Learning by Answer Sets

Chiaki Sakama
Department of Computer and Communication Sciences
Wakayama University
Sakaedani, Wakayama 640 8510, Japan
sakama@sys.wakayama-u.ac.jp *http://www.sys.wakayama-u.ac.jp/˜sakama*

Abstract

This paper presents a novel application of answer set programming to concept learning in nonmonotonic logic programs. Given an extended logic program as a background theory, we introduce techniques for inducing new rules using answer sets of the program. The produced new rules explain positive/negative examples in the context of inductive logic programming. The result of this paper combines techniques of two important fields of logic programming in the context of nonmonotonic inductive logic programming.

Introduction

Nonmonotonic logic programming (NMLP) introduces mechanisms of representing incomplete knowledge and reasoning with commonsense. An example of such extensions is *extended logic programs* with the *answer set semantics* (Gelfond and Lifschitz 1991). On the other hand, *inductive logic programming* (ILP) (Muggleton 1992) realizes inductive learning in logic programming. It is concerned with learning a general theory from examples and background knowledge.

NMLP realizes commonsense reasoning in logic programming, but a program never derives information that is not specified in the program. By contrast, ILP constructs new rules from given examples, but the present ILP mostly considers Horn logic programs and has limited applications to nonmonotonic theories. Thus, both NMLP and ILP have limitations in their present frameworks and complement each other. Since both commonsense reasoning and machine learning are indispensable for constructing powerful knowledge systems, combining techniques of the two fields in the context of *nonmonotonic inductive logic programming* (NMILP) is important. Such combination will extend the representation language on the ILP side, while it will introduce a learning mechanism to programs on the NMLP side.

This position paper presents techniques of realizing inductive learning in nonmonotonic logic programs. We consider an extended logic program as a background theory, and introduce a method of learning new rules using answer sets of the program. The produced new rules explain positive/negative examples in the context of inductive logic programming. From the viewpoint of answer set programming (ASP), it provides a novel application of ASP to concept learning in nonmonotonic logic programs.

Preliminaries

A *program* considered in this paper is an *extended logic program* (ELP) (Gelfond and Lifschitz 1991), which is a set of *rules* of the form:

$$L_0 \leftarrow L_1,\, \ldots,\, L_m,\, not\, L_{m+1},\, \ldots,\, not\, L_n \qquad (1)$$

where each L_i is a literal and not is negation as failure (NAF). The literal L_0 is the *head* and the conjunction $L_1,\, \ldots,\, L_m,\, not\, L_{m+1},\, \ldots,\, not\, L_n$ is the *body*. The conjunction in the body is identified with the set of conjuncts. For a rule R, $head(R)$ and $body(R)$ denote the head and the body of R, respectively. The head is possibly empty and a rule with the empty head is called an *integrity constraint*. A rule with the empty body $L \leftarrow$ is identified with the literal L and is called a *fact*. A program (rule, literal) is *ground* if it contains no variable. Any rule with variables is considered as a shorthand of its ground instances. A program or a rule is *NAF-free* if it contains no not (i.e., $m = n$ for the rule (1)).

Let Lit be the set of all ground literals in the language of a program. Any element in $Lit^+ = Lit \cup \{\, not\, L \mid L \in Lit \,\}$ is called an *LP-literal* and an LP-literal $not\, L$ is called an *NAF-literal*. When K is an LP-literal, it is defined $|K| = K$ if K is a literal; and $|K| = L$ if $K = not\, L$. For an LP-literal L, $pred(L)$ denotes the predicate of L and $const(L)$ denotes the set of constants appearing in L. A set $S (\subseteq Lit)$ *satisfies* the conjunction $C = (L_1,\, \ldots,\, L_m,\, not\, L_{m+1},\, \ldots,\, not\, L_n)$ (written as $S \models C$) if $\{L_1, \ldots, L_m\} \subseteq S$ and $\{\, L_{m+1}, \ldots, L_n \,\} \cap S = \emptyset$. S satisfies a ground rule R (written as $S \models R$) if $S \models body(R)$ implies $S \models head(R)$. In particular, S satisfies the ground integrity constraint

$$\leftarrow L_1,\, \ldots,\, L_m,\, not\, L_{m+1}, \ldots,\, not\, L_n$$

if $\{L_1, \ldots, L_m\} \not\subseteq S$ or $\{\, L_{m+1}, \ldots, L_n \,\} \cap S \neq \emptyset$.

The semantics of ELPs is given by the *answer set semantics* (Gelfond and Lifschitz 1991). The *answer sets* of a program are defined by the following two steps. First, let P be an NAF-free program and $S \subset Lit$. Then, S is a *consistent* *answer set* of P if S is a minimal set which satisfies every ground rule from P and does not contain both L and $\neg L$ for any $L \in Lit$. Next, let P be any program and $S \subset Lit$. Then, define the NAF-free program P^S as follows: a rule $L_0 \leftarrow L_1, \ldots, L_m$ is in P^S iff there is a ground rule of the form (1) from P such that $\{ L_{m+1}, \ldots, L_n \} \cap S = \emptyset$. Then, S is a *consistent answer set* of P if S is a consistent answer set of P^S. A consistent answer set is simply called an answer set hereafter. For an answer set S, we define

$$S^+ = S \cup \{ not\, L \mid L \in Lit \setminus S \}.$$

A program P is *consistent* if it has a (consistent) answer set; otherwise P is *inconsistent*. Throughout the paper, a program is assumed to be consistent unless stated otherwise. A program P is called *categorical* if it has a unique consistent answer set (Baral and Gelfond 1994). If a rule R (resp. a conjunction C) is satisfied in every answer set of P, it is written as $P \models R$ (resp. $P \models C$); otherwise $P \not\models R$ (resp. $P \not\models C$). In particular, $P \models L$ if a literal L is included in every answer set of P; otherwise $P \not\models L$.

Learning from Positive Examples

A typical induction problem is as follows. Given a background program P and a ground literal L which represents a *(positive) example*, construct a rule R satisfying

$$P \cup \{R\} \models L \tag{2}$$

and $P \cup \{R\}$ is consistent. Here, it is assumed that

$$P \not\models L\,, \tag{3}$$

since if $P \models L$ there is no need to introduce R. We also assume that P, R, and L have the same language.

The problem is how to compute R efficiently. We first present a couple of propositions which are used later.

Proposition 1 *Let P be a program, R a rule, and L a ground literal. Suppose that $P \cup \{R\}$ is consistent and $P \cup \{R\} \models L$. If $P \models R$, then $P \models L$.*

Proof: Suppose that $P \models R$ and $P \not\models L$. Then, there is an answer set S of P such that $L \notin S$. By $P \models R$, $S \models R$. If $\{ L_1, \ldots, L_m \} \not\subseteq S$ or $\{ L_{m+1}, \ldots, L_n \} \cap S \neq \emptyset$ for any ground instance R' : $L_0 \leftarrow L_1, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n$ of R, then S does not satisfy the bodies of those instances. Then S is an answer set of $P^S \cup \{R\}^S$, hence an answer set of $P \cup \{R\}$. As $L \notin S$, this contradicts the assumption $P \cup \{R\} \models L$. Else if $\{ L_1, \ldots, L_m \} \subseteq S$ and $\{ L_{m+1}, \ldots, L_n \} \cap S = \emptyset$ for some ground instance R' of R, $L_0 \leftarrow L_1, \ldots, L_m$ is in $\{R\}^S$. By $S \models R$, $L_0 \in S$. So S is an answer set of $P^S \cup \{R\}^S$, thereby an answer set of $P \cup \{R\}$. Again, this contradicts the assumption $P \cup \{R\} \models L$. □

Proposition 2 *Let P be a program and L a ground literal such that $pred(L)$ does not appear in P. If there is a rule R such that $P \cup \{R\}$ is consistent and there is a ground instance $R\theta$ of R with some substitution θ such that $P \cup \{ R\theta \} \models L$, then $P \cup \{R\} \models L$.*

Proof: Let $R = \alpha(x) \leftarrow \Gamma(x)$ and $R\theta = \alpha(t) \leftarrow \Gamma(t)$, where $\theta = \{ x/t \}$, $\alpha(x)$ is either $p(x)$ or $\neg p(x)$ for some predicate p and $\Gamma(x)$ is a conjunction of LP-literals. Since P does not contain the predicate of L, $P \cup \{ R\theta \} \models L$ implies $L = \alpha(t)$. On the other hand, $P \cup \{ \alpha(t) \leftarrow \Gamma(t) \} \models \alpha(t)$ implies $P \models \Gamma(t)$. Suppose that $P \cup \{R\} \not\models \Gamma(t)$. Then, there is an answer set S of $P \cup \{R\}$ such that $S \not\models \Gamma(t)$. As $P \cup \{ R\theta \} \models \Gamma(t)$, an introduction of some instance of R makes $\Gamma(t)$ unsatisfied. Let $\alpha(s) \leftarrow \Gamma(s)$ be a ground instance of R such that $S \models \Gamma(s)$, $S \models \alpha(s)$, and $\Gamma(t)$ contains $not\, \alpha(s)$. Then, $\Gamma(s)$ also contains $not\, \alpha(s)$, and the introduction of $\alpha(s) \leftarrow \Gamma(s)$ makes $P \cup \{R\}$ inconsistent. This contradicts the assumption that $P \cup \{R\}$ is consistent. □

By Proposition 1, $P \cup \{R\} \models L$ and $P \not\models L$ imply

$$P \not\models R. \tag{4}$$

The relation (4) is a necessary condition for the induction problem satisfying (2) and (3). By Proposition 2, on the other hand, if an example L has a predicate which does not appear in P and there is a rule R such that $P \cup \{R\}$ is consistent, then finding a ground instance $R\theta$ such that $P \cup \{ R\theta \} \models L$ leads to the construction of R satisfying (2). This is a sufficient condition for R.

In Proposition 2, if the predicate of L appears in P, the result does not hold in general.

Example 1 Consider the program P and the example L such that

$$\begin{array}{ll} P: & q(a) \leftarrow p(b). \\ L: & p(a). \end{array}$$

Then, for the rule $R = p(x) \leftarrow not\, q(x)$ and the substitution $\theta = \{ x/a \}$, $P \cup \{ R\theta \} \models L$ holds. But $P \cup \{ R \} \not\models L$ because $P \cup \{ R \}$ has the answer set $\{ p(b),\ q(a) \}$.

In many induction problems an example L is a newly observed evidence such that the background program P contains no information on $pred(L)$. When P contains a rule with $pred(L)$ in its head, the problem of computing hypotheses satisfying (2) is usually solved using *abduction*. On the other hand, the condition of Proposition 2 is relaxed when a program P and a rule R are NAF-free.

Proposition 3 *Let P be an NAF-free program and L a ground literal. If there is an NAF-free rule R such that $P \cup \{R\}$ is consistent and there is a ground instance $R\theta$ of R with some substitution θ such that $P \cup \{ R\theta \} \models L$, then $P \cup \{R\} \models L$.*

Proof: A consistent NAF-free program has a single answer set. Suppose that P_1 and P_2 are consistent NAF-free programs which have the answer sets S_1 and S_2, respectively. Then, $P_1 \subseteq P_2$ implies $S_1 \subseteq S_2$. Since $P \cup \{R\}$ is consistent and $P \cup \{ R\theta \} \subseteq P \cup \{ R \}$, the answer set of $P \cup \{ R\theta \}$ is a subset of the answer set of $P \cup \{ R \}$. Hence, the result holds. □

Proposition 3 is useful for ILP problems containing no NAF.

We use these necessary and sufficient conditions to construct rules in induction problems. To simplify the problem, in what follows we consider a program P which is *function-free* and *categorical*. Given two ground LP-literals L_1 and L_2, we define the relation $L_1 \sim L_2$ if $pred(L_1) = pred(L_2)$ and $const(L_1) = const(L_2)$. Let L_1 and L_2 be two ground LP-literals such that each literal has a predicate of arity ≥ 1. Then, L_1 in a ground rule R is *relevant* to L_2 if either (i) $L_1 \sim L_2$ or (ii) L_1 shares a constant with an LP-literal L_3 in R such that L_3 is relevant to L_2. On the other hand, given a program P and an example L, a ground LP-literal K is *involved* in $P \cup \{L\}$ if $|K|$ appears in the ground instance of $P \cup \{L\}$.

Suppose that a program P has the unique answer set S. By (4) the following relation holds.

$$S \not\models R. \tag{5}$$

Then, we start to find a rule R satisfying the condition (5). Consider the integrity constraint $\leftarrow \Gamma$ where Γ consists of ground LP-literals in S^+ such that every element in Γ is relevant to the example L, and is also involved in $P \cup \{L\}$. Since S does not satisfy this integrity constraint,

$$S \not\models\ \leftarrow \Gamma \tag{6}$$

holds. That is, $\leftarrow \Gamma$ is a rule which satisfies the condition (5).

Next, it holds that $P \not\models L$ for the example L by (3). Then, $S \not\models L$, so $not\, L$ is in S^+. Since $not\, L$ is relevant to L, the integrity constraint $\leftarrow \Gamma$ contains $not\, L$ in its body. We shift the literal L to the head and produce

$$L \leftarrow \Gamma' \tag{7}$$

where $\Gamma' = \Gamma \setminus \{not\, L\}$.

Finally, we generalize the rule (7) by constructing a rule R^* such that $R^*\theta = L \leftarrow \Gamma'$ for some substitution θ.

The next theorem presents that the rule R^* satisfies the condition (4).

Theorem 4 *Let P be a categorical program and R^* a rule obtained as above. Then, $P \not\models R^*$.*

Proof: Suppose a rule $L \leftarrow \Gamma'$ of (7). As $\Gamma' \subseteq S^+$ and $L \notin S$, $S \not\models L \leftarrow \Gamma'$. Thus, S does not satisfy a ground instance (7) of R^*. Hence $S \not\models R^*$, thereby $P \not\models R^*$. □

The next theorem presents a sufficient condition of R^* to satisfy the relation (2).

Theorem 5 *Let P be a categorical program, L a ground literal, and R^* a rule obtained as above. If $P \cup \{R^*\}$ is consistent and $pred(L)$ does not appear in P, then $P \cup \{R^*\} \models L$.*

Proof: Let $R^*\theta = L \leftarrow \Gamma'$ be the rule of (7). As $P \cup \{R^*\}$ is consistent, $P \cup \{\, R^*\theta \,\}$ is consistent. For the answer set S of P, $S \models \Gamma'$. Since $pred(L)$ does not appear in P, $S \cup \{\, L \,\}$ becomes the answer set of $P \cup \{\, R^*\theta \,\}$. Hence, $P \cup \{\, R^*\theta \,\} \models L$, and the result holds by Proposition 2. □

When $P \cup \{R^*\}$ is inconsistent, construct a rule R^{**} by dropping some LP-literals from the body of $\{R^*\}$. If $P \cup \{R^{**}\}$ is consistent and $pred(L)$ does not appear in P, $P \cup \{R^{**}\} \models L$ holds by Theorem 5.

Example 2 Let P be the program

$$bird(x) \leftarrow penguin(x),$$
$$bird(tweety) \leftarrow,$$
$$penguin(polly) \leftarrow .$$

Given the example $L = flies(tweety)$, it holds that $P \not\models flies(tweety)$. Our goal is then to construct a rule R satisfying $P \cup \{R\} \models L$.

First, the set S^+ of LP-literals becomes

$$\begin{aligned} S^+ \ = \ \{\, & bird(tweety),\ bird(polly),\ penguin(polly),\\ & not\, penguin(tweety),\ not\, flies(tweety),\\ & not\, flies(polly),\ not\, \neg bird(tweety),\\ & not\, \neg bird(polly),\ not\, \neg penguin(polly),\\ & not\, \neg penguin(tweety),\ not\, \neg flies(tweety),\\ & not\, \neg flies(polly)\,\}. \end{aligned}$$

From S^+ picking up LP-literals which are relevant to L and are involved in $P \cup \{L\}$, the integrity constraint:

$$\leftarrow bird(tweety),\ not\, penguin(tweety),\ not\, flies(tweety)$$

is constructed. Next, shifting $flies(tweety)$ to the head produces

$$flies(tweety) \leftarrow bird(tweety),\ not\, penguin(tweety)\,.$$

Finally, replacing $tweety$ by a variable x, the rule

$$R^* :\ flies(x) \leftarrow bird(x),\ not\, penguin(x)$$

is obtained, where $P \cup \{R^*\} \models L$ holds.

Learning from Negative Examples

In induction problems, *negative examples* are considered as well as positive ones. In contrast to positive examples, negative examples are facts that should not be entailed. For negative examples, an induction problem is stated as follows. Given a program P and a ground literal L which represents a negative example, construct a rule R satisfying

$$P \cup \{R\} \not\models L \tag{8}$$

and $P \cup \{R\}$ is consistent. Here, it is assumed that

$$P \models L\,, \tag{9}$$

since if $P \not\models L$ there is no need to introduce R.

As the case of positive examples, we first introduce necessary and sufficient conditions for computing R.

Proposition 6 *Let P be a program, R a rule, and L a ground literal. Suppose that $P \cup \{R\}$ is consistent and $P \cup \{R\} \not\models L$. If $P \models R$, then $P \not\models L$.*

Proof: Similar to the proof of Proposition 1. □

The *dependency graph* of a program P is a directed graph such that nodes are predicates in P and there is a *positive edge* (resp. *negative edge*) from p_1 to p_2 if there is a rule R in P such that $head(R)$ contains the predicate p_1 and $body(R)$ contains a literal L (resp. $not\, L$) such that $pred(L) = p_2$. We say that p_1 *depends* on p_2 (in P) if there is a path from p_1 to p_2. On the other hand, p_1 *strongly* depends on p_2 if for every path containing a node p_1, p_1 depends on p_2. Also, p_1 *negatively* depends on p_2 if any path from p_1 to p_2 contains an odd number of negative edges.

Proposition 7 *Let P be a program and L a ground literal. Suppose that there is a rule R such that $P \cup \{R\}$ is consistent and there is a ground instance $R\theta$ of R with some substitution θ such that $P \cup \{\, R\theta \,\} \not\models L$. If $pred(L)$ strongly and negatively depends on the predicate of $head(R\theta)$ in P, then $P \cup \{R\} \not\models L$.*

Proof: By $P \cup \{\, R\theta \,\} \not\models L$, L is not included in some answer set of $P \cup \{\, R\theta \,\}$. Since $pred(L)$ strongly and negatively depends on the predicate of $head(R)$ in P, the introduction of an instance $R\eta$ to $P \cup \{\, R\theta \,\}$ for any substitution η does not make L true in any answer set of $P \cup \{R\}$. Hence, $P \cup \{R\} \not\models L$ holds. □

Example 3 Consider the program P and the negative example L such that

$$\begin{aligned} P: \quad & p(x) \leftarrow q(x),\ not\, r(x), \\ & q(a) \leftarrow, \\ & s(a) \leftarrow . \\ L: \quad & p(a). \end{aligned}$$

Since p strongly and negatively depends on r in P, $P \cup \{\, r(a) \leftarrow s(a) \,\} \not\models L$ implies $P \cup \{\, r(x) \leftarrow s(x) \,\} \not\models L$.

By Proposition 6 the relation

$$P \not\models R$$

becomes a necessary condition for the problem satisfying (8) and (9). Then, we construct a hypothetical rule in a similar manner to the case of positive examples. We again assume function-free and categorical programs hereafter.

Suppose that a program P has the unique answer set S. Then, the relation

$$S \not\models R$$

holds. The integrity constraint $\leftarrow \Gamma$ is constructed for $\Gamma \subseteq S^+$ of ground LP-literals which are relevant to the negative example L and are involved in $P \cup \{L\}$. Then, the relation

$$S \not\models\ \leftarrow \Gamma$$

holds. On the other hand, to construct an objective rule from the integrity constraint $\leftarrow \Gamma$, we shift a literal which has a *target predicate*. Here, a target predicate is a pre-specified predicate which is subject to learn. In case of positive examples, we identified the target predicate with the one appearing in the positive example. This is because the purpose of learning from positive examples is to construct a new rule which entails the positive example. In case of negative examples, on the other hand, the negative example L is already entailed from the program P ($P \models L$). The purpose is then to block the entailment of L by introducing some rule R to P. In this situation, the rule R does not have the predicate of L in its head in general. So we distinguish the target predicate from the one appearing in L. We select a target predicate from predicates in P on which $pred(L)$ strongly and negatively depends. Thus, if Γ contains an LP-literal $not\, K$ which contains the target predicate satisfying this condition, we construct

$$K \leftarrow \Gamma' \tag{10}$$

from $\leftarrow \Gamma$ where $\Gamma' = \Gamma \setminus \{\, not\, K \,\}$. Note that Γ may contain no $not\, K$ with the target predicate. On the other hand, if the target predicate occurs in more than one NAF-literal in Γ, the rule (10) is constructed by shifting one of these literals. As a result, there are none, one, or several rules of the form (10) which are constructed from $\leftarrow \Gamma$.

Finally, we generalize the rule (10) to R^* such that $R^*\theta = K \leftarrow \Gamma'$ by replacing constants with appropriate variables. Then the following result holds.

Theorem 8 *Let P be a categorical program and R^* a rule obtained as above. Then, $P \not\models R^*$.*

Proof: Similar to the proof of Theorem 4. □

By contrast, whether the relation $P \cup \{R^*\} \not\models L$ holds or not depends on the existence of an appropriate target predicate.

Theorem 9 *Let P be a categorical program, L a ground literal, and R^* a rule obtained as above. If $P \cup \{R^*\}$ is consistent and $P \cup \{\, R^*\theta \,\} \not\models L$ for some substitution θ, then $P \cup \{R^*\} \not\models L$.*

Proof: Since $pred(L)$ strongly and negatively depends on the predicate of R^* in P, the result holds by Proposition 7. □

Example 4 Let P be the program

$$\begin{aligned} & flies(x) \leftarrow bird(x),\ not\, ab(x), \\ & bird(x) \leftarrow penguin(x), \\ & bird(tweety) \leftarrow, \\ & penguin(polly) \leftarrow . \end{aligned}$$

Given the negative example $L = flies(polly)$, it holds that $P \models flies(polly)$. Our goal is then to construct a rule R satisfying $P \cup \{R\} \not\models L$.

First, the set S^+ of LP-literals becomes

$$\begin{aligned} S^+ \ = \ \{\, & bird(tweety),\ bird(polly),\ penguin(polly), \\ & not\, penguin(tweety),\ flies(tweety), \\ & flies(polly),\ not\, ab(tweety),\ not\, ab(polly), \\ & not\, \neg bird(tweety),\ not\, \neg bird(polly), \\ & not\, \neg penguin(polly),\ not\, \neg penguin(tweety), \\ & not\, \neg flies(tweety),\ not\, \neg flies(polly), \\ & not\, \neg ab(tweety),\ not\, \neg ab(polly) \,\}. \end{aligned}$$

From S^+ picking up LP-literals which are relevant to L and are involved in $P \cup \{L\}$, the following integrity constraint is constructed:

$$\leftarrow bird(polly), penguin(polly), flies(polly), not\, ab(polly)\,.$$

Let ab be the target predicate on which $flies$ strongly and negatively depends. Then, shifting $ab(polly)$ to the head, it becomes

$$ab(polly) \leftarrow bird(polly),\ penguin(polly),\ flies(polly)\,.$$

Replacing $polly$ by a variable x, we get

$$R^*:\ ab(x) \leftarrow bird(x),\ penguin(x),\ flies(x)\,.$$

In this case, however, $P \cup \{R^*\}$ is inconsistent.

To get a consistent program, dropping $flies(x)$ from R^*, we get

$$R^{**}: \; ab(x) \leftarrow bird(x),\, penguin(x)$$

where $P \cup \{R^{**}\} \not\models L$ holds. The rule R^{**} is further simplified as

$$ab(x) \leftarrow penguin(x)$$

using the second rule in P.

Discussion

Learning from Multiple Examples

Our algorithm is also applicable to learning from a set of examples by iteratively applying the procedure to each example. For instance, suppose that the set of positive examples $E = \{\, flies(tweety),\, \neg flies(polly) \,\}$ is given to the program P of Example 2. Then, applying the algorithm to each example, the rule

$$R_1^* = flies(x) \leftarrow bird(x),\, not\, penguin(x)$$

is induced by the example $flies(tweety)$, and the rule

$$R_2^* = \neg flies(x) \leftarrow bird(x),\, penguin(x)$$

is induced by the example $\neg flies(polly)$. As a result, $P \cup \{\, R_1^*,\, R_2^* \,\} \models e$ for every $e \in E$. This is also the case for a set of negative examples. When sets of positive and negative examples are given to a program P, firstly induce rules by positive examples and incorporate them into P. In the resulting program P', subsequently induce rules by negative examples. Note that when examples are successively given, the result of induction depends on the order of examples in general. For instance, given the program $P = \{\, bird(tweety) \,\}$ and the positive example $E_1 = \{\, has\_wing(tweety) \,\}$, the rule

$$R_1^* = has\_wing(x) \leftarrow bird(x)$$

is induced. Next, from the program $P \cup \{R_1^*\}$ and the the positive example $E_2 = \{\, flies(tweety) \,\}$, the rule

$$R_2^* = flies(x) \leftarrow bird(x),\, has\_wing(x)$$

is induced. By contrast, from P and E_2 the rule

$$R_3^* = flies(x) \leftarrow bird(x)$$

is induced, and from $P \cup \{R_3^*\}$ and E_1 the rule

$$R_4^* = has\_wing(x) \leftarrow bird(x), flies(x)$$

is induced. Thus, in incremental learning the order of examples taken into consideration affects the program to be induced in general.

Learning from Non-Categorical Programs

In this paper we considered induction in categorical programs. The proposed algorithm is also extensible to programs having multiple answer sets. When a program has more than one answer set, different rules are induced by each answer set. For instance, consider the program P and the positive example L such that

$$\begin{aligned} P: \quad & p(a) \leftarrow not\, q(a), \\ & q(a) \leftarrow not\, p(a). \\ L: \quad & r(a). \end{aligned}$$

Here, P has two answer sets $S_1 = \{p(a)\}$ and $S_2 = \{q(a)\}$. Then, applying the algorithm to each answer set, the rule

$$R_1^* = r(x) \leftarrow p(x),\, not\, q(x)$$

is induced using S_1, while the rule

$$R_2^* = r(x) \leftarrow q(x),\, not\, p(x)$$

is induced using S_2. Consequently, $P \cup \{\, R_1^*,\, R_2^*\} \models L$. For non-categorical programs Theorems 4 and 8 hold, and the sufficient conditions of Theorems 5 and 9 are also extended in a straightforward manner. A formal theory of induction in programs having multiple answer sets will be discussed in the full version of this paper.

Correctness and Completeness

An induction algorithm is *correct* with respect to a positive example (resp. a negative example) L if a rule R produced by the algorithm satisfies $P \cup \{R\} \models L$ (resp. $P \cup \{R\} \not\models L$). We provided sufficient conditions to guarantee the correctness of the proposed algorithm with respect to positive examples (Theorem 5) and negative examples (Theorem 9).

On the other hand, an induction algorithm is *complete* with respect to a positive example (resp. a negative example) L if it produces every rule R satisfying $P \cup \{R\} \models L$ (resp. $P \cup \{R\} \not\models L$). The proposed algorithm is incomplete with respect to both positive and negative examples. For instance, in Example 2 the rule $flies(tweety) \leftarrow bird(polly)$ explains $flies(tweety)$ in P, while this rule is not constructed by the procedure. Generally, there exist possibly infinite solutions for explaining an example. For instance, consider the program P and the positive example L such that

$$\begin{aligned} P: \quad & r(f(x)) \leftarrow r(x), \\ & q(a) \leftarrow, \\ & r(b) \leftarrow . \\ L: \quad & p(a). \end{aligned}$$

Then, the following rules

$$\begin{aligned} & p(x) \leftarrow q(x), \\ & p(x) \leftarrow q(x),\, r(b), \\ & p(x) \leftarrow q(x),\, r(f(b)), \\ & \cdots \end{aligned}$$

all explain $p(a)$. However, every rule except the first one seems meaningless. In the presence of NAF, useless hypotheses

$$\begin{aligned} & p(x) \leftarrow q(x),\, not\, q(b), \\ & p(x) \leftarrow q(x),\, not\, r(a), \\ & \cdots \end{aligned}$$

are also constructed by attaching arbitrary NAF-literal $not\,L$ such that $P \not\models L$ to the body of the rule. These examples show that the completeness of an induction algorithm is of little value in practice because there are tons of useless hypotheses in general. What is important is selecting meaningful hypotheses in the process of induction, and this is realized in our algorithm by filtering out irrelevant or disinvolved literals in S^+ to construct $\leftarrow \Gamma$.

Computability

We considered an ELP which is function-free and has exactly one consistent answer set. With the function-free setting, S^+ is finite and selection of relevant and involved literals from S^+ is done in polynomial-time. An important class of categorical programs is *stratified programs*. When a stratified program is function-free, the perfect model (or equivalently, the answer set) S is constructed in time linear in the size of the program (Schlipf 1995). In this case, an inductive hypothesis R^* is efficiently constructed from S^+.

Connection to Answer Set Programming

Answer set programming (ASP) is a new paradigm of logic programming which attracts much attention recently (Marek and Truszczyński 1999; Niemelä 1999). In the presence of negation as failure in a program, a logic program has multiple intended models in general rather than the single least Herbrand model in a Horn logic program. ASP views a program as a set of constraints which every solution should satisfy, then extracts solutions from the collection of answer sets of the program. We constructed inductive hypotheses from answer sets. In this setting, a background theory and examples work as constraints which inductive hypotheses should satisfy, and induction in nonmonotonic logic programs is realized by computing answer sets of a program. Thus, induction problems in nonmonotonic logic programs are captured as a problem of ASP. The result also implies that existing proof procedures for answer set programming are used for computing hypotheses in nonmonotonic ILP.

Answer sets are used for computing hypotheses in logic programming. For instance, in abduction hypothetical facts which explain an observation are computed using the answer sets of an *abductive logic program* (Kakas and Mancarella 1990; Inoue and Sakama 1996; Sakama and Inoue 1999). By contrast, in induction hypothetical *rules* which explain positive/negative examples are constructed from a background theory and examples. We showed that such rules are automatically constructed using answer sets of a program. The result indicates that answer sets are useful for computing induction as well as abduction, and it enhances commonsense reasoning in logic programming.

Related Work

There are some ILP systems which perform induction in nonmonotonic logic programs. In (Bain and Muggleton 1992; Inoue and Kudoh 1997) a monotonic rule satisfying positive examples is firstly produced and subsequently specialized by incorporating NAF literals to the rule. (Dimopoulos and Kakas 1995) constructs default rules without NAF in a hierarchical structure. In (Bergadano *et al* 1996) the hypotheses space are prepared in advance. (Martin and Vrain 1996) considers learning normal logic programs under the 3-valued semantics. (Seitzer 1997) first computes stable models of the background program, then induces hypotheses using an ordinary ILP algorithms. Our algorithm is essentially different from these work on the point that we construct nonmonotonic rules directly from the background program using answer sets. (Muggleton 1998) constructs a hypothetical clause using the *enlarged bottom set* which is the least Herbrand model augmented by closed world negation. Our algorithm is close to him, but he considers Horn logic programs and induced rules do not contain NAF. The results in this paper are also formulated in a different manner by (Sakama 2000) using the *inverse entailment* in nonmonotonic logic programs.

Summary

This paper studied inductive learning in nonmonotonic logic programs. We provided algorithms for constructing new rules in the face of positive and negative examples, and showed how answer sets can be used to induce definitions of concepts in nonmonotonic logic programs. The results of this paper combines techniques of the two important fields of logic programming, NMLP and ILP, and contributes to a theory of nonmonotonic inductive logic programming.

The proposed algorithms are applicable to an important class of nonmonotonic logic programs including stratified programs. Future research includes extending the algorithms to a wider class of programs, and exploiting further connection between NMLP and ILP.

References

M. Bain and S. Muggleton. Non-monotonic learning. In: S. Muggleton (ed.), *Inductive Logic Programming*, Academic Press, pp. 145–161, 1992.

C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming* 19/20:73–148, 1994.

F. Bergadano, D. Gunetti, M. Nicosia, and G. Ruffo. Learning logic programs with negation as failure. In: L. De Raedt (ed.), *Advances in Inductive Logic Programming*, IOS Press, pp. 107–123, 1996.

Y. Dimopoulos and A. Kakas. Learning nonmonotonic logic programs: learning exceptions. In: *Proceedings of the 8th European Conference on Machine Learning, Lecture Notes in Artificial Intelligence* 912, Springer-Verlag, pp. 122–137, 1995.

M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9:365–385, 1991.

K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming* 27(2):107–136, 1996.

K. Inoue and Y. Kudoh. Learning extended logic programs. In: *Proceedings of the 15th International Joint Conference* *on Artificial Intelligence*, Morgan Kaufmann, pp. 176–181, 1997.

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

V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In: K. R. Apt *et al.* (eds.), *The Logic Programming Paradigm – A 25 Year Perspective*, Springer-Verlag, pp. 375–398, 1999.

L. Martin and C. Vrain. A three-valued framework for the induction of general logic programs. In: L. De Raedt (ed.), *Advances in Inductive Logic Programming*, IOS Press, pp. 219–235, 1996.

S. Muggleton (ed.). *Inductive Logic Programming*, Academic Press, 1992.

S. Muggleton. Completing inverse entailment. In: *Proceedings of the 8th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1446, Springer-Verlag, pp. 245–249, 1998.

I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25:241–273, 1999.

C. Sakama. Inverse entailment in nonmonotonic logic programs. In: *Proceedings of the 10th International Conference on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1866, Springer-Verlag, pp. 209–224, 2000.

C. Sakama and K. Inoue. Updating extended logic programs through abduction. In: *Proceedings of the 5th International Conference Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Artificial Intelligence* 1730, pp. 147–161, Springer-Verlag, 1999.

J. S. Schlipf. Complexity and undecidability results for logic programing. *Annals of Mathematics and Artificial Intelligence* 15:257–288, 1995.

J. Seitzer. Stable ILP: exploring the added expressivity of negation in the background knowledge. In: *Proceedings of IJCAI-95 Workshop on Frontiers of ILP*, 1997.