, we obtain two results: we increase the expressive power of the background and target theories and we make it possible to learn in presence of incomplete information.

We have presented the system LAP (Learning Abductive Programs) that solves the extended problem and is obtained from the basic top-down algorithm of ILP by using the abductive proof procedure in substitution of the Prolog proof procedure for testing the coverage of examples. We have shown, by means of examples, four different tasks that can be performed by the system: learning from incomplete information, learning exceptions, learning from integrity constraints on target predicates and learning recursive predicates. LAP has been implemented in Sicstus Prolog 3#5.

In the future, we will test the algorithm on real domains where the incompleteness of the data causes problems to others ILP systems. As regards the theoretical aspects, we will investigate the problem of extending the algorithm proposed for learning full abductive theories, comprehending as well integrity constraints. The integration of the algorithm with other systems for learning constraints, such as Claudien [12] and ICL [14], proposed in [28], seems very promising and more work is needed to reach the objective of a system for learning full abductive theories.

Acknowledgment

This research was partially funded by the MURST Project 40% "Rappresentazione della conoscenza e meccanismi di ragionamento". Fabrizio Riguzzi would like to thank Antonis Kakas for many interesting discussions on the topics of this paper.

References

[1] H. Adé and M. Denecker. AILP: Abductive inductive logic programming. In C.S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1201–1207. Morgan Kaufmann, 1995.

[2] H. Adé, B. Malfait, and L. De Raedt. RUTH: An ILP theory revision system. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, volume 869 of *Lecture Notes in Artificial Intelligence*, pages 336–345. Springer-Verlag, 1994.

[3] K.R. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier Science Publisher, 1990.

[4] M. Bain and S. Muggleton. Non-monotonic learning. In S. Muggleton, editor, *Inductive Logic Programming*, pages 145–161. Academic Press, 1992.

[5] F. Bergadano and D. Gunetti. *Inductive Logic Programming.* MIT press, 1995.

[6] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A unifying view for logic programming with non-monotonic reasoning. *Theoretical Computer Science*, 184:1–59, 1997.

[7] F. Bry. Intensional updates: Abduction via deduction. In D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 561–578. The MIT Press, 1990.

[8] W. W. Cohen. Abductive explanation-based learning: A solution to the multiple inconsistent explanation problem. *Machine Learning*, 8:167–219, 1992.

[9] L. Console, D. Theseider Duprè, and P. Torasso. Abductive reasoning through direct deduction from completed domains models. In Z. Ras, editor, *Proceedings of the 3rd International Symposium on Methodologies for Intelligent Systems*, pages 175–182. North Holland, 1989.

[10] L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept-learning. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 207–212. Pitman, 1990.

[11] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.

[12] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.

[13] L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 221–240. J. Stefan Institute, 1993.

[14] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 5th Conference on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1995.

[15] M. Denecker, L. De Raedt, P. Flach, and A. Kakas, editors. *Proceedings of ECAI96 Workshop on Abductive and Inductive Reasoning.* Catholic University of Leuven, 1996.

[16] Y. Dimopoulos and A. Kakas. Learning non-monotonic logic programs: Learning exceptions. In N. Lavrač and S. Wrobel, editors, *Proceedings of the 8th European Conference on Machine Learning*, volume 912 of *Lecture Notes in Artificial Intelligence*, pages 122–137. Springer-Verlag, 1995.

[17] Y. Dimopoulos and A. Kakas. Abduction and inductive learning. In *Advances in Inductive Logic Programming.* IOS Press, 1996.

[18] P.M. Dung. Negation as hypothesis: An abductive foundation for logic programming. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 3–17. MIT Press, 1991.

[19] S. Džeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, 1991.

[20] K. Eshghi and R.A. Kowalski. Abduction compared with Negation by Failure. In *Proceedings of the 6th International Conference on Logic Programming*, 1989.

[21] F. Esposito, E. Lamma, D. Malerba, P. Mello, M. Milano, F. Riguzzi, and G. Semeraro. Learning abductive logic programs. In Denecker et al. [15].

[22] P. Flach and A. Kakas, editors. *Proceedings of IJCAI97 Workshop on Abductive and Inductive Reasoning.* 1997.

[23] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1993.

[24] A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 5, pages 233–306. Oxford University Press, 1997.

[25] A.C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 1990.

[26] A.C. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In *Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence*, 1990.

[27] A.C. Kakas, P. Mancarella, and P.M. Dung. The acceptability semantics for logic programs. In *Proceedings of the 11th International Conference on Logic Programming*, 1994.

[28] A.C. Kakas and F. Riguzzi. Learning with abduction. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, 1997.

[29] E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Introducing Abduction into (Extensional) Inductive Logic Programming Systems. In M. Lenzerini, editor, *AI*IA97, Advances in Artificial Intelligence, Proceedings of the 5th Congress of the Italian Association for Artificial Intelligence*, number 1321 in LNAI. Springer-Verlag, 1997.

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

[31] R. Michalski, J.G. Carbonell, and T.M. Mitchell (eds). *Machine Learning - An Artificial Intelligence Approach.* Springer-Verlag, 1984.

[32] R. Michalski, J.G. Carbonell, and T.M. Mitchell (eds). *Machine Learning - An Artificial Intelligence Approach Vol. II.* Morgan Kaufmann, 1986.

[33] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[34] P. O'Rourke. Abduction and explanation-based learning: Case studies in diverse domains. *Computational Intelligence*, 10:295–330, 1994.

[35] J. Pearl. Embracing causality in formal reasoning. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 369–373, Seattle, WA, 1987.

[36] J.R. Quinlan and R.M. Cameron-Jones. Introduction of logic programs: FOIL and related systems. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):287–312, 1995.

[37] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[38] Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual*, 1997.

Appendix

In the following we recall the abductive proof procedure used by our algorithm. The procedure is taken from [26]. It is composed by two phases: abductive derivation and consistency derivation.

Abductive derivation
An abductive derivation from $(G_1\ \Delta_1)$ to $(G_n\ \Delta_n)$ in $\langle P, Ab, IC\rangle$ via a selection rule R is a sequence

$$(G_1\ \Delta_1), (G_2\ \Delta_2), \ldots, (G_n\ \Delta_n)$$

such that each G_i has the form $\leftarrow L_1, \ldots, L_k$, $R(G_i) = L_j$ and $(G_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

(A1) If L_j is not abducible or default, then $G_{i+1} = C$ and $\Delta_{i+1} = \Delta_i$ where C is the resolvent of some clause in P with G_i on the selected literal L_j;

(A2) If L_j is abducible or default and $L_j \in \Delta_i$ then $G_{i+1} = \leftarrow L_1, \dots, L_{j-1}, L_{j+1}, \dots, L_k$ and $\Delta_{i+1} = \Delta_i$;

(A3) If L_j is abducible or default, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$ and there exists a *consistency derivation* from $(L_j\ \Delta_i \cup \{L_j\})$ to $(\{\}\ \Delta')$ then $G_{i+1} = \leftarrow L_1, \dots, L_{j-1}, L_{j+1}, \dots, L_k$ and $\Delta_{i+1} = \Delta'$.

Steps $(A1)$ and $(A2)$ are SLD-resolution steps with the rules of P and abductive or default hypotheses, respectively. In step $(A3)$ a new abductive or default hypotheses is required and it is added to the current set of hypotheses provided it is consistent.

Consistency derivation
A consistency derivation for an abducible or default literal α from $(\alpha,\ \Delta_1)$ to $(F_n\ \Delta_n)$ in $\langle P, Ab, IC \rangle$ is a sequence

$$(\alpha\ \Delta_1), (F_1\ \Delta_1), (F_2\ \Delta_2), \dots, (F_n\ \Delta_n)$$

where :

(Ci) F_1 is the union of all goals of the form $\leftarrow L_1, \dots, L_n$ obtained by resolving the abducible or default α with the denials in IC with no such goal been empty, $\leftarrow$;

(Cii) for each $i > 1$, F_i has the form $\{\leftarrow L_1, \dots, L_k\} \cup F_i'$ and for some $j = 1, \dots, k$ $(F_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

(C1) If L_j is not abducible or default, then $F_{i+1} = C' \cup F_i'$ where C' is the set of all resolvents of clauses in P with $\leftarrow L_1, \dots, L_k$ on the literal L_j and $\leftarrow \notin C'$, and $\Delta_{i+1} = \Delta_i$;

(C2) If L_j is abducible or default, $L_j \in \Delta_i$ and $k > 1$, then
$F_{i+1} = \{\leftarrow L_1, \dots, L_{j-1}, L_{j+1}, \dots, L_k\} \cup F_i'$
and $\Delta_{i+1} = \Delta_i$;

(C3) If L_j is abducible or default, $\overline{L_j} \in \Delta_i$ then $F_{i+1} = F_i'$ and $\Delta_{i+1} = \Delta_i$;

(C4) If L_j is abducible or default, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$, and there exists an *abductive derivation* from $(\leftarrow \overline{L_j}\ \Delta_i)$ to $(\leftarrow\ \Delta')$ then $F_{i+1} = F_i'$ and $\Delta_{i+1} = \Delta'$.

In case (C1) the current branch splits into as many branches as the number of resolvents of $\leftarrow L_1, \dots, L_k$ with the clauses in P on L_j. If the empty clause is one of such resolvents the whole consistency check fails. In case (C2) the goal under consideration is made simpler if literal L_j belongs to the current set of hypotheses Δ_i. In case (C3) the current branch is already consistent under the assumptions in Δ_i, and this branch is dropped from the consistency checking. In case (C4) the current branch of the consistency search space can be dropped provided $\leftarrow \overline{L_j}$ is abductively provable.

Given a query L, the procedure succeeds, and returns the set of abducibles Δ if there exists an abductive derivation from $(\leftarrow L\ \{\})$ to $(\leftarrow\ \Delta)$. With abuse of terminology, in this case, we also say that the abductive derivation succeeds.