Learning Abductive Logic Programs

F. Esposito(1), E. Lamma(2), D. Malerba(1), P. Mello(3), M. Milano(2), F. Riguzzi(2), G. Semeraro(1)

(1) Dipartimento di Informatica, Università di Bari
Via Orabona 4, 70126 Bari, Italy
semeraro@vm.csata.it

(2) DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{elamma,mmilano}@deis.unibo.it

(3) Istituto di Ingegneria, Università di Ferrara
Via Saragat, 41100 Ferrara, Italy
pmello@ing.unife.it

Abstract. We propose an approach for the integration of abduction and induction in Logic Programming. In particular, we show how it is possible to learn, by induction, an abductive logic program. Abducibles and integrity constraints can be specified by the user as content of the background knowledge or can be generated by the abductive/inductive process. We ground our framework on the generalized stable model semantics defined for abductive logic programs, and its associated proof procedure. By integrating Inductive Logic Programming with Abductive Logic Programming we can learn in presence of incomplete knowledge, take into account negative examples and generate exceptions to (possibly induced) rules.

1 Introduction

Both abduction and induction have been recognized as powerful mechanisms for hypothetical reasoning in the presence of incomplete knowledge [9, 10, 16, 19, 22, 23]. 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.

In this work, we address the problem of how we can adapt Inductive Logic Programming (ILP, for short [4]) so that we can learn in cases where there is some missing information from the background theory, by exploiting proof procedures for Abductive Logic Programming (ALP, for short [18]). In particular, we propose a general approach where it is possible to learn, by induction, an abductive logic program. In more detail, the background knowledge of the inductive framework here considered is now an abductive logic program [16, 18], i.e., a logic program P (possibly with abducible atoms in clause bodies), a set of abducibles A and a set of integrity constraints IC. Thus, we start from abductive logic programs, and generate abductive programs as well. Abducibles and integrity constraints can be specified by the user as content of the background knowledge, and are also generated by the learning process.

The inductive algorithm here applied is an extension of a top-down algorithm adopted in ILP [4, 17]. The extended algorithm takes into account abducibles and integrity constraints, and is intertwined with the proof procedure defined in [20] for abductive logic programs.

By integrating ILP with ALP, we develop a generic framework for the problem of completing incomplete knowledge, as, for instance, in [14, 2]. We can take into account user-defined abducibles, and generate exceptions (as in [13]) to induced rules, by the introduction of new abducibles and integrity constraints as well.

The underlying declarative semantics for our framework is the generalized stable model semantics, defined for abductive logic programs by Kakas and Mancarella [19]. If no generalized stable model exists for the abductive program, then a 3-valued semantics can be adopted, as in [6].

The paper is organized as follows. In section 2, we describe the basic structure of our framework in terms of its top-level algorithm. Section 3 presents examples in order to better clarify the framework behaviour. Section 4 discusses some issues concerning the approach to abduction, optimizations to be introduced in the algorithm and the generation of integrity constraints. Related work is discussed in section 5. Conclusions and future work follow.

2 The Basic Algorithm

In the following, we first briefly recall Abductive Logic Programming (ALP), then we describe, at a high level, a basic Inductive Logic Programming (ILP) algorithm suitably extended in order to deal with abduction and incomplete knowledge. This extended algorithm exploits proof procedures for ALP, and the abductive and consistency derivation defined in [20] in particular. For the sake of completeness, the abductive and consistency derivation used, taken from [20], are reported in the Appendix.

In the context of abduction, missing information is represented by (user-defined) abducible predicates, possibly constrained by integrity constraints. The background theory is thus an *abductive logic program*, i.e., a triple $\langle P, \mathcal{A}, IC\rangle$ where:

- P is a normal logic program, that is, a set of clauses of the form $A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_{m+n}$, where $m, n \geq 0$ and each A_i $(i = 1, \ldots, m+n)$ is an atom;
- $\mathcal{A}$ a set of *abducible predicates*;
- IC is a set of integrity constraints (denials, for simplicity).

Following [19], this program can be transformed into its *positive version* by transforming P and IC. The basic idea is to view default literals of the kind $not\ p$ as new positive (abducible) atoms not_p (as in, e.g., [16, 15]) and introduce additional integrity constraints of the kind: $\leftarrow p, not\_p$. From now on, when speaking of an abductive logic program we intend its positive version.

The purpose of our framework is (in analogy with [14]) is to synthesize a new abductive theory $\langle P', \mathcal{A}', IC'\rangle$, starting from an abductive logic program $\langle P, \mathcal{A}, IC\rangle$ and a set of positive and negative observations for a concept c. The new theory contains rules for the concept c, and new abducibles and integrity constrains are possibly introduced in order to cope with exceptions and rule specialization. In particular, in order to rule out negative examples, we intertwine the introduction of new abducibles with the generation of integrity constraints (and rules) in order to block (some instances of) a rule.

The basic, top-down inductive algorithm [4] learns programs by generating clauses one after the other, and generates clauses by means of specialization. Let T denote the set of induced clauses, initially empty, and E^+ and E^- be the training set (positive and negative examples, respectively). The basic inductive algorithm can be sketched as follows:

while some positive example in E^+ is not covered by a clause in T **do**

- Generate one clause C
- Remove from E^+ the positive examples covered by C
- Add C to T

where the generation of a clause is performed as follows:

Select a predicate p that must be learned, and set clause C to be $p(\overline{X}) \leftarrow$.
while C covers some negative example **do**

- Select a literal L from the language bias
- Add L to the antecedent of C
- **if** C does not cover any positive example **then** backtrack to different choices for L

return C (or fail if backtracking exhausts all choices for L).

This basic inductive algorithm is extended into the following respects:

- First, when clause C is generated in order to cover positive examples, this explanation might also contain abducibles, in analogy with the framework in [14]. Thus, the selected literal L is in the language bias, i.e., it can be a literal of the background knowledge, of the training set or a user-defined abducible.
- Second, in order to determine the positive examples covered by the generated clause C, and to be removed, an *abductive* derivation is started (this is achieved by exploiting the abductive proof procedure defined in [20]). In the context of abductive reasoning, it is matter of discussion (see section 4) whether the set of positive examples $\{e_1^+, \ldots, e_n^+\}$, has to be intended as a conjunction (i.e., a successful abductive derivation exists for $\leftarrow e_1^+, \ldots, e_n^+.$) or they can be proved separately, possibly with incompatible sets of abducibles. In terms of model-theory, the issue is whether we are interested in a model satisfying all the positive examples or it suffices that each positive example belongs to at least one model. Of course this problem does not arise in standard ILP, since in this case only one least Herbrand model exists.
- As well, in order to check that no negative example is covered by the generated clause C, an *abductive* derivation is started (this is achieved, again, by exploiting the abductive proof procedure defined in [20]). In particular, if the set of negative examples is $\{e_1^-, \ldots, e_n^-\}$, then an abductive derivation for $\leftarrow not\_e_1^-, \ldots, not\_e_n^-.$ is attempted. If this derivation succeeds returning an empty set of abducibles, no negative example is covered. If a (non-empty) set of abducibles Δ is returned, these hypotheses must be assumed true in order to rule out negative examples. Abducibles in Δ can be positive atoms or default atoms of the kind not_p. The set of positive abducibles is then added to the background knowledge, whereas default abducibles can be added as additional integrity constraints (of the kind $\leftarrow p.$). A new phase of the inductive algorithm takes place in order to possibly generalize these (positive and default) abducibles.
- Finally, when generating a clause, rather than failing if backtracking exhausts all choices for literal L, we choose to exploit abduction in order to rule out covered negative examples possibly introducing new abducibles and integrity constraints. If no abducible is determined at previous step (i.e., the abductive derivation for $\leftarrow not\_e_1^-, \ldots, not\_e_n^-.$ fails), instead, then a new abducible predicate is automatically generated. In particular, a new atom, not_abnorm_i, is introduced in the body of the generated rule. Previous step is then tried again, in order to determine the set of abducibles, Δ, leading the abductive derivation for $\leftarrow not\_e_1^-, \ldots, not\_e_n^-.$ to succeed. This set will contain atoms for the newly introduced abductive predicate $abnorm_i$. Intuitively, at this point, we could just add these facts to the learned program, or better, try to generate a rule for them. Then, a new rule for predicate $abnorm_i$ is synthesized as exception to the previous one in order to rule out the negative examples, as done in [13]. The rule generalization for the new abducible would lead to a loop in some cases (see

the example in section 3.3). To avoid loop, we prefer to restrict the algorithm and not generate a rule for an abducible predicate containing only (new) abducible predicates in its body. Thus, rule generation possibly returns more than one clause (in general, it will return clauses and integrity constraints).

This behaviour is informally explained in section 3 through examples.

3 Examples

3.1 Learning Rules with Exceptions

The first example is inspired to [13]. Let us consider the following background knowledge P:

$bird(X) \leftarrow penguin(X).$
$penguin(X) \leftarrow superpenguin(X).$
$bird(a).$
$bird(b).$
$penguin(c).$
$penguin(d).$
$superpenguin(e).$
$superpenguin(f).$

and the set of examples:

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

Let A and IC be the empty set.

Let *bird*, *penguin* and *superpenguin* be the bias for *flies*. The algorithm generates the following rule (R_1):

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

and removes $flies(e)$ and $flies(f)$ from E^+. Then, rule generation produce the following rule (R_2):

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

which covers all the remaining positive examples, but also the negative ones, i.e., in other words, when added to the background knowledge it generates an inconsistency. Inconsistency is detected by checking that the abductive derivation for $\leftarrow not\_flies(c), not\_flies(d)$ succeeds. This means that some negative example (at least one) is covered. In order to restore consistency, by exploiting abduction, a new (abducible) predicate is generated, not_abnorm_1, and added to the body of R_2. Moreover, the integrity constraint:

$\leftarrow abnorm_1(X), not\_abnorm_1(X).$

is also added to the background knowledge. Now, the abductive derivation for $\leftarrow not\_flies(c), not\_flies(d)$ succeeds provided that the abducibles $abnorm_1(c)$ and $abnorm_1(d)$ are assumed true. Intuitively, at this point, we could just add these facts to the learned program, or better, try to generate a rule for them and generalize it, as they were considered as new positive examples to be covered for predicate $abnorm_1$. The negative examples for this new concept to be learned correspond to the positive ones (i.e., $abnorm_1(a)$, $abnorm_1(b)$ which correspond to the elements of E^+). The resulting induced rule is (R_3):

$abnorm_1(X) \leftarrow penguin(X).$

At this point, via rules R_2 and R_3 all the remaining positive examples are covered, whereas the negative ones are uncovered. Thus, E^+ becomes empty, and the algorithm ends by producing the following abductive program, where P' is:

$flies(X) \leftarrow superpenguin(X).$
$flies(X) \leftarrow bird(X), not\_abnorm_1(X).$
$abnorm_1(X) \leftarrow penguin(X).$

the set of abducibles is $\mathcal{A}' = \mathcal{A} \cup \{not\_abnorm_1\}$ and the set of integrity constraints is:

$IC' = IC \cup \{\leftarrow not\_abnorm_1(X), abnorm_1(X)\}$

An equivalent program for the same example is obtained in [13], but exploiting negation rather than abduction. However, in [16] the authors have argued that negation *by default* can be seen as a special case of abduction. The power of negation *by default* in induction is preserved and, what is more, is generalized by abduction. Thus, by integrating induction and abduction, we can achieve greater generality with respect to [13]. Nonetheless, the treatment of exceptions here adopted is very similar to that introduced in [13] through a limited form of "classical" negation and priority relations between rules (see also section 5).

3.2 Learning from Integrity Constraints

In the previous example, the background knowledge is very simple since the program P does not contain abducibles and integrity constraints defined by the user. As a further example, let us consider the abductive program $\langle P', A', IC' \rangle$ generated in previous subsection, and add to it the following constraint, I, defined by the user:

$\leftarrow rests(X), plays(X).$

Consider now the new training set:

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

Let *bird*, *penguin* and *superpenguin* (and any abducible predicate) be the bias for *plays* and *rests*. Notice that the added integrity constraint I can be mapped into the following (non-empty) set of negative examples:

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

The inductive algorithm first generates the following rule:[1]

$plays(X) \leftarrow bird(X).$

which covers all the positive examples for *plays*, but arises an inconsistency since the abductive derivation for

$\leftarrow not\_plays(e), not\_plays(f).$

fails. In practice, the generated rule violates the original integrity constraint.

Differently from example of section 3.1, in order to specialize the rule and possibly restore consistency, supposing that the predicate not_abnorm_1 is in the language bias, we can exploit this predicate, and add it to the body of the generated rule:

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

Now the the abductive derivation for

$\leftarrow not\_plays(e), not\_plays(f).$

succeeds provided that both the abducibles $abnorm_1(e)$ and $abnorm_1(f)$ are assumed.[2] This suffices for ruling out the negative examples for *plays*. Therefore, the predicate *plays* holds only for one subclass of the class *bird*, i.e., only for birds which are not penguins.

When iterating, the algorithm also generates a rule for *rests*:

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

In this way, we have increased the power of the learning process. We can learn not only from (positive and negative) ex-

[1] We will analyze the case of *rests* later.

[2] We remove from the sets of abducibles leading to the success of the abductive derivation the (trivial) elements $not\_plays(e)$ and $not\_plays(f)$.

amples but also from integrity constraints, like in [12].

3.3 Learning Integrity Constraints

Let us analyze another case. Suppose we have an empty background knowledge, no abducible and no integrity constraint. Suppose also we have the following positive and negative examples:

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

A standard inductive algorithm cannot infer any general information, since the fact:

$plays(X) \leftarrow .$

covers the negative example $plays(b)$, and the fact:

$rests(X) \leftarrow .$

covers the negative example $rests(a)$. This problem clearly derives from the universal quantification. The facts inferred are too general, and the only way in which they can be specialized is by adding the facts:

$plays(a) \leftarrow .$
$rests(b) \leftarrow .$

which are too specific. Therefore, we can think of an intermediate way of learning from these examples. By induction, our algorithm first generates the rule:

$plays(X) \leftarrow .$

that also covers the negative example $plays(b)$ (the abductive derivation for $\leftarrow not\_plays(b)$ being a failure). In order to restore consistency, a new (abducible) predicate is generated, not_abnorm_1, and added to the body of the rule:

$plays(X) \leftarrow not\_abnorm_1(X).$

Moreover, the following integrity constraint is generated:

$\leftarrow abnorm_1(X), not\_abnorm_1(X).$

Now, the abductive derivation for $\leftarrow not\_plays(b)$ succeeds provided that $abnorm_1(b)$ is assumed. At this point, we could just add this fact to the learned program, or better, try to generate a rule for it and generalize. Differently from example in section 3.1, however, rule generalization would lead to a loop. In fact, by following the proposed algorithm, $abnorm_1(b)$ would be generalized by introducing a new (default) abducible not_abnorm_2, thus leading to a loop. To avoid loop, we prefer to restrict the algorithm and not generate a rule for an abducible predicate containing only abducible predicates in its body. Thus, we prefer to avoid generalization, and define $abnorm_1$ as:

$abnorm_1(b) \leftarrow .$

Then, the algorithm generates a rule for $rests$:

$rests(X) \leftarrow abnorm_1(X).$

and terminates. It is matter of discussion whether it is convenient to generalize the induced integrity constraint or not. By generalizing the integrity constraint:

$\leftarrow abnorm_1(X), not\_abnorm_1(X).$

one obtains the constraint:

$\leftarrow plays(X), rests(X).$

This problem is part of a wider issue that is if to generalize negative examples or not. In fact, since negative examples can be considered as exceptions, it could be worthless to generate rules for exceptions.

In this case, however, the generated integrity constraint is quite meaningful. In fact, even if there is no relation between a and b, and we do not have information on them, we can observe that the property $plays$ and $rests$ are somehow contradictory. In our world, in fact, there is nothing that has both these properties. By induction and generalization of the learned integrity constraint, we can infer the integrity constraint which asserts that it is not possible that an object X in our world both $plays$ and $rests$.

4 Discussion

In this section, we consider some issues that are still matter of discussion.

First, we point out that the properties of soundness and completeness of inductive logic programs should be carefully evaluated in the context of abductive programs. In fact, with respect to completeness, while in ILP we can test each positive example separately and be sure that their conjunction is true, in our approach this is no longer true. More formally, if

$P \models e_1^+ \ \wedge \ldots \wedge \ P \models e_n^+ \ \not\Rightarrow \ P \models e_1^+ \wedge \ldots \wedge e_n^+$

For instance, suppose we have the following program with a background knowledge containing only the two integrity constraints:

$\leftarrow abd(1), abd(2).$
$\leftarrow abd(3).$

and P and $\mathcal{A}$ be empty. Let the training set be:

$E^+ = \{p(1), p(2)\}$
$E^- = \{p(3)\}$

Suppose also the bias contains only the abducible abd. If we generate the rule:

$p(X) \leftarrow abd(X).$

the query $\leftarrow p(1), p(2)$. fails even if $p(1)$ and $p(2)$ are separately covered by the program.

Therefore, the generated program should cover the conjunction of positive examples and of the negation of negative ones (see section 2). In fact, the set of abduced literals must be consistent for all the examples. This means that all the positive examples are true (and the negative false) in the same model. If we relax this assumption, we have a semantics in which each positive example must be true (each negative false) in at least one feasible model of the program. Consequently, different positive examples can be true (and negative false) in different models. This second approach is obviously simpler from a computational point of view, but less interesting in many cases.

In the following, we concentrate on the first approach and we point out some implementation issues. In order to check the completeness of the program generated so far, we have to test, the conjunction of positive examples, as sketched in section 2.

$\leftarrow e_1^+, e_2^+, \ldots, e_n^+.$

If the conjunction succeeds, the algorithm terminates. Otherwise, a new rule is generated accordingly. In practice, if the conjunction fails in covering one of the positive example, we have no information on which positive example has led to a failure. On the other hand, if we test each positive example separately by means of an abductive derivation, we are not ensured to find a unique model that covers all the examples. A solution to this problem is to test each positive example separately, but in a sort of pipeline, by taking into account the set of abduced literals generated in the previous step.

In the algorithm, we have to replace the step *Remove from* E^+ *the positive examples covered by* C with the following steps:

- Initialize $\Delta_0 = \Delta_{neg}$ where Δ_{neg} is a set of predicates abduced during the test of negative examples, as shown in the following ;
- For each $e_i^+ \in E^+$ an abductive derivation is started with an initial set of abducibles Δ_{i-1} where Δ_{i-1} is the set of literals abduced in the previous step;
- If the abductive derivation for e_i^+ succeeds, the positive example is removed from E^+.[3] Otherwise, the positive example is still in E^+, and we impose $\Delta_i = \Delta_{i-1}$.

Analogous steps must be performed in order to guarantee the soundness of the generated program. After the specialization of each rule we have to test the set of negative examples via an abductive derivation for:

$\leftarrow not\_e_1^-, not\_e_2^-, \ldots, not\_e_m^-$

We check this conjunction with the initial set of abducibles resulting from the test of positive examples (the first time, this set is empty). The resulting set of abducibles Δ_{neg} is used as positive examples for the induction of a new rule, as shown in section 3.1.

Another open issue concerns the possibility of learning integrity constraints. In section 3.3, we have shown how to learn binary integrity constraints. With the current algorithm, we are not able to learn general constraints. If we want to learn general constraints, we have to change the algorithm proposed. For instance, suppose we have the following set of positive and negative examples:

$E^+ = \{plays(a), plays(b), rests(b), rests(c), eats(a), eats(c)\}$

$E^- = \{plays(c), rests(a), eats(b)\}$

The bias is abd_1, abd_2 and abd_3. We want to obtain the following constraint

$\leftarrow abd_1(X), abd_2(X), abd_3(X).$

which can be generalized by means of the following rules:

$plays(X) \leftarrow abd_1(X).$

$rests(X) \leftarrow abd_2(X).$

$eats(X) \leftarrow abd_3(X).$

as:

$\leftarrow rests(X), plays(X), eats(X).$

We are currently investigating how to provide restrictions on the language bias in order to obtain a specialization of the integrity constraint instead of the specialization of the generated rules.

5 Related Work

The relationship between abduction and learning has been studied recently by several authors, mainly in the field of Artificial Intelligence (see, for instance, [7, 24]). 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.

Our work is an attempt to bring closer the fields of Abductive and Inductive Logic Programming. In particular, it addresses two issues:

- How to learn concepts in a possibly incomplete framework, where some (undefined) predicates can be assumed, provided that their assumption is consistent with integrity constraints;
- How to treat exceptions to induced rules by means of (user defined) abducibles, when possible, or by introducing new (default) abducibles and then synthesizing a rule for their complement literal.

In the following we compare our work with other related work within the Logic Programming area. To the best of our knowledge, related literature has been focussed on the two issues above separately, and this is the first, preliminary, attempt to integrate both.

As concern the integration of abduction and induction, a notable work is that by Dimopoulos and Kakas [14]. In this paper, the authors suggest a methodology for the integration of abduction in learning, where abduction is used first 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. As in our framework, the main advantage of the integration in [14] is that it allows us to possibly learn more accurate rules in presence of missing information, and later classify new examples that may be incompletely described. Also the framework in [14] starts from an abductive logic program $\langle P, A, IC\rangle$ and a set of positive and negative observations for a concept c, and synthesize a new abductive theory $\langle P', A, IC'\rangle$, where P' contains rules for the concept c. Minimality in finding the abducibles supporting an observation ensures that the inductive algorithm uses abducibles only when it is really needed. Differently from us, no new abducible is introduced when rule specialization is needed in order to rule out negative observations. Rather, they prefer to add new integrity constraints in order to partly specialize rules. We intertwine, instead, the introduction of new abducibles with the generation of integrity constraints (and rules) for their complement in order to block (some instances of) a rule. We generate new integrity constraints that would exclude abductive coverage of the negative examples.

In [2] 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 framework of [2] is based on Bry's work for intensional knowledge base updating [5]. To cope with program updating, a set of logical formulas is defined for a meta-predicate *new*. A set of rewrite rules translates the predicate definition into operations on the object level program. More in detail, given a logic program P and an evidence E, meta-predicate *new* expresses statements that describe the desired (updated) logic program P_{new}. This meta-predicate is defined (for both positive and negative atoms) in terms of operations on the object program facts and clauses. The definition for *new* is then extended with respect to [5] with additional rules in order to cope with abduction and induction. To cope with abduction, when a positive abducible atom is encountered, it is simply added to the program. When a negative abducible is encountered, it is properly constrained in order to not unify with any of the abduced atoms. To cope with induction, when a positive *inducible* atom (i.e., an atom corresponding to a concept to be learned) is encountered, first it is checked if it is already covered by program clauses; otherwise, a new clause is constructed that covers the atom and that is consistent with the negative evidence seen so far. When a negative inducible is encountered, instead, if it is covered by program clauses then clauses are refined, i.e., made more specific. In this way, the

[3] Note that this represents a choice point for the abductive derivation which has to be considered during backtracking.

two techniques are mixed, as in our framework, in order to address the problem of completing incomplete knowledge.

In [12] the authors present a method to automatically modify a knowledge base when violating a newly supplied integrity constraint. The method uses inductive learning techniques and relies on an oracle. The proposed method extends Shapiro's method and MIS system [25]. Even if with a different purpose and approach (i.e., knowledge updating), the work in [12] shows (as we do in our framework) that integrity constraints form a very useful way of generalization of examples in concept- learning.

Another related work is reported in [1], where the authors present a system, called RUTH, for theory revision based on ILP. RUTH is able to cope with definite, functor-free clauses, and (as the system described in [12]) integrates intensional database updating with incremental concept-learning. Apart from adding and deleting clauses and facts, in [1] the authors also employ an abductive operator which allows RUTH to introduce missing factual knowledge into the knowledge base. Added (and possibly violated) integrity constraints correspond to positive (uncovered) and negative (covered) examples. In order to handle (uncovered) positive examples, theory revision recovers from the arisen inconsistency by either (*i*) adding an example as a fact in the database, or (*ii*) building a maximally general clause that covers the example, or (*iii*) abducing one or more new facts. In order to handle (covered) negative examples, theory revision recovers from the arisen inconsistency by deleting one of the clauses that contribute to the SLDNF proof of the example. As [1] (which adopts standard SLDNF), we do not rely on any oracle, but rather on abductive proof procedure for determining the proof of an atom. Furthermore, both RUTH and our framework can treat as abducibles *some* of the program predicates. Differently from [1], we avoid clause retraction, and rather prefer to add integrity constraints in order to rule out negative examples. In this respect, we do not fully support theory revision.

As concerns the treatment of exception to induced rules, in [3], 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 non-monotonic knowledge representation. Several other authors have also addressed this problem, in the context of Logic Programming, by allowing for exceptions to (possibly induced) rules [13, 11]. In these frameworks, non-monotonicity and exceptions are dealt with by learning logic programs with negation. Our approach in the treatment of exceptions is very related to the work by Dimopoulos and Kakas [13]. They propose a framework for learning where the language is extended from definite to non-monotonic logic programs. They rely on a language which uses a limited form of "classical" negation together with a priority relations between the sentences of the program [21]. Nonetheless, a program written in this language can be easily translated in a normal logic program (i.e., in a logic program where *default* literals possibly occur in rule bodies), by introducing new predicates and capturing the priority between rules into the notion of abnormality. In this respect, when the starting background knowledge is a (definite) logic program, their and our approach coincide. On the other hand, in [16] the authors have argued that negation *by default* can be seen as a special case of abduction. Thus, in our framework, relying on ALP [18], we can achieve greater generality than [13]. In their paper, they also prove 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, this is not true since we generate clauses which possibly cover only a subset of the given input examples, and loop may arise (even if we are able to avoid loop situations as discussed in section 3.3).

6 Conclusions and Future Work

We have discussed how it is possible to learn an abductive logic program, addressing in this way non-monotonicity and exceptions. In the devised framework, abducibles and integrity constraints can be specified by the user as content of the background knowledge, and are also generated by the learning process. In this way, we increase the expressive power of the background knowledge and ameliorate the learning process.

We have grounded our framework on the generalized stable model semantics defined for abductive logic programs by Kakas and Mancarella [19]. There are cases, however, where no generalized stable model exists for an abductive logic program. This can occur, in general, because of the particular background knowledge or because of generated clauses and integrity constraints. In this case, a three-valued rather than a total semantics could be adopted as, for instance, the abductive semantics defined in [6]. The study of which class of programs is generated by the abductive-inductive algorithm here introduced and its properties is scope for future work.

In this respect, matter of investigation is the soundness and completeness of our system. That means, if the system produces an abductive logic program, it will be consistent (i.e., have a meaning with respect to the adopted underlying semantics). And if there exists a solution (namely an abductive logic program), the system will find one by using induction and abduction.

At the time being, this is a very preliminary paper. The basic algorithm is just sketched, and must be better formalized, and few examples have been considered till now. A number of issues are not considered in the paper, but they are subject for future work. First of all, the use and meaning of integrity constraints have to be better clarified. In ALP, integrity constraints can be defined by the user in order to constrain the generated explanations. In ILP, integrity constraints are usually implicit. Nonetheless, there is an intimate bond between the use of integrity constraints in both systems. In the paper, we have shown that ALP integrity constraints can be interpreted as negative examples for ILP, in accordance with [12]. How to learn integrity constraints and how and when generalize them is matter of discussion.

Another crucial notion present both in abductive and inductive reasoning is the minimality (respectively generality) of the explanation given (respectively, of rules inferred). A basis for discussion is [8] where the authors relate different notions of minimality applied in abductive reasoning with different kinds of generality achieved in inductive reasoning.

REFERENCES

[1] H. Adé, and M. Denecker. RUTH: An ILP Theory Revision System. In *Proceedings ISMIS94*, 1994.

[2] H. Adé, and M. Denecker. AILP: Abductive Inductive Logic Programming. In *Proceedings IJCAI95*, pp. 1201–1207, 1995.

[3] M. Bain, and S. Muggleton. Non-Monotonic Learning. In*Inductive Logic programming*, S. Muggleton (Ed.), Chapter 7, pp. 145–161, Academic Press, The A.P.I.C. Series No 38, 1992.

[4] F. Bergadano, and D. Gunetti. Inductive Logic Programming. The MIT Press, 1996.

[5] F. Bry. Intensional Updates: Abduction via Deduction. In*Proc. 7th Int. Conference on Logic Programming ICLP90.* pp. 561–578, MIT Press, 1990.

[6] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. An Abductive Framework for Extended Logic Programming. In*Proc. 3rd Int. Workshop on Logic Programming and Non Monotonic Reasoning.* LNAI, Vol. 928, pages 330–343, Springer-Verlag, 1995.

[7] W. W. Cohen. Abductive Explanation-Based Learning: A Solution to the Multiple Inconsistent Explanation Problem. *Machine Learning*, Vol. 8, pp. 167–219, 1992.

[8] L. Console, and L. Saitta. Generalization and explanation in machine learning. Submitted.

[9] L. Console, D. Theseider Duprè, and P. Torasso. Abductive Reasoning Through Direct Deduction from Completed Domains Models. In *Methodologies for Intelligent Systems*, Vol. 4, page 175, North Holland,1989.

[10] P.T. Cox, and T. Pietrzykowski. Causes for events: Their computation and applications. In *Proceedings CADE-86*, page 608, 1986.

[11] L. De Raedt, and M. Bruynooghe. On negation and three-valued logic in interactive concept learning. In *Proc. of the 9th European Conference on AI - ECAI90*, pp. 207–212. Pitman Publishing, 1990.

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

[13] Y. Dimopoulos, and A. Kakas. Learning Non-Monotonic Logic Programs: Learning Exceptions. In *Machine Learning: ECML-95*, Proceedings of the 8th European Conference on Machine Learning, N. Lavrac and S. Wrobel (Eds.), Lecture Notes in Artificial Intelligence No. 912, pp. 122–137. Springer-Verlag, 1995.

[14] Y. Dimopoulos, and A. Kakas. Abduction and Learning. In *Advances in Inductive Logic Programming*, L. de Raedt (Eds.), IOS Press, 1996.

[15] P.M. Dung. Negation as Hypothesis: An Abductive Foundation for Logic Programming", In *Proc. 8th Int. Conf. on Logic Programming ICLP91*, MIT Press, 1991.

[16] K. Eshghi, and R.A. Kowalski. Abduction Compared with Negation by Failure. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conf. on Logic Programming ICLP89*, pp. 234–254. The MIT Press, 1989.

[17] F. Esposito, D. Malerba, and G. Semeraro. Multistrategy Learning for Document Recognition. *Applied Artificial Intelligence*, 8:33–84, 1994.

[18] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[19] A.C. Kakas, and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proceedings of 9th European Conference on Artificial Intelligence ECAI90*, pp. 385–391. Pitman Publishing, 1990.

[20] A.C. Kakas, and P. Mancarella. On the relation between Truth Maintenance and Abduction. In *Proceedings PRICAI90*, 1990.

[21] A.C. Kakas, P. Mancarella, and P.M. Dung. The acceptability semantics for logic programs. In *Proceedings of 11th International Conference on Logic Programming ICLP94*, pp. 504–519. The MIT Press, 1994.

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

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

[24] P. O' Rourke. Abduction and Explanation-Based Learning: Case Studies in Diverse Domains. *Computational Intelligence*, Vol. 10, pp. 295–330, 1994.

[25] E. Shapiro. Algorithmic Program Debugging. MIT Press, Cambridge, MA, 1983.

Appendix

In the following we recall the abductive and consistency derivation used by our algorithm, which are taken from [20].

Abductive derivation

An abductive derivation from $(G_1\ \Delta_1)$ to $(G_n\ \Delta_n)$ in the abductive program $\langle P, Ab, IC\rangle$ 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:

(1) 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;

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

(3) 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 $G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and $\Delta_{i+1} = \Delta'$.

Steps (1) and (2) are SLD-resolution steps with the rules of P and abductive or default hypotheses, respectively. In step (3) 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

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

where :

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

(ii) for each $i > 1$, F_i has the form $\{\leftarrow L_1, \ldots, L_k\} \cup F'_i$ and for some $j = 1, \ldots, k$ $(F_{i+1}\ \Delta_{i+1})$ 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 $\leftarrow L_1, \ldots, L_k$ 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 $F_{i+1} = \{\leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k\} \cup F'_i$ 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 $\leftarrow L_1, \ldots, L_k$ 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 (atomic, for the sake of simplicity), the procedure of [20] 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 with answer Δ.