Induction From Answer Sets in Nonmonotonic Logic Programs

CHIAKI SAKAMA
Wakayama University

Inductive logic programming (ILP) realizes inductive machine learning in computational logic. However, the present ILP mostly handles classical clausal programs, especially Horn logic programs, and has limited applications to learning *nonmonotonic logic programs*. This article studies a method for realizing induction in nonmonotonic logic programs. We consider an *extended logic program* as a background theory, and 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 proposed methods extend the present ILP techniques to a syntactically and semantically richer framework, and contribute to a theory of nonmonotonic ILP.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computational logic*; I.2.6 [**Artificial Intelligence**]: Learning—*induction*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Answer sets, induction, nonmonotonic logic programs

1. INTRODUCTION

Induction realizes concept learning by constructing general sentences from examples. In the context of computational logic, inductive machine learning is realized in the framework of *Inductive Logic Programming* (ILP) [Muggleton 1992; Muggleton and De Raedt 1994; Nienhuys-Cheng and De Wolf 1997]. ILP provides a formal method for inductive learning and has advantages of using computational tools developed in logic programming. The goal of ILP is the inductive construction of logic programs from examples and background knowledge. Induction problems assume background knowledge which is incomplete, otherwise there is no need to learn. Therefore, representing and reasoning with incomplete knowledge are vital issues in ILP. On the other hand, the present ILP mostly handles classical clausal theories, especially Horn logic programs. However, it is known that logic programming based on classical Horn logic is not sufficiently expressive for representing and reasoning with incomplete

Author's address: Department of Computer and Communication Sciences, Wakayama University, Sakaedani, Wakayama 640 8510, Japan; email: sakama@sys.wakayama-u.ac.jp.

knowledge. In real life, humans perform *default reasoning* when one's knowledge is incomplete. Default reasoning is *nonmonotonic* in its feature, that is, previously concluded facts might be withdrawn by the introduction of new information. Horn logic programs are monotonic and inadequate for characterizing nonmonotonic reasoning with incomplete information. To overcome such limitations of Horn logic programming, *nonmonotonic logic programming* (NMLP) is introduced by extending the representation language and enhancing the inference mechanism [Baral and Gelfond 1994; Brewka and Dix 1997]. NMLP aims at representing incomplete knowledge and reasoning with commonsense in a program.

Induction is nonmonotonic reasoning in the sense that once induced hypotheses might be changed by the introduction of new evidence. In ILP based on first-order clausal logic, when one encounters new evidence, a theory which does not account for the evidence is to be modified. For instance, observing a flock of white swans one may induce the rule:

$$white(x) \leftarrow swan(x).$$

When one finds a black swan, the above rule must be changed to account for the new evidence. Using nonmonotonic logic, on the other hand, one does not necessarily need to abandon a previously induced hypothesis in face of a counter-example. For instance, by a flock of white swans one can induce a default rule of Reiter [1980]:

$$\frac{swan(x) : while(x)}{white(x)},$$

representing "swans are normally white". When one finds a black swan, the above default rule is still effective by considering the black swan as an *exception*. Thus, nonmonotonic logic would simplify an induction procedure and effectively represent hypotheses by restricting the domain of applications [Gabbay et al. 1992].

In NMLP, default reasoning is realized using *negation as failure* (NAF). NAF represents default negation in a program, and infers negation of a fact if it is not provable in a program. Using an *extended logic program* [Gelfond and Lifschitz 1990], the above default rule is written as

$$white(x) \leftarrow swan(x),\ not\ \neg white(x)$$

meaning that "if $swan(x)$ holds and $\neg white(x)$ is not proved, then conclude $white(x)$". Here, *not* is the negation-as-failure operator, which is distinguished from classical negation $\neg$.

Some researchers in ILP, however, argue that negation as failure is inappropriate in machine learning. De Raedt and Bruynooghe [1990] say:

> For concept learning, negation as failure (and the underlying closed world assumption) is unacceptable because it acts as if everything is known. Clearly, in learning this is not the case, since otherwise nothing ought to be learned.

Although NAF or the closed world assumption is used for completing a theory, it does not forbid the extension of the theory through induction. In fact, when

background knowledge is given as a Horn logic program, NAF infers negative facts which are not proved from the program. If a new evidence E which is initially assumed false under NAF is observed, this just means that the old assumption $\neg E$ is rebutted. The task of inductive learning is then to revise the old theory to explain the new evidence. Thus, the use of NAF never invalidates the need of learning in logic programming.

On the other hand, if NAF is excluded from a background program, we lose the way of representing default negation in the program. This is a significant drawback in representing knowledge and restricts application of ILP. In fact, NAF enables to write shorter and simpler programs and appears in many basic but practical Prolog programs. For instance, set difference in the relational algebra is written as

$$r\text{-}s(x_1, \ldots, x_k) \leftarrow r(x_1, \ldots, x_k),\ not\ s(x_1, \ldots, x_k),$$

where r and s are relations. Such a rule is not handled if background knowledge is restricted to Horn logic programs. Moreover, induction produces general rules, while rules normally have exceptions in real life. As presented above, nonmonotonic logic can effectively express exceptions which are represented using NAF in NMLP. For instance, consider the following background knowledge B and positive/negative examples E^+/E^-:

$$\begin{aligned} B: \quad & bird(x) \leftarrow penguin(x), \\ & bird(a) \leftarrow, \quad bird(b) \leftarrow, \quad bird(c) \leftarrow, \\ & penguin(d) \leftarrow . \\ E^+: \quad & fly(a), \quad fly(b), \quad fly(c). \\ E^-: \quad & fly(d). \end{aligned}$$

Then, a possible hypothesis is given as

$$fly(x) \leftarrow bird(x),\ not\ penguin(x).$$

By contrast, it would require indirect and unsuccinct representation to specify the same hypothesis using Horn logic programs. Handling exceptions is particularly important in the field of *data mining*. This is because in many applications a simple generalization might not cover all the instances in a database. For instance, suppose that 95% of the patients suffering from a disease might have similar symptoms, but 5% of them might have some unusual symptoms. Under such circumstances, a small number of unusual cases is viewed as noisy or exceptional data, but must be handled appropriately in the process of generalization [Cai et al. 1991].

Thus, realizing induction in the context of NMLP is an important and meaningful step in ILP research. To realize induction in NMLP, it is necessary to extend the representation language and enhance reasoning ability in ILP. There are several studies which extend the ordinary Horn ILP framework to nonmonotonic logic programs [Bain and Muggleton 1992; Dimopoulos and Kakas 1995; Martin and Vrain 1996; Inoue and Kudoh 1997; Seitzer 1997; Fogel and Zaverucha 1998; Lamma et al. 2000; Otero 2001]. Generally speaking, however, ILP techniques in Horn logic programs are not directly applicable to

induction in NMLP due to the difference between nonmonotonic logic and classical clausal logic. For example, *inverse resolution* [Muggleton and Buntine 1992] causes problems in the presence of NAF [Sakama 1999], and *inverse entailment* [Muggleton 1995] is not applicable to NMLP in the original form [Sakama 2000]. Thus, to realize induction in nonmonotonic logic programs, it is necessary to perform dedicated studies to develop new techniques for nonmonotonic ILP.

This article presents techniques of induction in nonmonotonic logic programs. We consider an *extended logic program* as a background theory. An extended logic program can represent incomplete information in a program by distinguishing two types of negation; classical (or explicit) negation and default negation (negation as failure). The semantics of an extended logic program is given by the *answer sets* [Gelfond and Lifschitz 1990], which represent possible beliefs of the program. We then introduce a method of constructing new rules from the answer sets of a program. The produced new rules explain positive/negative examples in the context of ILP. The proposed algorithms are efficiently realized for an important class of nonmonotonic logic programs including stratified programs. From the viewpoint of NMLP, it provides a novel application of *answer set programming* to concept learning in nonmonotonic logic programs. Thus, the results of this paper combine techniques of the two important fields of logic programming, NMLP and ILP, and contribute to a theory of nonmonotonic inductive logic programming.

This article is a revised and extended version of Sakama [2001a]. The rest of this article is organized as follows. Section 2 defines basic notions used in this article. Section 3 provides induction algorithms for learning extended logic programs from positive and negative examples. Section 4 discusses several issues and Section 5 presents related work. Finally, Section 6 concludes the article.

2. EXTENDED LOGIC PROGRAMS

A *program* considered in this article is an *extended logic program* (ELP) [Gelfond and Lifschitz 1990], which is a set of *rules* of the form:

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n \quad (n \geq m), \tag{1}$$

where each L_i is a literal and *not* represents *negation as failure* (NAF). The literal L_0 is the *head* and the conjunction $L_1, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n$ is the *body*. The conjunction in the body is identified with the set of conjuncts. For a rule R, $\mathit{head}(R)$ and $\mathit{body}(R)$ denote the head and the body of R, respectively. We allow a rule with the empty head of the form:

$$\leftarrow L_1, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n \quad (n \geq 1), \tag{2}$$

which is also called an *integrity constraint*. A rule with the empty body $L \leftarrow$ is identified with the literal L and is called a *fact*. An ELP is called a *normal logic program* (NLP) if every literal appearing in the program is an atom. An NLP P is called a *Horn logic program* if no rule in P contains NAF. A Horn logic program is *definite* if it contains no integrity constraint. A rule is *NAF-free* if it contains no *not*, that is, $m = n$ for the rule (1) or (2). A program is NAF-free if it consists of NAF-free rules.

The rule (1) is semantically equivalent to the default rule:

$$\frac{L_1 \wedge \cdots \wedge L_m : \neg L_{m+1}, \ldots, \neg L_n}{L_0}, \tag{3}$$

where $\neg \neg L = L$. Thus, an ELP is considered a *default theory* of Reiter [1980].

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*. For any LP-literal K, $|K| = K$ if K is a literal; and $|K| = L$ if $K = not\, L$. For any LP-literal L, $pred(L)$ denotes the predicate of L and $const(L)$ denotes the set of constants appearing in L. A proposition p is identified with the 1-ary atom $p(nil)$ with the reserved constant nil.[1] A program, a rule, or an LP-literal is *ground* if it contains no variable. Any variable in a program is interpreted as a free variable. A program P is semantically identified with its ground instantiation, that is, the set of ground rules obtained from P by substituting variables in P with elements of the Herbrand universe in every possible way. A ground instance of a rule R is represented as $R\theta$ where θ is a (ground) substitution that maps variables in R to ground terms. We shall use lowercase Greek letters to represent substitutions.

Given a set $S \subset Lit$, we define

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

S^+ is called the *expansion set* of S. A set $S(\subseteq Lit)$ *satisfies* the conjunction of ground LP-literals $C = (L_1, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n)$ (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 a ground integrity constraint of the form (2) if $\{L_1, \ldots, L_m\} \not\subseteq S$ or $\{L_{m+1}, \ldots, L_n\} \cap S \neq \emptyset$. When a rule R contains variables, $S \models R$ means that S satisfies every ground instance of R.

The semantics of ELPs is given by the *answer set semantics* [Gelfond and Lifschitz 1990]. 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 that satisfies every ground rule in the ground instantiation of 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 $L_0 \leftarrow L_1, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n$ in the ground instantiation of P such that $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$. Here, L_0 is possibly empty. 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. An ELP may have none, one, or multiple answer sets. When an ELP P is an NLP, answer sets coincide with *stable models* of [Gelfond and Lifschitz 1988]; a definite Horn logic program has the unique answer set which is the *least Herbrand model*. Regarding an ELP as a collection of default rules of (3), the deductive closure of each answer set coincides with an extension of the corresponding default theory [Gelfond and Lifschitz 1990].

[1]This is just a technical requirement to handle propositions and predicate atoms in a uniform manner.

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

Example 2.1. The program

$$a \leftarrow not\ b,$$
$$b \leftarrow not\ c$$

has the unique answer set $\{b\}$, hence, it is categorical.

The program

$$a \leftarrow not\ b,$$
$$b \leftarrow not\ a$$

has two answer sets $\{a\}$ and $\{b\}$, which represents two alternative beliefs.

The program

$$a \leftarrow not\ a$$

has no answer set, hence it is inconsistent.

The *predicate dependency graph* of a program P is a directed graph such that nodes are predicates in P and there is a *positive edge* (respectively, *negative edge*) from p_1 to p_2 if there is a rule R in P such that $head(R)$ has the predicate p_1 and $body(R)$ contains a literal L (respectively, $not\ L$) 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 the 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. A program contains a *negative-cycle* if there is a predicate which negatively depends on itself. A rule R is called *negative-cycle-free* if there is no negative edge from $pred(head(R))$ to itself. The *literal dependency graph* is defined in the same manner as the predicate dependency graph with the only difference that nodes are ground literals in the ground instantiation of P. For two ground literals L_1 and L_2, the relation that L_1 (strongly/negatively) depends on L_2 is defined analogously to the predicate case.

Example 2.2. Let P be the program:

$$R_1 : p(x) \leftarrow not\ q(x),$$
$$R_2 : q(x) \leftarrow not\ r(x),$$
$$R_3 : q(x) \leftarrow s(x),$$
$$R_4 : r(x) \leftarrow not\ r(x).$$

In P, p strongly and negatively depends on q, and p negatively depends on s but not strongly depends on s. Rules $R_1 - R_3$ are negative-cycle-free, while R_4 is not.

3. INDUCTION IN NONMONOTONIC LOGIC PROGRAMS

3.1 Induction from a Positive Example

Positive examples are facts which are known to be true. In this section, we first consider induction from a positive example. The problem setting considered here is as follows.

Given:

—A *background knowledge base* P as an extended logic program;
—A *positive example* L as a ground literal such that

$$P \not\models L; \tag{4}$$

—A *target predicate* that is subject to learn.

Find: A rule R such that

—$P \cup \{R\}$ is consistent and satisfying the relation

$$P \cup \{R\} \models L; \tag{5}$$

—R has the target predicate in its head.

It is assumed that P, R, and L have the same underlying language.[2]

In the above problem setting, Assumption (4) means that the given example L is not true in the background knowledge base, since if $P \models L$ there is no need to introduce R. A target predicate is a prespecified predicate representing a concept that is subject to learn. In case of induction from a positive example, we identify the target predicate with the one appearing in the example. The problem is then how to construct a rule R satisfying Relation (5) automatically.

We first provide a couple of propositions that are used later.

PROPOSITION 3.1. *Let P be a program and R a rule such that $P \cup \{R\}$ is consistent. For any ground literal L, $P \cup \{R\} \models L$ and $P \models R$ imply $P \models L$.*

PROOF. Suppose that $P \cup \{R\} \models L$, $P \models R$ and $P \not\models L$. Then, there is an answer set S of P such that $L \notin S$. By $P \models R$, $S \models R$ and $S \models \{R\}^S$. 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 $P \cup \{R\} \models L$. □

PROPOSITION 3.2. *Let P be a program and L a ground literal such that $pred(L)$ appears nowhere in P. For any rule R that is negative-cycle-free, if there is a ground instance $R\theta$ of R with a substitution θ such that $P \cup \{R\theta\} \models L$, then $P \cup \{R\} \models L$.*

PROOF. Let $R = (H \leftarrow \Gamma)$ where Γ is a conjunction of LP-literals. Since P does not contain the predicate of L, $P \cup \{R\theta\} \models L$ implies $L = H\theta$. Suppose that $P \cup \{R\theta\}$ has the answer sets $S_1, \ldots, S_n$. By $P \cup \{R\theta\} \models L$, $L \in S_i$ and $S_i \models \Gamma\theta$ for $i = 1, \ldots, n$. Since $pred(L) = pred(H)$ appears nowhere in P and R is negative-cycle-free, $P \cup \{R\}$ has the answer sets

[2]Throughout the article, in induction problems we assume that predicates, functions and constants in the language are those appear in background knowledge and examples.

$T_i = S_i \cup \{H\sigma \mid R\sigma$ is a ground instance of R such that $T_i \models \Gamma\sigma\}$. Thus, every answer set of $P \cup \{R\}$ contains L, hence $P \cup \{R\} \models L$. □

The result of Proposition 3.2 does not hold in general if a rule R contains a negative-cycle.

Example 3.1. Let P be the program:

$$move(a, b) \leftarrow,$$
$$move(b, a) \leftarrow .$$

Then, for the rule

$$R: \ win(x) \leftarrow move(x, y), \ not \ win(y)$$

and the substitution $\theta = \{x/a, \ y/b\}$, $P \cup \{R\theta\}$ has the unique answer set $\{move(a, b), move(b, a), win(a)\}$. Thus, $P \cup \{R\theta\} \models win(a)$. On the other hand, $P \cup \{R\} \not\models win(a)$, because $P \cup \{R\}$ has another answer set $\{move(a, b), move(b, a), win(b)\}$.

By Proposition 3.1, $P \not\models L$ and $P \cup \{R\} \models L$ imply

$$P \not\models R. \tag{6}$$

Relation (6) is a necessary condition for the induction problem satisfying (4) and (5). By Proposition 3.2, on the other hand, if an example L has a predicate which does not appear in P, then finding a negative-cycle-free rule $R\theta$ such that $P \cup \{R\theta\} \models L$ leads to the construction of R satisfying (5). This is a sufficient condition for R.

Proposition 3.2 assumes that the predicate of L appears nowhere in P just by a technical reason. In induction problems, this condition is considered less restrictive by the following reason. In many induction problems, an example L is a newly observed evidence such that the background program P contains no information on $pred(L)$. On the other hand, when P contains a rule with $pred(L)$ in its head, the problem of computing hypotheses satisfying $P \cup \{R\} \models L$ is usually solved using *abduction* [Kakas et al. 1998], which seeks the cause of L in P. Moreover, when $pred(head(R))(= pred(L))$ appears nowhere in P, the condition that R is negative-cycle-free guarantees the consistency of $P \cup \{R\}$ whenever P is consistent. The conditions of Proposition 3.2 are relaxed when P and R are NAF-free.

COROLLARY 3.3. *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 a substitution θ such that $P \cup \{R\theta\} \models L$, then $P \cup \{R\} \models L$.*

PROOF. A consistent NAF-free ELP has the unique answer set. Suppose that P_1 and P_2 are consistent NAF-free programs that 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. □

Corollary 3.3 is useful for ILP problems without NAF.

We use these necessary and sufficient conditions for induction in ELPs. To simplify the problem, in the rest of this section we consider a program P, which is *function-free* and *categorical*. We later consider noncategorical programs in Section 4.1.

Definition 3.1. Given two ground LP-literals L_1 and L_2, the relation $L_1 \sim L_2$ holds if $pred(L_1) = pred(L_2)$ and $const(L_1) = const(L_2)$. Let L be a ground LP-literal and S a set of ground LP-literals. Then, L_1 in S is *relevant* to L if either (i) $L_1 \sim L$ or (ii) for some LP-literal L_2 in S, $const(L_1) \cap const(L_2) \neq \emptyset$ and L_2 is relevant to L. Otherwise, L_1 is *irrelevant* to L.

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 $P \cup \{L\}$. Otherwise, K is *disinvolved* in $P \cup \{L\}$.

Suppose that a background knowledge base P has the unique answer set S. By (6), the following relation holds.

$$S \not\models R. \tag{7}$$

Then, we start on finding a rule R which satisfies Condition (7). Consider the integrity constraint $\leftarrow \Gamma$ with

$$\Gamma = \{K \in S^+ \mid K \text{ is relevant to } L \text{ and is involved in } P \cup \{L\}\}$$

where S^+ is the expansion set of S. Note that we consider a function-free program P, so Γ consists of finite LP-literals. Since S does not satisfy this integrity constraint,

$$S \not\models \leftarrow \Gamma \tag{8}$$

holds. That is, $\leftarrow \Gamma$ is a rule which satisfies Condition (7).

Next, by (4), it holds that $P \not\models L$. Then, $L \notin S$, thereby $not\ L \in S^+$. Since $not\ L$ in S^+ is relevant to L, the integrity constraint $\leftarrow \Gamma$ contains $not\ L$ in its body. Then, shifting the literal L to the head produces

$$L \leftarrow \Gamma' \tag{9}$$

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

Finally, Rule (9) is generalized by constructing a rule R^* such that

$$R^*\theta = (L \leftarrow \Gamma') \tag{10}$$

for some substitution θ.

The algorithm IAS^{pos} (*Induction from Answer Sets* using a positive example) is presented in Figure 1.

The rule R^* satisfies Condition (6).

PROPOSITION 3.2. *Let P be a categorical program and R^* a rule obtained by the algorithm IAS^{pos}. Then, $P \not\models R^*$.*

PROOF. Suppose a rule $L \leftarrow \Gamma'$ of (10). As $\Gamma' \subseteq S^+$ and $L \notin S$, $S \not\models L \leftarrow \Gamma'$. Thus, S does not satisfy a ground instance (10) of R^*. Hence, $S \not\models R^*$, thereby $P \not\models R^*$. □

Procedure: IAS^{pos}

Input. : a function-free and categorical ELP P, a ground literal L representing a positive example, a target predicate $pred(L)$;
Output. : a rule R^*.

(1) Compute the answer set S of P;
(2) Construct the integrity constraint $\leftarrow \Gamma$ from the expansion set S^+;
(3) Produce the rule $L \leftarrow \Gamma'$ by shifting $not\, L$ in Γ to L in the head;
(4) Generate the rule R^* where $R^*\theta = (L \leftarrow \Gamma')$ for some substitution θ.

Fig. 1. Induction from answer sets using a positive example.

The next theorem provides a condition of R^* to become a solution satisfying Relation (5).

THEOREM 3.5. *Let P be a categorical program, L a ground literal representing a positive example, and R^* a rule obtained by the algorithm IAS^{pos}. If R^* is negative-cycle-free and $pred(L)$ appears nowhere in P, then $P \cup \{R^*\} \models L$.*

PROOF. Let $R^*\theta = (L \leftarrow \Gamma')$ be the rule of (10). 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 $P \cup \{R^*\theta\}$. Thus, $P \cup \{R^*\theta\} \models L$. As R^* is negative-cycle-free, the result holds by Proposition 3.2. □

COROLLARY 3.6. *Let P be a categorical program and R^* a rule obtained by the algorithm IAS^{pos}. If R^* is negative-cycle-free and $pred(head(R^*))$ appears nowhere in P, then $P \cup \{R^*\}$ is also categorical.*

PROOF. Let S be the answer set of P. When R^* is negative-cycle-free and $pred(head(R^*))$ appears nowhere in P, $S \cup \{head(R^*\sigma) \mid R^*\sigma$ is a ground instance of R^* such that $S \models body(R^*\sigma)\}$ becomes the unique answer set of $P \cup \{R^*\}$. Hence, the result follows. □

Example 3.2. Consider the following background program P and the positive example L:

$$\begin{aligned} P:\ & bird(x) \leftarrow penguin(x), \\ & bird(tweety) \leftarrow, \\ & penguin(polly) \leftarrow . \\ L:\ & flies(tweety). \end{aligned}$$

Initially, it holds that

$$P \not\models flies(tweety).$$

Our goal is then to construct a rule R satisfying

$$P \cup \{R\} \models flies(tweety).$$

First, the expansion set S^+ of P is

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

$$not\ \neg bird(tweety),\ not\ \neg bird(polly),\ not\ \neg penguin(polly),$$
$$not\ \neg penguin(tweety),\ not\ \neg flies(tweety),\ not\ \neg flies(polly)\}.$$

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 $P \cup \{R^*\} \models flies(tweety)$ holds.

3.2 Induction from a Negative Example

In induction problems, *negative examples* are considered as well as positive ones. Negative examples are facts which are known not to be true. We next consider induction from a negative example. The problem is presented as follows:

Given:

—A background knowledge base P as an extended logic program;
—A *negative example* L as a ground literal such that

$$P \models L\,; \tag{11}$$

—A target predicate which is subject to learn.

Find: A rule R such that

—$P \cup \{R\}$ is consistent and satisfying the relation

$$P \cup \{R\} \not\models L\,; \tag{12}$$

—R has the target predicate in its head.

As before, it is assumed that P, R, and L have the same underlying language.

The problem setting is in contrast to the case of induction from a positive example. Condition (11) presents that the given negative example is initially true in the background knowledge base. The goal is then to find a rule R satisfying (12). Another difference is in the selection of a target predicate. In case of positive examples, the target predicate is identified with the one appearing in a positive example. This is because the purpose of induction from a positive example is to construct a new rule which entails the positive example. In case of negative examples, on the other hand, a negative example L is already entailed from the background program P. 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. The target predicate is then distinguished from the one appearing in L.

Note that in induction from a negative example, our problem setting is slightly different from the one in classical ILP. In classical monotonic logic,

when $P \models L$ for a negative example L, there is no way to introduce a new rule R to satisfy $P \cup \{R\} \not\models L$. The task is then to change some existing rules in P to block the derivation of L, and the target predicate is usually identical to the one appearing in a negative example. By contrast, we do not change existing rules in a program, but introduce a new rule for this purpose. Such a solution is characteristic when the background theory is a nonmonotonic logic program.[3]

As the case of positive examples, we first introduce a necessary condition for computing R.

PROPOSITION 3.7. *Let P be a program and R a rule such that $P \cup \{R\}$ is consistent. For any ground literal L, $P \cup \{R\} \not\models L$ and $P \models R$ imply $P \not\models L$.*

PROOF. Suppose that $P \cup \{R\} \not\models L$, $P \models R$ and $P \models L$. Then, for any answer set S of P, $L \in S$. By $P \models R$, $S \models R$ and $S \models \{R\}^S$. Then, S is an answer set of $P^S \cup \{R\}^S$, hence an answer set of $P \cup \{R\}$. As $L \in S$, this contradicts the assumption $P \cup \{R\} \not\models L$. □

PROPOSITION 3.8. *Let P be a program and L a ground literal. Suppose a ground rule $R\theta$ with a substitution θ such that $P \cup \{R\theta\} \not\models L$. If $P \cup \{R\theta\} \models R$ and $P \cup \{R\}$ is consistent, then $P \cup \{R\} \not\models L$.*

PROOF. Since $P \cup \{R\theta\} \not\models L$, there is an answer set S of $P \cup \{R\theta\}$ such that $L \notin S$. By $P \cup \{R\theta\} \models R$, every answer set of $P \cup \{R\theta\}$ satisfies R, thereby $S \models R$. Then, S is an answer set of a consistent program $P \cup \{R\}$. Hence, $P \cup \{R\} \not\models L$. □

Example 3.3. Consider the following background program P and the negative example L:

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

Let $R = q(x) \leftarrow r(x)$ and $\theta = \{x/a\}$. Then, $P \cup \{R\theta\} \models R$, so that $P \cup \{R\theta\} \not\models L$ implies $P \cup \{R\} \not\models L$.

By Proposition 3.7, the relation

$$P \not\models R \tag{13}$$

is a necessary condition for the problem satisfying (11) and (12). On the other hand, if we successfully find a ground rule $R\theta$ such that $P \cup \{R\theta\} \not\models L$, $P \cup \{R\theta\} \models R$, and $P \cup \{R\}$ is consistent, then $P \cup \{R\} \not\models L$ holds by Proposition 3.8.

Using these results, we construct a rule in a manner similar to the case of a positive example.

In the rest of this subsection, a program P is assumed to be function-free and categorical. Suppose that a background knowledge base P has the unique

[3]We later discuss (dis)advantage of our solution in Section 4.2.

answer set S. By (13), the relation

$$S \not\models R$$

holds. The integrity constraint $\leftarrow \Gamma$ is constructed with the set $\Gamma \subseteq S^+$ of ground LP-literals which are relevant to the negative example L and are involved in $P \cup \{L\}$. Remark that $L \in \Gamma$ by (11). Then, the relation

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

holds. To construct an objective rule from the integrity constraint $\leftarrow \Gamma$, we shift a literal which has a target predicate.

We select a target predicate as the predicate of an LP-literal K on which L strongly and negatively depends in the literal dependency graph of P. Thus, if Γ contains an LP-literal *not* K, which contains the target predicate satisfying this condition, we construct the rule

$$K \leftarrow \Gamma' \tag{14}$$

from $\leftarrow \Gamma$ where $\Gamma' = \Gamma \setminus \{not\ K\}$. Note that there may be none, one, or multiple K satisfying the above condition. Here, we restrict our attention to the case that the selection of K is possible and decidable, that is, there is a single *not* K in Γ such that L strongly and negatively depends on K in P.

Rule (14) is generalized to R^* as

$$R^*\theta = (K \leftarrow \Gamma')$$

by replacing constants with appropriate variables. The rule R^*, however, produces a negative-cycle in $P \cup \{R^*\}$ and the program $P \cup \{R^*\}$ is inconsistent in general. This is because $pred(L)$ strongly and negatively depends on $pred(K)$ in P, and Γ' contains L. To obtain a consistent program, construct a rule R^{**} as

$$R^{**}\theta = (K \leftarrow \Gamma''), \tag{15}$$

where Γ'' is obtained from Γ' by dropping every LP-literal with a predicate that strongly and negatively depends on $pred(K)$ in P.

The algorithm IAS^{neg} (*Induction from Answer Sets* using a negative example) is presented in Figure 2.

By its construction, R^{**} satisfies Condition (13).

PROPOSITION 3.9. *Let P be a categorical program and R^{**} a rule obtained by the algorithm IAS^{neg}. Then, $P \not\models R^{**}$.*

PROOF. Suppose a rule $K \leftarrow \Gamma''$ of (15). As $\Gamma'' \subseteq S^+$ and $K \notin S, S \not\models K \leftarrow \Gamma''$. Thus, S does not satisfy a ground instance (15) of R^{**}, hence $S \not\models R^{**}$ thereby $P \not\models R^{**}$. □

The next theorem provides a sufficient condition for R^{**} to become a solution satisfying (12).

THEOREM 3.10. *Let P be a categorical program, L a ground literal representing a negative example, and $R^{**}\theta$ a rule produced in the fourth step of the algorithm IAS^{neg}. If $P \cup \{R^{**}\theta\} \models R^{**}$ and $P \cup \{R^{**}\}$ is consistent, then $P \cup \{R^{**}\} \not\models L$.*

Procedure: IAS^{neg}

Input. : a function-free and categorical ELP P, a ground literal L representing a negative example, a target predicate $pred(K)$ where L strongly and negatively depends on a literal K in P;
Output. : a rule R^{**}.

(1) Compute the answer set S of P;
(2) Construct the integrity constraint $\leftarrow \Gamma$ from the expansion set S^+;
(3) Produce the rule $K \leftarrow \Gamma'$ by shifting $not\, K$ in Γ to K in the head;
(4) Generate the rule R^{**} where $R^{**}\theta = (K \leftarrow \Gamma'')$ for some substitution θ and Γ'' is obtained from Γ' by dropping every LP-literal with a predicate which strongly and negatively depends on $pred(K)$ in P.

Fig. 2. Induction from answer sets using a negative example.

PROOF. Let $R^{**}\theta = (K \leftarrow \Gamma'')$ be the rule of (15). Suppose that there is an answer set S of $P \cup \{R^{**}\theta\}$ such that $S \not\models \Gamma''$. Then, S is also the answer set of P. On the other hand, by the construction of $R^{**}\theta$, $S \models \Gamma''$. Contradiction. Thus, for any answer set S of $P \cup \{R^{**}\theta\}$, $S \models \Gamma''$. Then, $K \in S$. Since L strongly and negatively depends on K in P, $L \notin S$. Hence, $P \cup \{R^{**}\theta\} \not\models L$. When $P \cup \{R^{**}\theta\} \models R^{**}$ and $P \cup \{R^{**}\}$ is consistent, the result holds by Proposition 3.8. □

Example 3.4. Consider the following background program P and the negative example L:

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

Initially, it holds that

$$P \models flies(polly).$$

Our goal is then to construct a rule R satisfying

$$P \cup \{R\} \not\models flies(polly).$$

First, the expansion set S^+ of P is

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

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

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

Select ab as the target predicate where *flies*(*polly*) strongly and negatively depends on ab(*polly*) in P. Then, shifting ab(*polly*) to the head, it becomes

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

Dropping *flies*(*polly*) from the body and replacing *polly* by a variable x, we get

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

Here, $P \cup \{R^{**}\theta\} \models R^{**}$ with $\theta = \{x/polly\}$ and $P \cup \{R^{**}\}$ is consistent, then $P \cup \{R^{**}\} \not\models flies(polly)$ holds.

The rule R^{**} is further simplified as

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

using the second rule in P.

In contrast to the case of positive examples, for a categorical program P, $P \cup \{R^{**}\}$ is not necessarily categorical.

Example 3.5. Consider the following background program P and the negative example L:

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

Here, P has the unique answer set $\{p(a),\ p(b),\ r(b)\}$. Then, the integrity constraint $\leftarrow \Gamma$ becomes

$$\leftarrow p(a),\ not\ q(a),\ not\ r(a).$$

As the target predicate is q, R^{**} becomes

$$R^{**}:\ q(x) \leftarrow not\ r(x).$$

The program $P \cup \{R^{**}\}$ has two answer sets: $\{q(a), q(b)\}$ and $\{p(b), q(a), r(b)\}$, and $P \cup \{R^{**}\} \not\models p(a)$ holds.

3.3 Induction from Multiple Examples

In Section 3.1, we presented an algorithm for induction from a single positive example. The algorithm is also applicable to induction from a set of positive 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 3.2. Then, applying the algorithm IAS^{pos} 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, $P \cup \{R_1^*,\ R_2^*\} \models L$ for every $L \in E$. Generally, given a set of positive examples $\{L_1, \ldots, L_n\}$, the algorithm

IAS^{pos} is applied to each $P \cup \{L_1\}, \ldots, P \cup \{L_n\}$. When the resulting rules $R_1^*, \ldots, R_n^*$ satisfy the condition of Theorem 3.5, $P \cup \{R_1^*, \ldots, R_n^*\} \models L_i$ holds for $i = 1, \ldots, n$. Formally,

THEOREM 3.11. *Let P be a categorical program and $\{L_1, \ldots, L_n\}$ a set of ground literals representing positive examples. Suppose that a rule R_i^* is obtained by the algorithm IAS^{pos} from P and L_i $(1 \leq i \leq n)$. If each R_i^* is negative-cycle-free and $pred(L_i)$ $(1 \leq i \leq n)$ appears nowhere in P, then $P \cup \{R_1^*, \ldots, R_n^*\} \models L_i$ holds for $i = 1, \ldots, n$.*

PROOF. If each R_i^* is negative-cycle-free and $pred(L_i) = pred(head(R_i^*))$ appears nowhere in P, the program $P \cup \{R_1^*, \ldots, R_n^*\}$ has the answer set $S \cup T$ where S is the answer set of P and $T = \{head(R_i^*\sigma) \mid R_i^*\sigma$ is a ground instance of R_i^* $(1 \leq i \leq n)$ such that $S \models body(R_i^*\sigma)\}$. Let $R_i^*\theta_i = (L_i \leftarrow \Gamma_i')$ be a rule produced in the fourth step of IAS^{pos}. Then, by the construction of $R_i^*\theta_i$, $S \models \Gamma_i'$, thereby $L_i \in T$ for $i = 1, \ldots, n$. Hence, the result holds. □

The following result holds for induction from a set of negative examples.

THEOREM 3.12. *Let P be a categorical program and $\{L_1, \ldots, L_n\}$ a set of ground literals representing negative examples. Suppose that a rule $R_i^{**}\theta_i = (K_i \leftarrow \Gamma_i'')$ is produced in the fourth step of the algorithm IAS^{neg} from P and L_i $(1 \leq i \leq n)$ where $P \cup \{R_i^{**}\theta_i\} \models R_i^{**}$ and $P \cup \{R_i^{**}\}$ is consistent. If $pred(K_i)$ depends on no $pred(K_j)$ $(1 \leq i, j \leq n)$ in the predicate dependency graph of a consistent program $P \cup \{R_1^{**}, \ldots, R_n^{**}\}$, then $P \cup \{R_1^{**}, \ldots, R_n^{**}\} \not\models L_i$ for $i = 1, \ldots, n$.*

PROOF. By the proof of Theorem 3.10, for any answer set S of $P \cup \{R_i^{**}\theta_i\}$, $L_i \notin S$ and $K_i \in S$. Suppose that $P \cup \{R_1^{**}, \ldots, R_n^{**}\} \models L_i$ for some i. Then, $L_i \in T$ and $K_i \notin T$ for every answer set T of $P \cup \{R_1^{**}, \ldots, R_n^{**}\}$. Here, $K_i \in S \setminus T$ is due to the existence of some ground literal $head(R_j^{**}\sigma)$ in $T \setminus S$ $(1 \leq j \leq n)$. However, this cannot happen, since $pred(K_i)$ does not depend on $pred(K_j) = pred(head(R_j^{**}\sigma))$. Thus, $P \cup \{R_1^{**}, \ldots, R_n^{**}\} \not\models L_i$ for $i = 1, \ldots, n$. □

When $pred(K_i)$ depends on some $pred(K_j)$ $(1 \leq i, j \leq n)$, the obtained rules $R_1^{**}, \ldots, R_n^{**}$ may interact with one another, and $P \cup \{R_i^{**}\} \not\models L_i$ $(1 \leq i \leq n)$ does not imply $P \cup \{R_1^{**}, \ldots, R_n^{**}\} \not\models L_i$ in general.

Example 3.6. Consider the following background program P and the negative examples L_1 and L_2:

$$
\begin{aligned}
P:\ & p(x) \leftarrow not\ q(x),\\
& r(a) \leftarrow q(b),\\
& r(b) \leftarrow, \quad s(a) \leftarrow .\\
L_1:\ & p(a).\\
L_2:\ & p(b).
\end{aligned}
$$

Here, P has the single answer set $\{p(a),\ p(b),\ r(b),\ s(a)\}$. Applying the algorithm IAS^{neg},

$$R_1^{**}:\ q(x) \leftarrow s(x),\ not\ r(x)$$

is obtained from P and L_1, and

$$R_2^{**}:\ q(x) \leftarrow r(x),\ not\ s(x)$$

is obtained from P and L_2. The program $P \cup \{R_1^{**}\}$ has the single answer set $\{p(b),\ q(a),\ r(b),\ s(a)\}$, thereby $P \cup \{R_1^{**}\} \not\models L_1$. And the program $P \cup \{R_2^{**}\}$ has the single answer set $\{p(a),\ q(b),\ r(a),\ r(b),\ s(a)\}$, thereby $P \cup \{R_2^{**}\} \not\models L_2$. However, $P \cup \{R_1^{**}, R_2^{**}\}$ has the single answer set $\{p(a),\ q(b),\ r(a),\ r(b),\ s(a)\}$ and $P \cup \{R_1^{**}, R_2^{**}\} \models L_1$. Note here that q depends on itself in $P \cup \{R_1^{**},\ R_2^{**}\}$.

When a mixed set of positive and negative examples is given to a program P, we firstly induce rules by positive examples and incorporate them into P. Then, in the resulting program P', we subsequently induce rules by negative examples. In this case, however, it does not necessarily produce a solution which satisfies both positive and negative examples. For instance, given the program

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

and the positive example $L = p(a)$, the hypothesis

$$R^*:\ p(x) \leftarrow q(x),\ not\, r(x)$$

is induced by IAS^{pos} and $P \cup \{R^*\} \models L$. Next, given the negative example $L' = p(b)$ to $P \cup \{R^*\}$, the hypothesis

$$R^{**}:\ r(x) \leftarrow q(x)$$

is induced by IAS^{neg} and $P \cup \{R^*,\ R^{**}\} \not\models L'$, but $P \cup \{R^*,\ R^{**}\} \not\models L$. This failure comes from the fact that the program P has no information to distinguish between a and b. As a result, the procedure fails to construct a general rule which distinguishes $p(a)$ and $p(b)$.

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

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

is induced by IAS^{pos}. 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 induced program in general.[4]

In classical ILP, once a positive example is covered by induction, the example is covered after further generalization. Suppose that a positive example L_1 is given to a first-order theory P and induction constructs a rule R_1 satisfying $P \cup \{R_1\} \models L_1$. Suppose further that another positive example L_2 is given to the new knowledge base $P_1 = P \cup \{R_1\}$ and induction constructs a rule R_2 satisfying $P_1 \cup \{R_2\} \models L_2$. In this case, the knowledge base $P_1 \cup \{R_2\}$ still implies the previous example L_1, that is, $P_1 \cup \{R_2\} \models L_1$. This is due to the monotonicity of classical logic. In case of nonmonotonic theories, however, such monotonicity does not hold in general. That is, introducing R_2 to P_1 may block the derivation of the previous example L_1. For instance, put $P = \emptyset$, $R_1 = (p \leftarrow not\, q)$, $R_2 = (q \leftarrow not\, r)$, $L_1 = p$, and $L_2 = q$. Then, $P \cup \{R_1\} \models p$ and $P \cup \{R_1,\ R_2\} \models q$, but $P \cup \{R_1,\ R_2\} \not\models p$.

Such nonmonotonicity inevitably happens in nonmonotonic logic programs, but it is an undesired property for induction. Our induction algorithm IAS^{pos} has the monotonicity property with respect to positive examples if the obtained rule R is negative-cycle-free and *pred(head*(R)) appears nowhere in the background program P. More precisely,

THEOREM 3.13. *Let P be a categorical program, and L_1 and L_2 be ground literals representing positive examples such that pred(L_1) and pred(L_2) appear nowhere in P. Suppose that a negative-cycle-free rule R_1^* is obtained by the algorithm IAS^{pos} from P and L_1. Also, suppose that a negative-cycle-free rule R_2^* is obtained by the algorithm IAS^{pos} from $P \cup \{R_1^*\}$ and L_2. Then, $P \cup \{R_1^*,\ R_2^*\} \models L_i$ $(i = 1, 2)$.*

PROOF. By Theorem 3.5, $P \cup \{R_1^*\} \models L_1$ and $P \cup \{R_1^*\}$ is categorical (Corollary 3.6). Since R_2^* has the predicate $pred(L_2)$ in its head and $pred(L_2)$ appears nowhere in P, it holds that either (i) $pred(L_2)$ appears nowhere in $P \cup \{R_1^*\}$ or (ii) $pred(L_2) = pred(L_1)$. In case of (i), $P \cup \{R_1^*,\ R_2^*\} \models L_2$ by Theorem 3.5 and $P \cup \{R_1^*,\ R_2^*\}$ is categorical. As the body of R_1^* is still satisfied in the answer set of $P \cup \{R_1^*,\ R_2^*\}$, $P \cup \{R_1^*,\ R_2^*\} \models L_1$. In case of (ii), let $L_2 = head(R_2^*\theta)$. By the construction of $R_2^*\theta$, the answer set S of $P \cup \{R_1^*\}$ satisfies $body(R_2^*\theta)$. Then, $P \cup \{R_1^*,\ R_2^*\theta\}$ has the answer set $S \cup \{L_2\}$, so that $P \cup \{R_1^*,\ R_2^*\theta\} \models L_1$ and $P \cup \{R_1^*,\ R_2^*\theta\} \models L_2$. Since R_1^* and R_2^* are negative-cycle-free, the introduction of any ground instance $R_2^*\sigma$ to $P \cup \{R_1^*,\ R_2^*\theta\}$ does not affect the entailment of L_1 and L_2. Hence, $P \cup \{R_1^*,\ R_2^*\} \models L_1$ and $P \cup \{R_1^*,\ R_2^*\} \models L_2$. □

For induction from negative examples, the following result holds:

THEOREM 3.14. *Let P be a categorical program, and L_1 and L_2 be ground literals representing negative examples. Suppose that a rule $R_1^{**}\theta_1 = (K_1 \leftarrow \Gamma_1'')$ is produced in the fourth step of the algorithm IAS^{neg} from P and L_1, where $P \cup \{R_1^{**}\theta_1\} \models R_1^{**}$ and $P \cup \{R_1^{**}\}$ is consistent and categorical. Also, suppose that*

[4]In this example, however, unfolding R_2^* in $P \cup \{R_1^*\}$ produces R_3^*, and unfolding R_4^* in $P \cup \{R_3^*\}$ produces R_1^*. Thus, the two programs $P \cup \{R_1^*,\ R_2^*\}$ and $P \cup \{R_3^*,\ R_4^*\}$ are semantically equivalent. At the moment, it is unknown whether such an equivalence relation holds in general.

a rule $R_2^{**}\theta_2 = (K_2 \leftarrow \Gamma_2'')$ *is produced by the algorithm* IAS^{neg} *from* $P \cup \{R_1^{**}\}$ *and* L_2*, where* $P \cup \{R_1^{**}, R_2^{**}\theta_2\} \models R_2^{**}$ *and* $P \cup \{R_1^{**}, R_2^{**}\}$ *is consistent. If* $pred(K_i)$ *does not depend on* $pred(K_j)$ *for* $i, j = 1, 2$ *in* $P \cup \{R_1^{**}, R_2^{**}\}$*, then* $P \cup \{R_1^{**}, R_2^{**}\} \not\models L_i$ $(i = 1, 2)$.

PROOF. Similar to the proof of Theorem 3.12. □

4. DISCUSSION

4.1 Induction in Noncategorical Programs

In Section 3, we considered induction in categorical programs. The proposed algorithm is also extensible to noncategorical programs having multiple answer sets. When a program has more than one answer set, different rules are induced by each answer set. In induction from a positive example, Relation (6)

$$P \not\models R$$

is a necessary condition as before. When P has multiple answer sets, this relation presents that some answer set of P does not satisfy R. For each answer set S_i of P, we consider a rule R_i satisfying Relation (7)

$$S_i \not\models R_i\,.$$

A rule R_i^* is constructed from each S_i in the same manner as presented in Section 3.1. The result of Theorem 3.5 is then extended to noncategorical programs as follows:

THEOREM 4.1. *Let* P *be a program and* L *a ground literal representing a positive example. Suppose that* P *has the answer sets* $S_1, \ldots, S_n$ *and the algorithm* IAS^{pos} *produces the rules* R_i^* *using* S_i $(i = 1, \ldots, n)$*. If each* R_i^* *is negative-cycle-free and* $pred(L)$ *appears nowhere in* P*, then* $P \cup \{R_1^*, \ldots, R_n^*\} \models L$.

PROOF. If each R_i^* is negative-cycle-free and $pred(L)$ appears nowhere in P, $P \cup \{R_1^*, \ldots, R_n^*\}$ is consistent. Let $R_i^*\theta_i = (L \leftarrow \Gamma_i')$ be a rule produced in the fourth step of IAS^{pos}. By the construction of $R_i^*\theta_i$, $S_i \models \Gamma_i'$. Thus, $P \cup \{R_1^*\theta_1, \ldots, R_n^*\theta_n\}$ has the answer sets $S_1 \cup \{L\}, \ldots, S_n \cup \{L\}$. Then, $P \cup \{R_1^*, \ldots, R_n^*\}$ has the answer sets $T_i = S_i \cup \{L\} \cup \{head(R_j^*\sigma) \mid R_j^*\sigma$ is a ground instance of R_j^* such that $S_i \models body(R_j^*\sigma)$ $(1 \le j \le n)\}$ for $i = 1, \ldots, n$. Hence, $P \cup \{R_1^*, \ldots, R_n^*\} \models L$. □

Example 4.1. Consider the following background program P and the positive example L:

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

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

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

is induced using S_1, while the rule

$$R_2^* : r(x) \leftarrow q(x), \mathit{not}\ p(x)$$

is induced using S_2. As a result, $P \cup \{R_1^*, R_2^*\} \models L$ holds.

For induction from a negative example, Theorem 3.10 is extended to non-categorical programs as follows.

THEOREM 4.2. *Let P be a program and L a ground literal representing a negative example. Suppose that P has the answer sets $S_1, \ldots, S_n$ and the algorithm IAS^{neg} produces the rules $R_i^{**}\theta_i = (K \leftarrow \Gamma_i'')$ using S_i $(i = 1, \ldots, n)$. If $P \cup \{R_1^{**}\theta_1, \ldots, R_n^{**}\theta_n\} \models R_i^{**}$ $(i = 1, \ldots, n)$ and $P \cup \{R_1^{**}, \ldots, R_n^{**}\}$ is consistent, then $P \cup \{R_1^{**}, \ldots, R_n^{**}\} \not\models L$.*

PROOF. Suppose that there is an answer set S of $P \cup \{R_1^{**}\theta_1, \ldots, R_n^{**}\theta_n\}$ such that $S \not\models \Gamma_i''$ for any $i = 1, \ldots, n$. Then, S becomes an answer set of P. On the other hand, $R_i^{**}\theta_i$ $(i = 1, \ldots, n)$ is constructed by each answer set of P, so that there is a rule $R_i^{**}\theta_i$ satisfying $S \models \Gamma_i''$. Contradiction. Thus, for every answer set S of $P \cup \{R_1^{**}\theta_1, \ldots, R_n^{**}\theta_n\}$, there is a rule $R_i^{**}\theta_i$ such that $S \models \Gamma_i''$. Then, $K \in S$ and $L \notin S$. Hence, $P \cup \{R_1^{**}\theta_1, \ldots, R_n^{**}\theta_n\} \not\models L$. By $P \cup \{R_1^{**}\theta_1, \ldots, R_n^{**}\theta_n\} \models R_i^{**}$ $(i = 1, \ldots, n)$, every answer set of $P \cup \{R_1^{**}\theta_1, \ldots, R_n^{**}\theta_n\}$ satisfies R_i^{**}, thereby $S \models R_i^{**}$ $(i = 1, \ldots, n)$. Thus, S is an answer set of a consistent program $P \cup \{R_1^{**}, \ldots, R_n^{**}\}$. Hence, $P \cup \{R_1^{**}, \ldots, R_n^{**}\} \not\models L$. □

In Section 3, we considered function-free background programs. When a program P contains functions, the expansion set S^+ is infinite in general. In this case, the integrity constraint $\leftarrow \Gamma$ is built from a finite subset Γ of S^+, where Γ consists of LP-literals which are relevant to the example L and are involved in $P \cup \{L\}$, and $\mathit{not}\ L \in \Gamma$.

Example 4.2. Consider the following background program P and the positive example L:

$$\begin{aligned} P :\ & \mathit{even}(s(x)) \leftarrow \mathit{not}\, \mathit{even}(x), \\ & \mathit{even}(0) \leftarrow . \\ L :\ & \mathit{odd}(s(0)), \end{aligned}$$

where $P \not\models L$. Then, the expansion set of P becomes

$$\begin{aligned} S^+ = \{ & \mathit{even}(0), \mathit{not}\ \mathit{even}(s(0)), \mathit{even}(s(s(0))), \ldots, \\ & \mathit{not}\ \mathit{odd}(0), \mathit{not}\ \mathit{odd}(s(0)), \mathit{not}\ \mathit{odd}(s(s(0))), \ldots\}. \end{aligned}$$

Build the constraint as

$$\leftarrow \mathit{even}(0), \mathit{not}\ \mathit{odd}(s(0)).$$

By shifting $\mathit{odd}(s(0))$ to the head and generalizing it, the rule

$$R^* : \mathit{odd}(s(x)) \leftarrow \mathit{even}(x)$$

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

In the above example, other solutions such as $odd(s(x)) \leftarrow not\ even(s(x))$ and $odd(s(x)) \leftarrow even(s(s(x)))$ are also constructed by different selection of Γ. The rule $odd(s(x)) \leftarrow not\ odd(x)$ containing a negative-cycle is also constructed, which becomes a solution. However, a rule with a negative-cycle does not always become a solution, e.g., $odd(s(x)) \leftarrow not\ odd(s(s(x)))$.

4.2 Correctness and Completeness

An induction algorithm is *correct* with respect to a positive example (respectively, a negative example) L if a rule R produced by the algorithm satisfies $P \cup \{R\} \models L$ (respectively, $P \cup \{R\} \not\models L$). We provided sufficient conditions to guarantee the correctness of the proposed algorithm with respect to a positive example (Theorem 3.5) and a negative example (Theorem 3.10).

On the other hand, an induction algorithm is *complete* with respect to a positive example (respectively, a negative example) L if it produces every rule R satisfying $P \cup \{R\} \models L$ (respectively, $P \cup \{R\} \not\models L$). The proposed algorithm is incomplete with respect to both positive and negative examples. For instance, in Example 3.2, the rule $flies(tweety) \leftarrow bird(polly)$ explains $flies(tweety)$ in P, while this rule is not constructed by the procedure IAS^{pos}. Generally, there exist possibly infinite solutions for explaining an example. For instance, consider the program P and the positive example L such that

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

Then, the following rules

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

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

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

are easily constructed by attaching arbitrary NAF-literal $not\ L$ such that $P \not\models L$ to the body of a rule. These examples show that the completeness of induction algorithms without any restriction is of little value in practice because there are huge number of useless hypotheses in general. What is important is selecting meaningful hypotheses in the process of induction, and this is realized in our algorithms by filtering out irrelevant or disinvolved LP-literals in the expansion set S^+ to construct the constraint $\leftarrow \Gamma$. Intuitively, irrelevant LP-literals have no connection with the given example in a background program. In the

above example, $r(b)$, $r(f(b)), \ldots$ are irrelevant to $p(a)$. On the other hand, a disinvolved LP-literal is an LP-literal L or $not\, L$ such that L never appears in a background program nor an example. For instance, in the above example $not\, \neg q(a)$ is relevant to $p(a)$ in P, but $\neg q(a)$ never appears in $P \cup \{L\}$ hence $not\, \neg q(a)$ is disinvolved. The conditions of relevance and involvement are considered *inductive bias* to reduce the search space of inductive hypotheses.

Our induction algorithms compute a single rule as a hypothesis, but cannot find a set of hypotheses which work together to explain an example. For instance, consider the background program P and the positive example L:

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

Then, $H = \{p \leftarrow r,\ \ q \leftarrow\}$ satisfies $P \cup H \models L$, but such a set of rules is not built from the algorithm IAS^{pos}. In the above example, P is inconsistent (having no answer set), so that the rule $q \leftarrow$ is introduced to make $P \cup H$ consistent.

Our induction algorithms explain positive/negative examples by introducing a new rule to a program, but they do not change nor delete existing rules in the background program to explain new evidences. Techniques of modifying the existing knowledge base using given examples are studied in the context of *theory refinement* [Richards and Mooney 1995]. Our present algorithms are considered restrictive in the sense that they do not modify the existing knowledge base. However, this restriction brings an advantage that it is easy to recover the old knowledge base when there is an error in examples. That is, we can recover the old knowledge base just by abandoning newly acquired knowledge. Our algorithms do not change existing rules, but the result of induction often has the same effect as modifying rules in a program. For instance, consider the background program P and the positive example L:

$$\begin{aligned} P:\ & p \leftarrow q,\ r, \\ & q \leftarrow . \\ L:\ & p. \end{aligned}$$

The expansion set S^+ of P becomes $\{q,\ not\, p,\ not\, r\}$, then the hypothesis $R = (p \leftarrow q,\ not\, r)$ is constructed by IAS^{pos}. On the other hand, the rules $p \leftarrow q, r$ and $p \leftarrow q,\ not\, r$ in $P \cup \{R\}$ are merged as $p \leftarrow q$, which is equivalent to the rule obtained by dropping r from the first rule in P. Thus, combining our induction algorithms with appropriate program transformation techniques would build possible hypotheses in a more flexible manner.

4.3 Computability

In Section 3, we considered a program which is function-free and categorical. An important class of categorical programs is *(locally) stratified programs* [Przymusinski 1988]. When a stratified program P is function-free, the answer set (or equivalently, the *perfect model*) M of P is constructed in time linear in the size of the program [Schlipf 1995]. In this case, the expansion set $M^+ = M \cup \{not\, A \mid A \in HB \setminus M\}$, where HB is the Herbrand base of P, is

also finite and the selection of relevant LP-literals (with respect to the input example L) from M^+ is done in polynomial-time.[5] More precisely, let $\#p$ be the number of predicates with the maximum arity k in $P \cup \{L\}$ and $\#c$ the number of constants in the Herbrand universe of $P \cup \{L\}$. Then, the size of the expansion set M^+ of P has an upper bound of $\#p \cdot (\#c)^k$. As a result, a hypothesis R^* or R^{**} is efficiently constructed from M^+.

In a non-stratified program, computation of answer sets is intractable in general. Nevertheless, once an answer set is obtained, a possible hypothesis based on its expansion set is constructed in polynomial time.

5. RELATED WORK

5.1 Learning Nonmonotonic LPs

There are some ILP systems which learn nonmonotonic logic programs. Bain and Muggleton [1992] introduce an algorithm called *closed world specialization*. In this algorithm, monotonic rules satisfying positive examples are first constructed and they are subsequently specialized by incorporating NAF literals to the bodies of rules. Inoue and Kudoh [1997] and Lamma et al. [2000] introduce algorithms for learning extended logic programs. They also divide the process of learning nonmonotonic logic programs into two steps: producing monotonic rules by an ordinary induction algorithm for Horn ILP, then specializing them by introducing NAF-literals to the bodies. However, these algorithms based on Horn ILP have problems such that once nonmonotonic rules are learned and incorporated into the background knowledge, Horn induction algorithms cannot be applied anymore to the new (nonmonotonic) background knowledge. Nicolas and Duval [2001] present an algorithm for learning a default theory from a set of positive/negative examples. Since an ELP is viewed as a default theory, the algorithm proposed in this article is also considered a method of learning default theories. Comparing the results of induction, they learn a set of seminormal default rules of the form $(\alpha : \beta \wedge \gamma)/\gamma$, while default rules induced by our algorithms are not necessarily seminormal. Like Bain and Muggleton [1992], Inoue and Kudoh [1997], and Lamma et al. [2000], their algorithm relies on an ordinary Horn ILP procedure to acquire concept definitions, which is followed by a prover for default logic to remove exceptions.

In addition, there is an important difference between [Inoue and Kudoh 1997; Lamma et al. 2000; Nicolas and Duval 2001] and ours for handling negative examples. Those studies represent negative examples by negative literals and define the problem of induction from a negative example L as finding a set of default rules H satisfying $P \cup H \models L$. By contrast, we give a negative example L as a positive/negative literal and define the problem as computing a rule R satisfying $P \cup \{R\} \not\models L$. Thus, they capture negative examples as negative facts which should be entailed from a program, while we capture negative examples as positive/negative facts which should *not* be entailed from a program. In the

[5]Normal logic programs (including stratified programs) do not include negative literals, so that the condition of involvement is automatically satisfied (cf. footnote 2).

framework of Inoue and Kudoh [1997], Lamma et al. [2000], and Nicolas and Duval [2001], however, the distinction between positive and negative examples no longer exists, since in ELPs and default theories positive and negative literals have equal standings with respect to provability. In fact, their definition of induction from negative examples is identical to the definition of induction from positive examples. Our problem setting is based on the usual one in the clausal ILP [Muggleton and De Raedt 1994; Nienhuys-Cheng and De Wolf 1997; Flach 2000] in which negative examples are defined as evidences which are not entailed by the new knowledge base after induction. Note that we do not assume the closed world assumption in our problem setting; hence, there is a room for the existence of facts that are unknown to be true/false. On the other hand, our method is also used for induction from negative examples which is considered in Inoue and Kudoh [1997], Lamma et al. [2000], and Nicolas and Duval [2001] by applying the algorithm IAS^{pos} to negative examples represented by negative literals (see the introductory example in Section 3.3).

Dimopoulos and Kakas [1995] construct default rules with exceptions. They represent exceptions by a prioritized hierarchy without using NAF and use an ordinary ILP algorithm for learning, which is in contrast to our algorithms that learn nonmonotonic rules with NAF. Bergadano et al. [1996] propose a system for learning normal logic programs, but it selects hypotheses from candidates given in input to the system and does not construct hypotheses. For other systems, Quinlan [1990] provides an induction algorithm which learns definite clauses possibly containing negation in its body. Martin and Vrain [1996] introduce an algorithm for learning NLPs under the 3-valued semantics. Fogel and Zaverucha [1998] learn strict and call-consistent NLPs using subsumption and iteratively constructed training examples. Algorithms of Quinlan [1990], Martin and Vrain [1996], and Fogel and Zaverucha [1998] construct hypotheses in a top-down manner, that is, generating hypotheses one after another from general one to specific one. By contrast, our algorithms are bottom-up, that is, a specific hypothesis is firstly constructed by an expansion set and it is subsequently generalized. Generally, top-down algorithms are liable to produce a relatively large hypotheses space without a guide for intelligent search.

5.2 Induction from Stable Models

Seitzer [1997] proposes a system called *INDED*. It consists of a deductive engine which computes the stable models or the well-founded model of a background normal logic program, and an inductive engine which empirically constructs hypotheses using the computed models and positive/negative examples. In the inductive part, the algorithm constructs hypotheses using a generic top-down manner, which is in contrast to the bottom-up method considered in this article. The top-down induction algorithm is an ordinary one used in Horn ILP.

Otero [2001] characterizes induction problems in normal logic programs under the stable model semantics. He considers minimal and most specific solutions by introducing facts to a program. For instance, in Example 3.2,

$R = \mathit{flies}(\mathit{tweety}) \leftarrow$ is the most specific solution to satisfy $P \cup \{R\} \models L$. He also considers *induction from several sets of examples* such that: given an NLP P and several sets of examples $E_1, \ldots, E_n$ where $E_i = E_i^+ \cup E_i^-$ $(1 \leq i \leq n)$, H is a solution of induction if there is a stable model M_i of $P \cup H$ such that $M_i \models E_i^+$ and $M_i \not\models E_i^-$ for each E_i. He then concludes that nonmonotonic hypotheses are only truly needed when the problem is induction from several sets of examples. We do not consider problems of this type in this article. However, his solution for a single set of examples is not a "usual" solution of induction. In fact, the goal of induction is to explain examples not by just introducing facts, but by constructing general rules. His solution is considered *abduction* rather than induction.

5.3 Nonmonotonic Induction

In the field of ILP, it is often considered the so-called *nonmonotonic problem setting* [Helft 1989]. Given a background Horn logic program P and a set E of positive examples, it computes a hypothesis H which is satisfied in the least Herbrand model of $P \cup E$. This is also called the *weak* setting of ILP [De Raedt and Lavrač 1993]. In this setting, any fact which is not derived from $P \cup E$ is assumed to be false under the *closed world assumption* (CWA). By contrast, the *strong* setting of ILP computes a hypothesis H which, together with P, implies E, and does not imply negative examples. The strong setting is widely explored in ILP and is also considered in this article.[6] The nonmonotonic setting is called "nonmonotonic" in the sense that it performs a kind of default reasoning based on the CWA. Some systems take similar approaches using *Clark's completion* (e.g., De Raedt and Bruynooghe [1993]). The above-mentioned nonmonotonic setting is clearly different from our problem setting. They still capture an induction problem within clausal logic, while we consider the problem in nonmonotonic logic programs.

5.4 Inverse Entailment

Given the background Horn logic program B and a positive example E, *inverse entailment* (IE) [Muggleton 1995] is based on the idea that a possible hypothesis H satisfying the relation

$$B \cup \{H\} \models E$$

is deductively constructed from B and E by inverting the entailment relation as

$$B \cup \{\neg E\} \models \neg H.$$

When a background theory is a nonmonotonic logic program, however, the IE technique cannot be used. This is because IE is based on the *deduction theorem* in first-order logic, but it is known that the deduction theorem does not hold in nonmonotonic logics in general [Shoham 1987]. Then, Sakama [2000] reconstructs a theory of IE in nonmonotonic logic programs and introduces the

[6]The weak setting is also called *descriptive/confirmatory induction*, while the strong setting is called *explanatory/predictive induction* [Flach 2000].

inverse entailment theorem in normal logic programs under the stable model semantics as follows:

THEOREM 5.1 (SAKAMA 2000). *Let P be an NLP and R a rule such that $P \cup \{R\}$ is consistent. For any ground LP-literal L, $P \models not\ L$ and $P \cup \{R\} \models L$ imply $P \models not\ R$.*

Here, L is either A or *not* A for a ground atom A. And $P \models not\ L$ means that L is false in every stable model of P, and $P \models not\ R$ means that R is satisfied in no stable model of P. Theorem 5.1 is close to Propositions 3.1 and 3.7 in this article, while there is a subtle difference between them. For the case of induction from a positive example, this article assumes the initial condition $P \not\models L$, then $P \cup \{R\} \models L$ implies $P \not\models R$. When P is an NLP, the condition $P \not\models L$ presents that L is false in *some* stable model of P, while $P \models not\ L$ presents that L is false in *every* stable model of P. The former condition is weaker than the latter one in the sense that the former presents the falsity of the example in P under *credulous* inference, while the latter is under *skeptical* inference. As a result, the weaker condition implies $P \not\models R$, and the stronger one implies $P \models not\ R$. These two conditions coincide when a program is categorical. Similar arguments are done for the case of induction from a negative example. Compared with Sakama [2000], this paper formalizes the result for a wider class of programs in a much simpler manner.

Muggleton [1998] proposes a modified version of IE which is complete for building hypotheses in Horn theories. It constructs a hypothetical clause using the *enlarged bottom set* which is the least Herbrand model of a background program augmented by closed world negation. In a Horn logic program, the expansion set S^+ considered in this article has an effect similar to the enlarged bottom set. However, Muggleton does not provide a condition for the correctness nor handle induction from negative examples.

5.5 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; Lifschitz 2002]. In the presence of negation as failure, a logic program generally has multiple intended models, which is in contrast to the single least Herbrand model in a Horn logic program. ASP views a program as a set of constraints that every solution should satisfy, then it extracts solutions from the collection of answer sets of the program. We constructed inductive hypotheses from answer sets where a background theory and examples work as constraints which inductive hypotheses should satisfy. Thus, induction problems in nonmonotonic logic programs are captured as a problem of ASP. The result 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; Eiter et al. 1999; Sakama and

Inoue 1999]. By contrast, in induction hypothetical *rules* that 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 exploits new application of ASP.

6. CONCLUSION

Induction is a nonmonotonic inference from incomplete knowledge, while techniques in ILP have been centered on monotonic clausal logic so far, especially on Horn logic. To enrich the language and to enhance the reasoning capability, extending the ILP framework to nonmonotonic logic programs is an important step towards a better learning tool in AI. To achieve the goal, this article built a theory of induction from nonmonotonic logic programs. We introduced algorithms for constructing new rules from the answer sets of an extended logic program, and provided conditions to explain positive and negative examples. The proposed method extends the ILP framework to a syntactically and semantically richer framework, and contributes to a theory of induction in nonmonotonic logic programs.

Commonsense reasoning and machine learning are indispensable for realizing intelligent information systems. Then, combining techniques of NMLP and ILP in the context of *nonmonotonic inductive logic programming* (NMILP) is meaningful and important [Sakama 2001]. Such combination will extend the representation language on the ILP side, while it will introduce a learning mechanism to programs on the NMLP side. From the practical viewpoint, such combination will be beneficial for ILP to use well-established techniques of NMLP. In fact, our induction algorithms construct hypotheses using answer sets, so that it is realized on top of the existing procedures for answer set programming.[7] In contrast to clausal ILP, the field of NMILP is less explored and several issues remain open. On the theoretical side, we used the answer set semantics in this article, while a different theory of induction would be constructed using other NMLP semantics. On the practical side, NAF is useful to express exceptions, and induction of nonmonotonic default rules would have an important application in the field of *data mining*. These issues have yet to be investigated.

ACKNOWLEDGMENTS

The author thanks the anonymous referees for their valuable comments.

REFERENCES

BAIN, M. AND MUGGLETON, S. 1992. Non-monotonic learning. In *Inductive Logic Programming*, S. Muggleton, Ed. Academic Press, Orlando, Fla., 145–161.

BARAL, C. AND GELFOND, M. 1994. Logic programming and knowledge representation. *J. Logic Prog. 19/20*, 73–148.

[7]The algorithm IAS^{pos} of induction from a positive example is implemented by Guray Alsac at Arizona State University. The system is available at `http://www.wakayama-u.ac.jp/~sakama/IAS`.

BERGADANO, F., GUNETTI, D., NICOSIA, M., AND RUFFO, G. 1996. Learning logic programs with negation as failure. In *Advances in Inductive Logic Programming*, L. De. Raedt, Ed. IOS Press, 107–123.

BREWKA, G. AND DIX, J. 1997. Knowledge representation with logic programs. In *Proceedings of the 3rd Workshop on Logic Programming and Knowledge Representation*. Lecture Notes in Artificial Intelligence, vol. 1471. Springer-Verlag, New York, 1–51.

CAI, Y., CERCONE, N., AND HAN, J. 1991. Attribute-oriented induction in relational databases. In *Knowledge Discovery in Databases*, G. P.-Shapiro and W. J. Frawley, Eds. AAAI Press, 213–228.

DE RAEDT, L. AND BRUYNOOGHE, M. 1990. On negation and three-valued logic in interactive concept learning. In *Proceedings of the 9th European Conference on Artificial Intelligence*. Pitman, 207–212.

DE RAEDT, L. AND BRUYNOOGHE, M. 1993. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Morgan-Kaufmann, San Francisco, Calif., 1058–1063.

DE RAEDT, L. AND LAVRAČ, N. 1993. The many faces of inductive logic programming. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*. Lecture Notes in Artificial Intelligence, vol. 689. Springer-Verlag, New York, 435–449.

DIMOPOULOS, Y. AND KAKAS, A. 1995. Learning nonmonotonic logic programs: learning exceptions. In *Proceedings of the 8th European Conference on Machine Learning*. Lecture Notes in Artificial Intelligence, vol. 912. Springer-Verlag, New York, 122–137.

EITER, T., FABER, W., LEONE, N., AND PFEIFER, G. 1999. The diagnosis frontend of the dlv system. *AI Commun. 12*, 99–111.

FLACH, P. A. 2000. Logical characterisations of inductive learning. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, D. M. Gabbay and R. Kruse, Eds., Vol. 4. Kluwer Academic Publishers, 155–196.

FOGEL, L. AND ZAVERUCHA, G. 1998. Normal programs and multiple predicate learning. In *Proceedings of the 8th International Workshop on Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 1446. Springer-Verlag, New York, 175–184.

GABBAY, D., GILLIES, D., HUNTER, A., MUGGLETON, S., NG, Y., AND RICHARDS, B. 1992. The rule-based systems project: using confirmation theory and non-monotonic logics for incremental learning. In *Inductive Logic Programming*, S. Muggleton, Ed. Academic Press, Orlando, Fla., 213–230.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*. MIT Press, Cambridge, Mass., 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1990. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 579–597.

HELFT, N. 1989. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, San Francisco, Calif., 149–156.

INOUE, K. AND KUDOH, Y. 1997. Learning extended logic programs. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*. Morgan-Kaufmann, San Francisco, Calif., 176–181.

INOUE, K. AND SAKAMA, C. 1996. A fixpoint characterization of abductive logic programs. *J. Logic Prog. 27*, 2, 107–136.

KAKAS, A. C., KOWALSKI, R. A., AND TONI, F. 1998. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Vol. 5. Oxford University Press, 235–324.

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

LAMMA, E., RIGUZZI, F., AND PEREIRA, L. M. 2000. Strategies in combined learning via logic programs. *Mach. Learn. 38*, 1/2, 63–87.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Artif. Intel. 138*, 39–54.

MAREK, V. W. AND TRUSZCZYŃSKI, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm—A 25 Year Perspective*, K. R. Apt, et al., Ed. Springer-Verlag, New York, 375–398.

MARTIN, L. AND VRAIN, C. 1996. A three-valued framework for the induction of general logic programs. In *Advances in Inductive Logic Programming*, L. De. Raedt, Ed. IOS Press, 219–235.

MUGGLETON, S., Ed. 1992. *Inductive Logic Programming*. Academic Press, Orlando, Fla.

MUGGLETON, S. 1995. Inverse entailment and progol. *New Gen. Comput. 13*, 245–286.

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

MUGGLETON, S. AND BUNTINE, W. 1992. Machine invention of first-order predicate by inverting resolution. In *Inductive Logic Programming*, S. Muggleton, Ed. Academic Press, Orlando, Fla., 261–280.

MUGGLETON, S. AND DE RAEDT, L. 1994. Inductive logic programming: theory and methods. *J. Logic Prog. 19/20*, 629–679.

NICOLAS, P. AND DUVAL, B. 2001. Representation of incomplete knowledge by induction of default theories. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Artificial Intelligence, vol. 2173. Springer-Verlag, New York, 160–172.

NIEMELA, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intel. 25*, 241–273.

NIENHUYS-CHENG, S.-H. AND DE WOLF, R. 1997. *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 1228. Springer-Verlag, New York.

OTERO, R. P. 2001. Induction of stable models. In *Proceedings of the 11th International Conference on Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 2157. Springer-Verlag, New York, 193–205.

PRZYMUSINSKI, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan-Kaufmann, San Francisco, Calif., 193–216.

QUINLAN, J. R. 1990. Learning logical definitions from relations. *Mach. Learn. 5*, 239–266.

REITER, R. 1980. A logic for default reasoning. *Artif. Intel. 13*, 81–132.

RICHARDS, B. L. AND MOONEY, R. J. 1995. Automated refinement of first-order Horn-clause domain theories. *Mach. Learn. 19*, 2, 95–131.

SAKAMA, C. 1999. Some properties of inverse resolution in normal logic programs. In *Proceedings of the 9th International Workshop on Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 1634. Springer-Verlag, New York, 279–290.

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

SAKAMA, C. 2001. Nonmonotonic inductive logic programming. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Artificial Intelligence, vol. 2173. Springer-Verlag, New York, 62–80.

SAKAMA, C. 2001a. Learning by answer sets. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming*. AAAI Press, 181–187.

SAKAMA, C. AND INOUE, K. 1999. Updating extended logic programs through abduction. In *Proceedings of the 5th International Conference Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Artificial Intelligence, vol. 1730. Springer-Verlag, New York, 147–161.

SCHLIPF, J. S. 1995. Complexity and undecidability results for logic programming. *Ann. Math. Artif. Intel. 15*, 257–288.

SEITZER, J. 1997. Stable ILP: Exploring the added expressivity of negation in the background knowledge. In *Proceedings of IJCAI-97 Workshop on Frontiers of ILP*.

SHOHAM, Y. 1987. Nonmonotonic logics: meaning and utility. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan-Kaufmann, San Francisco, Calif., 388–393.

Received November 2001; revised November 2001; accepted August 2003