Nonmonotonic Inductive Logic Programming

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. *Nonmonotonic logic programming* (NMLP) and *inductive logic programming* (ILP) are two important extensions of logic programming. The former aims at representing incomplete knowledge and reasoning with commonsense, while the latter targets the problem of inductive construction of a general theory from examples and background knowledge. NMLP and ILP thus have seemingly different motivations and goals, but they have much in common in the background of problems, and techniques developed in each field are related to one another. This paper presents techniques for combining these two fields of logic programming in the context of *nonmonotonic inductive logic programming* (NMILP). We review recent results and problems to realize NMILP.

1 Introduction

Representing knowledge in computational logic gives formal foundations of artificial intelligence (AI) and provides computational methods for solving problems. Logic programming supplies a powerful tool for representing declarative knowledge and computing logical inference. However, logic programming based on classical Horn logic is not sufficiently expressive for representing incomplete human knowledge, and is inadequate for characterizing nonmonotonic commonsense reasoning. *Nonmonotonic logic programming* (NMLP) [3, 5] is introduced to overcome such limitations of Horn logic programming by extending the representation language and enhancing the inference mechanism. The purpose of NMLP is to represent incomplete knowledge and reason with commonsense in a program.

On the other hand, *machine learning* concerns with the problem of building computer programs that automatically construct new knowledge and improve with experience [27]. The primary inference used in learning is *induction* which constructs general sentences from input examples. *Inductive Logic Programming* (ILP) [28, 30, 33] realizes inductive machine learning in logic programming, which provides a formal background to inductive learning and has advantages of using computational tools developed in logic programming. The goal of ILP is the inductive construction of first-order clausal theories from examples and background knowledge.

NMLP and ILP thus have seemingly different motivations and goals, however, they have much in common in the background of problems, and techniques developed in each field are related to one another. First, the process of discovering new knowledge by humans is the iteration of hypotheses generation and revision, which is inherently nonmonotonic. Indeed, induction is nonmonotonic reasoning in the sense that once induced hypotheses might be changed by the introduction of new evidences. Second, 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. Third, NMLP uses hypotheses in the process of commonsense reasoning, and hypotheses generation is particularly important in *abductive logic programming*. Abduction generates hypotheses in a different manner from induction, but they are both inverse deduction and extend theories to account for evidences. Indeed, abduction and induction interact, and work complementarily in many phases [14]. Fourth, in NMLP updates of general rules are considered in the context of *intentional knowledge base update* [6], while a similar problem is captured in ILP as *concept-learning* [26]. It is argued in [9] that these two researches handle the same problem when formulated in a logical framework. With these reasons, it is clear that both NMLP and ILP cope with similar problems and have close links to each other.

Comparing NMLP and ILP, NMLP performs default reasoning and derives plausible conclusions from incomplete knowledge bases. Various types of inferences and semantics are introduced to extract intuitive conclusions from a program. NMLP may change conclusions by the introduction of new information, but it has no mechanism of learning new knowledge from the input. By contrast, ILP extends a theory by constructing new rules from input examples and background knowledge. Discovered rules reveal hidden laws between examples and background knowledge, and are also used for predicting unseen phenomena. However, the present ILP mostly considers Horn logic programs or classical clausal programs as background knowledge, and has limited applications to nonmonotonic situations.

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 realizing intelligent information systems, combining techniques of the two fields in the context of *nonmonotonic inductive logic programming* (NMILP) is meaningful and 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. Moreover, linking different extensions of logic programming will strengthen the capability of logic programming as a knowledge representation tool in AI. From the practical viewpoint, the combination will be beneficial for ILP to use well-established techniques in NMLP, and will open new applications of NMLP.

NMLP realizes nonmonotonic reasoning using *negation as failure* (NAF). Some researches in ILP, however, argue that negation as failure is inappropriate in machine learning. In [8], the authors 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 the account is plausible, it does not justify excluding NAF in ILP. Suppose that background knowledge is given as a Horn logic program, and the CWA or NAF infers negative facts which are not derived from the program. When a new evidence E which is initially assumed false under the CWA or 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. On the other hand, if one excludes NAF in a background program, she loses the way of representing default negation in the program. This is a significant drawback in representing knowledge and restricts the application of ILP. In fact, NAF enables to write shorter and simpler programs and appears in many basic but practical Prolog programs such as computing set differences, finding union/intersection of two lists, etc [42]. Horn ILP precludes every program including these rules with NAF. Thus, NAF is also important in ILP, and the use of NAF never invalidates the need of learning.

In the field of ILP, it is often considered the so-called *nonmonotonic problem setting* [18]. 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 [11]. 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 usually employed in ILP and is also considered in this paper (see Section 2.2).[1] The nonmonotonic setting is called "nonmonotonic" in the sense that it performs a kind of default reasoning based on the closed world assumption. Some systems take similar approaches using Clark's completion ([10], for instance). The above mentioned nonmonotonic setting is clearly different from our problem setting. The former still considers an induction problem within clausal logic, while we extend the problem to nonmonotonic logic programs.

This paper presents techniques for realizing inductive machine learning in nonmonotonic logic programs. The paper is not intended to provide a comprehensive survey of the state of the art, but mainly consists of recent research results of the author. The rest of this paper is organized as follows. Section 2 reviews frameworks of NMLP and ILP. Section 3 presents various techniques for induction in nonmonotonic logic programs. Section 4 summarizes the paper and addresses open issues.

[1] The weak setting is also called *descriptive/confirmatory induction*, while the strong setting is called *explanatory/predictive induction* [15].

2 Preliminaries

2.1 Nonmonotonic Logic Programming

Nonmonotonic logic programs considered in this paper are *normal logic programs*, logic programs with negation as failure.

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

$$A \leftarrow B_1, \ldots, B_m, \mathit{not}\, B_{m+1}, \ldots, \mathit{not}\, B_n \tag{1}$$

where each A, B_i $(1 \leq i \leq n)$ is an atom and *not* presents *negation as failure* (NAF). The left-hand side of $\leftarrow$ is the *head*, and the right-hand side is the *body* of the rule. The conjunction in the body of (1) is identified with the set $\{ B_1, \ldots, B_m, \mathit{not}\, B_{m+1}, \ldots, \mathit{not}\, B_n \}$. For a rule R, $\mathit{head}(R)$ and $\mathit{body}(R)$ denote the head of R and the body of R, respectively. The conjunction in the body is often written by the Greek letter Γ. A rule with the empty body $A \leftarrow$ is called a *fact*, which is identified with the atom A. A rule with the empty head $\leftarrow \Gamma$ with $\Gamma \neq \emptyset$ is also called an *integrity constraint*. Throughout the paper a program means a normal logic program unless stated otherwise. A program P is *Horn* if no rule in P contains NAF. A Horn program is *definite* if it contains no integrity constraint. The *Herbrand base* $\mathcal{HB}$ of a program P is the set of all ground atoms in the language of P. Given the Herbrand base $\mathcal{HB}$, we define $\mathcal{HB}^+ = \mathcal{HB} \cup \{ \mathit{not}\, A \mid A \in \mathcal{HB} \}$. Any element in $\mathcal{HB}^+$ is called an *LP-literal*, and an LP-literal of the form $\mathit{not}\, A$ is called an *NAF-literal*. We say that two LP-literals L_1 and L_2 have the same *sign* if either ($L_1 \in \mathcal{HB}$ and $L_2 \in \mathcal{HB}$) or ($L_1 \notin \mathcal{HB}$ and $L_2 \notin \mathcal{HB}$). For an LP-literal L, $\mathit{pred}(L)$ denotes the predicate in L and $\mathit{const}(L)$ denotes the set of constants appearing in L. A program, a rule, or an LP-literal is *ground* if it contains no variable. A program/rule containing variables is semantically identified with its ground instantiation, i.e., the set of ground rules obtained from the program/rule by substituting variables with elements of the Herbrand universe in every possible way.

An *interpretation* is a subset of $\mathcal{HB}$. An interpretation I *satisfies* the ground rule R of the form (1) if $\{B_1, \ldots, B_m\} \subseteq I$ and $\{B_{m+1}, \ldots, B_n\} \cap I = \emptyset$ imply $A \in I$ (written as $I \models R$). In particular, I satisfies the ground integrity constraint $\leftarrow B_1, \ldots, B_m, \mathit{not}\, B_{m+1}, \ldots, \mathit{not}\, B_n$ if either $\{B_1, \ldots, B_m\} \setminus I \neq \emptyset$ or $\{B_{m+1}, \ldots, B_n\} \cap I \neq \emptyset$. When a rule R contains variables, $I \models R$ means that I satisfies every ground instance of R. An interpretation which satisfies every rule in a program is a *model* of the program. A model M of a program P is *minimal* if there is no model N of P such that $N \subset M$. A Horn logic program has at most one minimal model called the *least model*.

For the semantics of NLPs, we consider the *stable model semantics* [17] in this paper. Given a program P and an interpretation M, the ground Horn logic program P^M is defined as follows: the rule $A \leftarrow B_1, \ldots, B_m$ is in P^M iff there is a ground rule of the form (1) in the ground instantiation of P such that $\{B_{m+1}, \ldots, B_n\} \cap M = \emptyset$. If the least model of P^M is identical to M, M is called a *stable model* of P. A program may have none, one, or multiple stable models in general. A program having exactly one stable model is called *categorical* [3].

A stable model coincides with the least model in a Horn logic program. A *locally stratified program* [36] has the unique stable model which is called the *perfect model*. Given a stable model M, we define $M^+ = M \cup \{\, not\, A \mid A \in \mathcal{HB} \setminus M \,\}$.

A program is *consistent* (under the stable model semantics) if it has a stable model; otherwise a program is *inconsistent*. Throughout the paper, a program is assumed to be consistent unless stated otherwise. If every stable model of a program P satisfies a rule R, it is written as $P \models_s R$. Else if no stable model of a program P satisfies a rule R, it is written as $P \models_s not\, R$. In particular, $P \models_s A$ if a ground atom A is true in every stable model of P; and $P \models_s not\, A$ if A is false in every stable model of P. By contrast, if every model of P satisfies R, it is written as $P \models R$. Note that when P is Horn, the meaning of $\models$ coincides with the classical entailment.

2.2 Inductive Logic Programming

A typical ILP problem is stated as follows. Given a logic program B representing background knowledge and a set E^+ of positive examples and a set E^- of negative examples, find hypotheses H satisfying[2]

1. $B \cup H \models e$ for every $e \in E^+$.
2. $B \cup H \not\models f$ for every $f \in E^-$.
3. $B \cup H$ is consistent.

The first condition is called *completeness* with respect to positive examples, and the second condition is called *consistency* with respect to negative examples. It is also implicitly assumed that $B \not\models e$ for some $e \in E^+$ or $B \models f$ for some $f \in E^-$, because otherwise there is no need to introduce H. A hypothesis H *covers* (resp. *uncovers*) an example e if $B \cup H \models e$ (resp. $B \cup H \not\models e$).

The goal of ILP is then to develop an algorithm which efficiently computes hypotheses satisfying the above three conditions. Induction algorithms are roughly classified into two categories by the direction of searching hypotheses. A *top-down* algorithm firstly generates a most general hypothesis and refines it by means of specialization, while a *bottom-up* algorithm searches hypotheses by generalizing (positive) examples. Each algorithm locally alternates search directions from general to specific and vice versa to correct hypotheses. Algorithms presented in Sections 3.1–3.3 of this paper are bottom-up on this ground.

An induction algorithm is *correct* if every hypothesis produced by the algorithm satisfies the above three conditions. By contrast, an induction algorithm is *complete* if it produces every rule satisfying the conditions. Note that the correctness is generally requested for algorithms, while the completeness is problematic in practice. For instance, consider the background program B and the positive example E such that

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

[2] When there is no negative example, E^+ is just written as E.

Then, any of the following rules

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

explains $p(a)$. Generally, there exist possibly infinite solutions for explaining an example, and designing a complete induction algorithm without any restriction is of little value in practice. In order to extract meaningful hypotheses, additional conditions are usually imposed on possible hypotheses to reduce the search space. Such a condition is called an *induction bias* and is defined as any information that syntactically or semantically influences learning processes.

In the field of ILP, most studies consider a Horn logic program as background knowledge and induce Horn clauses as hypotheses. In this paper, we consider an NLP as background knowledge and induce hypothetical rules possibly containing NAF. In the next section, we give several algorithms which realize this.

3 Induction in Nonmonotonic Logic Programs

3.1 Least Generalization

Generalization is a basic operation to perform induction. In his seminal work [34], Plotkin introduces generalization in clausal theories based on *subsumption*. Given two clauses C_1 and C_2, C_1 *θ-subsumes* C_2 if $C_1\theta \subseteq C_2$ for some substitution θ. Then, C_1 is *more general than C_2 under θ-subsumption* if C_1 θ-subsumes C_2. In normal logic programs, a subsumption relation between rules is defined as follows.

Definition 3.1. (subsumption relations between rules) Let R_1 and R_2 be two rules. Then, R_1 *θ-subsumes* R_2 (written as $R_1 \succeq_\theta R_2$) if $head(R_1)\theta = head(R_2)$ and $body(R_1)\theta \subseteq body(R_2)$ hold for some substitution θ. In this case, R_1 is said *more general than R_2 under θ-subsumption*.

Thus subsumption is defined for comparison of rules with the same predicate in the heads. The same definition is employed by Taylor [43]. Fogel and Zaverucha [16] discuss the effect of subsumption to reduce the search space in normal logic programs.

For generalization in clausal theories, *least generalizations* of clauses are particularly important. The notion is defined for nonmonotonic rules as follows.

Definition 3.2. (least generalization under subsumption) Let $\mathcal{R}$ be a finite set of rules such that every rule in $\mathcal{R}$ has the same predicate in the head. Then, a rule R is a *least generalization of* $\mathcal{R}$ under θ-subsumption if $R \succeq_\theta R_i$ for every rule R_i in $\mathcal{R}$, and for any other rule R' satisfying $R' \succeq_\theta R_i$ for every R_i in $\mathcal{R}$, it holds that $R' \succeq_\theta R$.

In the clausal language every finite set of clauses has a least generalization. In particular, every finite set of Horn clauses has a least generalization as a Horn clause [33, 34].[3] When we consider normal logic programs, rules are syntactically regarded as Horn clauses by viewing NAF-literal $not\, p(x)$ as an atom $not\_p(x)$ with the new predicate not_p. Then the result of Horn logic programs is directly carried over to normal logic programs.

Theorem 3.1. *(existence of a least generalization) Let $\mathcal{R}$ be a finite set of rules such that every rule in $\mathcal{R}$ has the same predicate in the head. Then, every non-empty set $R \subseteq \mathcal{R}$ has a least generalization under θ-subsumption.*

A least generalization of two rules is computed as follows. First, a least generalization of two terms $f(t_1, \ldots, t_n)$ and $g(s_1, \ldots, s_n)$ is a new variable v if $f \neq g$; and is defined as $f(lg(t_1, s_1), \ldots, lg(t_n, s_n))$ if $f = g$, where $lg(t_i, s_i)$ means a least generalization of t_i and s_i. Next, a least generalization of two LP-literals $L_1 = (not)p(t_1, \ldots, t_n)$ and $L_2 = (not)q(s_1, \ldots, s_n)$ is undefined if L_1 and L_2 do not have the same predicate and sign; otherwise, it is defined as $lg(L_1, L_2) = (not)p(lg(t_1, s_1), \ldots, lg(t_n, s_n))$.

Then, a least generalization of two rules $R_1 = A_1 \leftarrow \Gamma_1$ and $R_2 = A_2 \leftarrow \Gamma_2$, where A_1 and A_2 have the same predicate, is obtained as

$$lg(A_1, A_2) \leftarrow \Gamma$$

where $\Gamma = \{\, lg(\gamma_1, \gamma_2) \mid \gamma_1 \in \Gamma_1,\ \gamma_2 \in \Gamma_2 \text{ and } lg(\gamma_1, \gamma_2) \text{ is defined} \,\}$. In particular, if A_1 and A_2 are empty, a least generalization of two integrity constraints $\leftarrow \Gamma_1$ and $\leftarrow \Gamma_1$ is given by $\leftarrow \Gamma$. A least generalization of a finite set of rules is computed by repeatedly applying the above procedure.

In ILP generalization is usually considered in relation to the background knowledge. Plotkin [35] extends subsumption to *relative subsumption* for this use. Given the background knowledge B as a clausal theory, a clause C *subsumes* D *relative to* B if there is a substitution θ such that $B \models \forall(C\theta \rightarrow D)$.

We apply relative subsumption to normal logic programs. Let $R = H \leftarrow A, \Gamma$ be a rule where A is an atom and Γ is a conjunction. Suppose that there is a rule $A' \leftarrow \Gamma'$ in a program P such that $A\theta = A'\theta$ for some substitution θ. Then, we say that the rule $(H \leftarrow \Gamma', \Gamma)\theta$ is obtained by *unfolding* R in P. We also say that R_k is obtained by unfolding R_0 in P if there is a sequence $R_0, \ldots, R_k$ of rules such that R_i $(1 \leq i \leq k)$ is obtained by unfolding R_{i-1} in P.

Definition 3.3. (relative subsumption) Let P be an NLP, and R_1 and R_2 be two rules. Then, R_1 *θ-subsumes R_2 relative to P* (written as $R_1 \succeq^P_\theta R_2$) if there is a rule R that is obtained by unfolding R_1 in P and R θ-subsumes R_2. In this case, R_1 is said *more general than R_2 relative to P under θ-subsumption.*

The above definition reduces to Definition 3.1 when P is empty. By the definition relative subsumption is also defined for two rules having the same

[3] If two clauses have no predicate with the same sign in common, the empty clause becomes the least generalization.

predicate in the heads. In clausal theories, Buntine [7] introduces *generalized subsumption* which is defined between definite clauses having the same predicate in the heads. Comparing two definitions, Buntine's definition is model theoretic, while our definition is operational. Taylor [43] introduces *normal subsumption* which extends Buntine's subsumption to normal logic programs and is defined in a model theoretic manner.

Example 3.1. Suppose the background program P, and two rules R_1 and R_2 as follows.

$$\begin{aligned} P:\ & has\_wing(x) \leftarrow bird(x), not\, ab(x), \\ & bird(x) \leftarrow sparrow(x), \\ & ab(x) \leftarrow broken\text{-}wing(x). \\ R_1:\ & flies(x) \leftarrow has\_wing(x). \\ R_2:\ & flies(x) \leftarrow sparrow(x), full\_grown(x), not\, ab(x). \end{aligned}$$

From P and R_1, the rule

$$R_3:\ flies(x) \leftarrow sparrow(x), not\, ab(x)$$

is obtained by unfolding. As R_3 θ-subsumes R_2, $R_1 \succeq_\theta^P R_2$.

In clausal theories, a least generalization does not always exist under relative subsumption. However, when background knowledge is a finite set of ground atoms, a least generalization of two clauses is constructed [33, 35]. The result is extended to nonmonotonic rules and is rephrased in our context as follows. Let P be a finite set of ground atoms, and R_1 and R_2 be two rules. Then, a least generalization of these rules under relative subsumption is constructed as a least generalization of R_1' and R_2' where $head(R_i') = head(R_i)$ and $body(R_i') = body(R_i) \cup B$.

Example 3.2. Suppose the background program P, and two (positive) examples R_1 and R_2 as follows.

$$\begin{aligned} P:\ & bird(tweety) \leftarrow, \quad bird(polly) \leftarrow . \\ R_1:\ & flies(tweety) \leftarrow has\_wing(tweety), not\, ab(tweety). \\ R_2:\ & flies(polly) \leftarrow sparrow(polly), not\, ab(polly). \end{aligned}$$

Then, R_1' and R_2' becomes

$$\begin{aligned} R_1':\ & flies(tweety) \leftarrow bird(tweety), bird(polly), has\_wing(tweety), not\, ab(tweety), \\ R_2':\ & flies(polly) \leftarrow bird(tweety), bird(polly), sparrow(polly), not\, ab(polly). \end{aligned}$$

The least generalization of R_1' and R_2' is

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

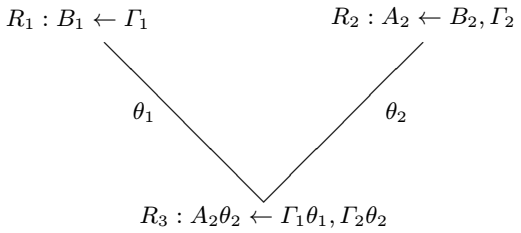
Removing redundant literals, it becomes

$$R:\ flies(x) \leftarrow bird(x), not\, ab(x).$$

In this case, it holds that $P \cup \{R\} \models_s R_i$ $(i = 1, 2)$.

3.2 Inverse Resolution

Inverse resolution [29] is based on the idea of inverting the resolution step between clauses. There are two operators that carry out inverse resolution, *absorption* and *identification*, which are called the *V-operators* together. Each operator builds one of the two parent clauses given the other parent clause and the resolvent. Suppose two rules $R_1 : B_1 \leftarrow \Gamma_1$ and $R_2 : A_2 \leftarrow B_2, \Gamma_2$. When $B_1\theta_1 = B_2\theta_2$, the rule $R_3 : A_2\theta_2 \leftarrow \Gamma_1\theta_1, \Gamma_2\theta_2$ is produced by unfolding R_2 with R_1. Absorption constructs R_2 from R_1 and R_3, while identification constructs R_1 from R_2 and R_3 (see figure).



Given a normal logic program P containing the rules R_1 and R_3, absorption produces the program $A(P)$ such that

$$A(P) = (P \setminus \{R_3\}) \cup \{R_2\}\,.$$

On the other hand, given an NLP P containing the rules R_2 and R_3, identification produces the program $I(P)$ such that

$$I(P) = (P \setminus \{R_3\}) \cup \{R_1\}\,.$$

Note that there are multiple $A(P)$ or $I(P)$ exist in general according to the choice of the input rules in P. We write $V(P)$ to mean either $A(P)$ or $I(P)$.

When P is a Horn logic program, any information implied by P is also implied by $V(P)$, namely

$$V(P) \models P\,.$$

In this regard, the V-operators generalize a Horn logic program. In the presence of negation as failure in a program, however, the V-operators do not work as generalization operations in general.

Example 3.3. Let P be the program:

$$p(x) \leftarrow not\, q(x), \quad q(x) \leftarrow r(x), \quad s(x) \leftarrow r(x), \quad s(a) \leftarrow,$$

which has the stable model $\{\, p(a), s(a) \,\}$. Absorbing the third rule into the second rule produces $A(P)$:

$$p(x) \leftarrow not\, q(x), \quad q(x) \leftarrow s(x), \quad s(x) \leftarrow r(x), \quad s(a) \leftarrow,$$

which has the stable model $\{\, q(a), s(a) \,\}$. Then, $P \models_s p(a)$ but $A(P) \not\models_s p(a)$.

A counter-example for identification is constructed in a similar manner. The reason is clear, since in nonmonotonic logic programs newly proven facts may block the derivation of other facts which are proven beforehand. As a result, the V-operators may not generalize the original program. Moreover, the next example shows that the V-operators often make a consistent program inconsistent.

Example 3.4. Let P be the program:

$$p(x) \leftarrow q(x), \mathit{not}\, p(x), \quad q(x) \leftarrow r(x), \quad s(x) \leftarrow r(x), \quad s(a) \leftarrow,$$

which has the stable model $\{ s(a) \}$. Absorbing the third rule into the second rule produces $A(P)$:

$$p(x) \leftarrow q(x), \mathit{not}\, p(x), \quad q(x) \leftarrow s(x), \quad s(x) \leftarrow r(x), \quad s(a) \leftarrow,$$

which has no stable model.

The above example shows that the V-operators have destructive effect on the meaning of programs in general. It is also known that they may destroy the syntactic structure of programs such as acyclicity and local stratification [37].

These observations give us a caution to apply the V-operators to NMLP. A condition for the V-operators to generalize an NLP is as follows.

Theorem 3.2. *(conditions for the V-operators to generalize programs) [37] Let P be an NLP, and R_1, R_2, R_3 be rules at the beginning of this section. For any NAF-literal not L in P,*[4]

(i) if L does not depend on the head of R_3 in P, then $P \models_s N$ implies $A(P) \models_s N$ for any $N \in \mathcal{HB}$.
(ii) if L does not depend on the atom B_2 of R_2 in P, then $P \models_s N$ implies $I(P) \models_s N$ for any $N \in \mathcal{HB}$.

Example 3.5. Suppose the background program P and a (positive) example E as follows.

$$\begin{aligned} P:\ & \mathit{flies}(x) \leftarrow \mathit{sparrow}(x), \mathit{not}\, \mathit{ab}(x), \\ & \mathit{bird}(x) \leftarrow \mathit{sparrow}(x), \\ & \mathit{sparrow}(\mathit{tweety}) \leftarrow, \quad \mathit{bird}(\mathit{polly}) \leftarrow . \\ E:\ & \mathit{flies}(\mathit{polly}). \end{aligned}$$

Initially, $P \models_s \mathit{flies}(\mathit{tweety})$ but $P \not\models_s \mathit{flies}(\mathit{polly})$. Absorbing the second rule into the first rule in P produces the program $A(P)$ in which the first rule of P is replaced by the next rule in $A(P)$:

$$\mathit{flies}(x) \leftarrow \mathit{bird}(x), \mathit{not}\, \mathit{ab}(x).$$

Then, $A(P) \models_s \mathit{flies}(\mathit{polly})$. Notice that $A(P) \models_s \mathit{flies}(\mathit{tweety})$ also holds.

Taylor [43] introduces a different operator called *normal absorption*, which generalizes normal logic programs.

[4] Here, *depends on* is a transitive relation defined as: A depends on B if there is a ground rule from P s.t. A appears in the head and B appears in the body of the rule.

3.3 Inverse Entailment

Suppose an induction problem

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

where B is a Horn logic program and H and E are each single Horn clauses. *Inverse entailment* (IE) [31] is based on the idea that a possible hypothesis H 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 [41].

To solve the problem, Sakama [38] introduced the *entailment theorem* in normal logic programs. A *nested rule* is defined as

$$A \leftarrow R$$

where A is an atom and R is a rule of the form (1). An interpretation I satisfies a ground nested rule $A \leftarrow R$ if $I \models R$ implies $A \in I$. For an NLP P, $P \models_s (A \leftarrow R)$ if $A \leftarrow R$ is satisfied in every stable model of P.

Theorem 3.3. *(entailment theorem [38]) Let P be an NLP and R a rule such that $P \cup \{R\}$ is consistent. For any ground atom A, $P \cup \{R\} \models_s A$ implies $P \models_s A \leftarrow R$. In converse, $P \models_s A \leftarrow R$ and $P \models_s R$ imply $P \cup \{R\} \models_s A$.*

The entailment theorem corresponds to the deduction theorem and is used for inverting entailment in normal logic programs.

Theorem 3.4. *(IE in normal logic programs [38]) Let P be an NLP and R a rule such that $P \cup \{R\}$ is consistent. For any ground LP-literal L, if $P \cup \{R\} \models_s L$ and $P \models_s \leftarrow L$, then $P \models_s not\ R$.*

Thus, the relation

$$P \models_s not\ R \tag{2}$$

provides a necessary condition for computing a rule R satisfying $P \cup \{R\} \models_s L$ and $P \models_s \leftarrow L$. When L is an atom (resp. NAF-literal), it represents a positive (resp. negative) example. The condition $P \models_s \leftarrow L$ states that the example L is initially false in every stable model of P. To simplify the problem, a program P is assumed to be *function-free* and *categorical* in the rest of this section.

Given two ground LP-literals L_1 and L_2, the relation $L_1 \sim L_2$ is defined if $pred(L_1) = pred(L_2)$ with a predicate of arity ≥ 1 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) L_1 shares a constant with an LP-literal L_2 in S such that L_2 is relevant to L.

Let P be a program with the unique stable model M and A a ground atom representing a positive example. Suppose that the relation $P \cup \{R\} \models_s A$ and $P \models_s \leftarrow A$ hold. By Theorem 3.4, the relation (2) holds, thereby

$$M \not\models R. \tag{3}$$

Then, we start to find a rule R satisfying the condition (3). Consider the integrity constraint $\leftarrow \Gamma$ where Γ consists of ground LP-literals in M^+ which are relevant to the positive example A.[5] Since M does not satisfy this integrity constraint,

$$M \not\models \leftarrow \Gamma \tag{4}$$

holds. That is, $\leftarrow \Gamma$ is a rule which satisfies the condition (3).

Next, by $P \models_s \leftarrow A$, it holds that $A \notin M$, thereby $not\, A \in M^+$. Since $not\, A$ is relevant to A, the integrity constraint $\leftarrow \Gamma$ contains $not\, A$ in its body. Then, shifting the atom A to the head produces

$$A \leftarrow \Gamma' \tag{5}$$

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

Finally, the rule (5) is generalized by constructing a rule R^* such that $R^*\theta = A \leftarrow \Gamma'$ for some substitution θ. It is verified that the rule R^* satisfies the condition (2), i.e., $P \models_s not\, R^*$.

The next theorem presents a sufficient condition for the correctness of R^* to induce A.

Theorem 3.5. *(correctness of the IE rule [39]) Let P be a function-free and categorical NLP, A a ground atom, and R^* a rule obtained as above. If $P \cup \{R^*\}$ is consistent and $pred(A)$ does not appear in P, then $P \cup \{R^*\} \models_s A$.*

Example 3.6. Let P be the program

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

Given the example $L = flies(tweety)$, it holds that $P \models_s \leftarrow flies(tweety)$. Our goal is then to construct a rule R satisfying $P \cup \{R\} \models_s L$.

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

$$M^+ = \{\, bird(tweety),\ bird(polly),\ penguin(polly),$$
$$not\, penguin(tweety),\ not\, flies(tweety),\ not\, flies(polly)\,\}.$$

From M^+ picking up LP-literals which are relevant to L, the integrity constraint:

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

[5] Since P is function-free, Γ consists of finite LP-literals.

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 $P \cup \{R^*\} \models_s L$ holds.

The inverse entailment algorithm is also used for learning programs by negative examples [38].

3.4 Other Techniques

This section reviews other techniques for learning nonmonotonic logic programs.

Bain and Muggleton [2] introduce the algorithm called *Closed World Specialization* (CWS). In the algorithm, an initial program and an intended interpretation that a learned program should satisfy are given. In this setting, any atom which is not included in the interpretation is considered false. For instance, suppose the program:

$$\begin{aligned} P:\ & flies(x) \leftarrow bird(x), \\ & bird(eagle) \leftarrow,\ \ bird(emu) \leftarrow, \end{aligned}$$

and the intended interpretation:

$$M:\ \{\, flies(eagle),\, bird(eagle),\, bird(emu)\,\},$$

where $flies(emu)$ is not in M and is interpreted false. As P implies $flies(emu)$, the CWS algorithm specializes P and produces

$$\begin{aligned} & flies(x) \leftarrow bird(x),\ not\, ab(x), \\ & bird(eagle) \leftarrow,\ \ bird(emu) \leftarrow,\ \ ab(emu) \leftarrow . \end{aligned}$$

Here, $ab(x)$ is a newly introduced atom.[6] In this algorithm NAF is used for specializing Horn clauses and the CWS produces normal logic programs.

Inoue and Kudoh [19] propose an algorithm called *LELP* which learns *extended logic programs* (ELP) under the answer set semantics. The algorithm is close to Bain and Muggleton's method but is different from it on the point that [19] uses *Open World Specialization* (OWS) rather than the CWS under the 3-valued setting. The OWS does not use the closed world assumption to identify negative instances of the target concept.

Given positive and negative examples, LELP firstly constructs (monotonic) rules that cover positive examples by using an ordinary ILP algorithm,[7] then generates *default rules* to uncover negative examples by incorporating NAF literals

[6] Such an atom is called *invented*.

[7] An "Ordinary ILP" means any top-down/bottom-up ILP algorithm which is used in clausal logic.

to the bodies of rules. In addition, exceptions to rules are identified from negative examples and are then generalized to *default cancellation rules*. In LELP, hierarchical defaults can be learned by recursively calling the exception identification algorithm. Moreover, when some instances are possibly classified as both positive and negative, *nondeterministic rules* can also be learned so that there are multiple answer sets for the resulting program. Lamma *et al*. [22] formalize the same problem under the well-founded semantics. In their algorithms, different levels of generalization are strategically combined in order to learn solutions for positive and negative concepts.

Dimopoulos and Kakas [12] construct default rules with exceptions. For instance, suppose the background program:

$$\begin{aligned} P:\ & bird(x) \leftarrow penguin(x), \\ & penguin(x) \leftarrow super\_penguin(x), \\ & bird(a) \leftarrow,\ \ bird(b) \leftarrow, \\ & penguin(c) \leftarrow,\ \ super\_penguin(d) \leftarrow, \end{aligned}$$

and the positive and negative examples:

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

Their algorithm first computes a rule which covers all the positive examples:

$$r_1:\ flies(x) \leftarrow bird(x)\,.$$

This rule also covers the negative example, then the algorithm next computes a rule which explains the negative example:

$$r_2:\ \neg flies(x) \leftarrow penguin(x)\,.$$

In order to avoid drawing contradictory conclusions on c, the rule r_2 is given priority over r_1. Likewise, the algorithm next computes the rule

$$r_3:\ flies(x) \leftarrow super\_penguin(x)$$

and r_3 is given priority over r_2. A unique feature of their algorithm is that they learn rules using an ordinary ILP algorithm, and represent exceptions by a prioritized hierarchy without using NAF.

Sakama [39] presents a method of computing inductive hypotheses using answer sets of extended logic programs. Given an ELP P and a ground literal L, suppose a rule R satisfying $P \cup \{R\} \models_{AS} L$, where $\models_{AS}$ is an entailment under the answer set semantics. It is shown that this relation together with $P \not\models_{AS} L$ implies $P \not\models_{AS} R$. This provides a necessary condition for any possible hypothesis R which explains L. A candidate hypothesis is then obtained by computing answer sets of P, and constructing a rule which is unsatisfied in an answer set. The method provides the same result as [38] in a much simpler

Table 1. Comparison of Algorithms

Learned Programs	Algorithms	References
NLP	Ordinary ILP + specialization	[2]
	Selection from candidates	[4]
	Top-down	[16, 25, 40]
	Inverse resolution	[37, 43]
	Inverse entailment	[38]
	Least generalization	Section 3.1
ELP	Ordinary ILP	[12]
	Ordinary ILP + specialization	[19, 22]
	Computing Answer Sets	[39]

manner. In function-free stratified programs the algorithm constructs inductive hypotheses in polynomial-time.

Bergadano *et al*. [4] propose the system called $TRACY^{not}$ which learns NLPs using the derivation information of examples. In this system candidate hypotheses are given in input to the system, and from those candidates the system selects hypotheses which cover/uncover positive/negative examples. Martin and Vrain [25] introduce an algorithm to learn NLPs under the 3-valued semantics. Given a 3-valued model of a background program, it constructs (possibly recursive) rules to explain examples. Seitzer [40] proposes a system called *INDED*. It consists of a deductive engine which computes stable models or the well-founded model of a background NLP, and an inductive engine which induces hypotheses using the computed models and positive/negative examples. It can learn unstratified programs. Fogel and Zaverucha [16] propose an algorithm for learning strict and call-consistent NLPs, which effectively searches the hypotheses space using subsumption and iteratively constructed training examples.

Finally, the algorithms presented in this paper are summarized in Table 1.

For related research, *learning abductive logic programs* [13, 20, 21, 23] and *learning action theories* [24] are important applications of NMILP.

4 Summary and Open Issues

We presented an overview of techniques for realizing induction in nonmonotonic logic programs. Techniques in ILP have been centered on clausal logic so far, especially on Horn logic. However, as nonmonotonic logic programs are different from classical logic, existing techniques are not directly applicable to nonmonotonic situations. In contrast to clausal ILP, the field of nonmonotonic ILP is less explored and several issues remain open. Such issues include:

- *Generalization under implication*: In Section 3.1, we introduced the subsumption order between rules and provided an algorithm of computing a least generalization, which is an easy extension of the one in clausal logic. On the

other hand, in clausal theories there is another generalization based on the *implication order* which uses the entailment relation $C_1 \models C_2$ between two clauses C_1 and C_2. Concerning generalizations under implication in NMLP, however, the result of clausal logic is not directly applicable to NMLP. This is because the entailment relation in NMLP is considered under the commonsense semantics, which is different from the classical entailment relation. For instance, under the stable model semantics, the relation $\models_s$ is used instead of $\models$. Generality relations under implication would have properties different from the subsumption order, and the existence of least generalizations and their computability are to be examined.

- *Generalization operations in nonmonotonic logic programs*: In clausal theories, operations by inverting resolution generalize programs, but as presented in Section 3.2, they do not generalize programs in nonmonotonic situations in general. Then, it is important to develop program transformations which generalize nonmonotonic logic programs (under particular semantics) in general. Such transformations would serve as fundamental operations in nonmonotonic ILP. An example of this kind of transformations is seen in [43].

- *Relations between induction and other commonsense reasoning*: Induction is a kind of nonmonotonic inference, hence theoretical relations between induction and other nonmonotonic formalisms, including nonmonotonic logic programming, are of interest. Such relations will enable us to implement ILP in terms of NMLP, and also open possibilities to integrate induction and commonsense reasoning. Researches in this direction are found in [1, 14].

Ten years have passed since the first LPNMR conference was held in 1991. In [32] the preface says:

> ... there has been growing interest in the relationship between logic programming semantics and non-monotonic reasoning. It is now reasonably clear that there is ample scope for each of these areas to contribute to the other.

As a concluding remark, we rephrase the same sentence between NMLP and ILP. Combining NMLP and ILP in the framework of nonmonotonic inductive logic programming is an important step towards a better knowledge representation tool, and will bring fruitful advance in each field.

Acknowledgements

The author thanks Katsumi Inoue for comments on an earlier draft of this paper.

References

1. H. Ade and M. Denecker. AILP: abductive inductive logic programming. In: *Proc. 14th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 1201–1207, 1995.

2. M. Bain and S. Muggleton. Non-monotonic learning. In: S. Muggleton (ed.), *Inductive Logic Programming*, Academic Press, pp. 145–161, 1992.
3. C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming* 19/20:73–148, 1994.
4. 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.
5. G. Brewka and J. Dix. Knowledge representation with logic programs. In: *Proc. 3rd Workshop on Logic Programming and Knowledge Representation, Lecture Notes in Artificial Intelligence* 1471, Springer-Verlag, pp. 1–51, 1997.
6. F. Bry. Intensional updates: abduction via deduction. In: *Proc. 7th International Conference on Logic Programming*, MIT Press, pp. 561–575, 1990.
7. W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence* 36:149–176, 1988.
8. L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept learning. In: *Proc. 9th European Conference on Artificial Intelligence*, Pitmann, pp. 207–212, 1990.
9. L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence* 53:291–307, 1992.
10. L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In: *Proc. 13th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 1058-1063, 1993.
11. L. De Raedt and N. Lavrač. The many faces of inductive logic programming. In: *Proc. 7th International Symposium on Methodologies for Intelligent Systems, Lecture Notes in Artificial Intelligence* 689, Springer-Verlag, pp. 435–449, 1993.
12. Y. Dimopoulos and A. Kakas. Learning nonmonotonic logic programs: learning exceptions. In: *Proc. 8th European Conference on Machine Learning, Lecture Notes in Artificial Intelligence* 912, Springer-Verlag, pp. 122–137, 1995.
13. Y. Dimopoulos and A. Kakas. Abduction and inductive learning. In: L. De Raedt (ed.), *Advances in Inductive Logic Programming*, IOS Press/Ohmsha, pp. 144–171, 1996.
14. P. A. Flach and A. C. Kakas (eds). *Abduction and Induction: Essays on their Relation and Integration*, Applied logic series 18, Kluwer Academic, 2000.
15. P. A. Flach. Logical characterisations of inductive learning. In: D. M. Gabbay and R. Kruse (eds.), *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, vol. 4, Kluwer Academic Publishers, pp. 155–196, 2000.
16. L. Fogel and G. Zaverucha. Normal programs and multiple predicate learning. In: *Proc. 8th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1446, Springer-Verlag, pp. 175–184, 1998.
17. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In: *Proc. 5th International Conference and Symposium on Logic Programming*, MIT Press, pp. 1070–1080, 1988.
18. N. Helft. Induction as nonmonotonic inference. In: *Proc. 1st International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, pp. 149–156, 1989.
19. K. Inoue and Y. Kudoh. Learning extended logic programs. In: *Proc. 15th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 176–181, 1997.
20. K. Inoue and H. Haneda. Learning abductive and nonmonotonic logic programs. In: [14], pp. 213–231, 2000.

21. A. C. Kakas and F. Riguzzi. Learning with abduction. In: *Proc. 7th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1297, Springer-Verlag, pp. 181–188, 1997.
22. E. Lamma, F. Riguzzi, and L. M. Pereira. Strategies in combined learning via logic programs. *Machine Learning* 38(1/2), pp. 63–87, 2000.
23. E. Lamma, P. Mello, F. Riguzzi, F. Esposito, S. Ferilli, and G. Semeraro. Cooperation of abduction and induction in logic programming. In: [14], pp. 233–252, 2000.
24. D. Lorenzo and R. P. Otero. Learning to reason about actions. In: *Proc. 14th European Conference on Artificial Intelligence*, IOS Press, pp. 316–320, 2000.
25. 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.
26. R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence* 20:111-161, 1983.
27. T. M. Mitchell. *Machine Learning*, McGraw-Hill, 1997.
28. S. Muggleton (ed.). *Inductive Logic Programming*, Academic Press, 1992.
29. S. Muggleton and W. Buntine. Machine invention of first-order predicate by inverting resolution. In: [28], pp. 261–280, 1992.
30. S. Muggleton and L. De Raedt. Inductive logic programming: theory and methods. *Journal of Logic Programming* 19/20:629–679, 1994.
31. S. Muggleton. Inverse entailment and Progol. *New Generation Computing* 13:245–286, 1995.
32. A. Nerode, W. Marek, and V.S. Subrahmanian (eds.). *Proc. First International Workshop of Logic Programming and Nonmonotonic Reasoning*, MIT Press, 1991.
33. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of inductive logic programming. Lecture Notes in Artificial Intelligence* 1228, Springer-Verlag, 1997.
34. G. D. Plotkin. A note on inductive generalization. In: B. Meltzer and D. Michie (eds.), *Machine Intelligence* 5, Edinburgh University Press, pp. 153–63, 1970.
35. G. D. Plotkin. A further note on inductive generalization. In: B. Meltzer and D. Michie (eds.), *Machine Intelligence* 6, Edinburgh University Press, pp. 101–124, 1971.
36. T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, pp. 193–216, 1988.
37. C. Sakama. Some properties of inverse resolution in normal logic programs. In: *Proc. 9th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1634, Springer-Verlag, pp. 279–290, 1999.
38. C. Sakama. Inverse entailment in nonmonotonic logic programs. In: *Proc. 10th International Conference on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1866, Springer-Verlag, pp. 209–224, 2000.
39. C. Sakama. Learning by answer sets. In: *Proc. AAAI Spring Symposium on Answer Set Programming*, AAAI Press, pp. 181–187, 2001.
40. 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.
41. Y. Shoham. Nonmonotonic logics: meaning and utility. In: *Proc. 10th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 388–393, 1987.
42. L. Sterling and E. Shapiro. *The Art of Prolog*, 2nd Edition, MIT Press, 1994.
43. K. Taylor. Inverse resolution of normal clauses. In: *Proc. 3rd International Workshop on Inductive Logic Programming*, J. Stefan Institute, pp. 165–177, 1993.