

帰納推論による論理プログラムと規則性の学習

秋葉 澄孝 佐藤 泰介

電子技術総合研究所

本報告では現在我々が開発中の帰納推論システムについて述べる。このシステムでは、まず背景知識を用いて証明可能な正の単一節の集合の部分集合の最小汎化を計算し、次にそれらを組み合わせて探索すべき節の空間を生成する。その結果、我々のシステムは節の空間を効率よく探索でき様々な節を求めることが可能である。特に、プログラムとして用いられる節だけでなく、述語の規則性を表す節も求めることができることを示す。

Learning Logic Programs and Regularities from
Examples by Inductive Inference

Sumitaka Akiba Taisuke Sato

Electrotechnical Laboratory

1-1-4, Umezono, Tsukuba-shi, Ibaraki, 305, Japan

The purpose of this paper is to introduce our inductive inference system which we are developing now. Our system first calculates the least general generalizations of positive unit clauses provable from background knowledge and then combines them to form clauses. As a result our system can search the space of candidate clauses efficiently and is capable of finding clauses which not only can be used as a part of a logic program but also represent basic regularities of predicates.

1 はじめに

最近、帰納推論の研究の一つとして帰納論理プログラミング (ILP) が注目されている [5]。ILP の目標は、背景知識 K 、正例の集合 $\mathcal{E}^+$ 、負例の集合 $\mathcal{E}^-$ が与えられた時に、 $K \cup \mathcal{H} \vdash \mathcal{E}^+$, $K \cup \mathcal{H} \not\vdash \mathcal{E}^-$ となる仮説 $\mathcal{H}$ を求めることである。

例えば、既存の ILP システムは適当な正例と負例を与えられると

$$\left\{ \begin{array}{l} \text{concat}(X, [Y|Z], [Y|W]) \leftarrow \text{concat}(X, Z, W) \\ \text{concat}(X, [], [X]) \end{array} \right\}$$

といった論理プログラムを求めることができ [8]、さらにこのプログラムを背景知識として適当な正例と負例と共に与えられると

$$\left\{ \begin{array}{l} \text{reverse}([X|Y], Z) \leftarrow \text{reverse}(Y, W), \text{concat}(X, W, Z) \\ \text{reverse}([], []) \end{array} \right\}$$

というような背景知識を用いたプログラムを求めることもできる [8]。また、プログラムを完成させるために必要な述語を自動的に生成できるので、背景知識を与えられなくても concat に相当する述語 aux を自動生成して

$$\left\{ \begin{array}{l} \text{reverse}([X|Y], Z) \leftarrow \text{reverse}(Y, W), \text{aux}(X, W, Z) \\ \text{reverse}([], []) \\ \text{aux}(X, [Y|Z], [Y|W]) \leftarrow \text{aux}(X, Z, W) \\ \text{aux}(X, [], [X]) \end{array} \right\}$$

といったプログラムを求めることができる [3]。

ところで最初に述べた ILP の目標は、ILP だけでなく、帰納推論全般の目標である。

$$\begin{array}{l} \text{reverse}(X, Y) \leftarrow \text{reverse}(Y, X) \\ \text{append}(X, V, W) \leftarrow \text{append}(X, Y, U), \text{append}(U, Z, W), \text{append}(Y, Z, V) \end{array}$$

という節は reverse と append の基本的な規則性を表しており、これらの節を求めることも帰納推論の目標である。しかし、このような通常論理プログラムとしては使用されない節を ILP システムは求めることができない [2, 3, 4, 8]。

現在我々は節空間探索型の帰納推論システムを開発中である。このシステムでは、まず背景知識を用いて証明可能なアトム（正の単一節¹）を集めた集合の部分集合の最小汎化 (lgg) [6] をすべて計算し、次にそれらを組み合わせて節を生成する。その結果効率良く節空間を探索することが可能であり、論理プログラムとして使用できる節だけでなく、述語の規則性を表す節も求めることができる。

本稿では開発中のシステムの理論的背景、lgg の計算アルゴリズム、試作版のシステムの概要について述べる。

2 帰納推論システムの理論的背景

我々の帰納推論システムの理論的背景は帰納推論規則系 $\mathcal{R}$ である。 $\mathcal{R}$ は節に適用される 3 つの規則 • case rule: $D \xrightarrow{\text{case}} A \vee D$, $\neg A \vee D$ • inverse-or rule: $L \vee D \xrightarrow{\text{or}^{-1}} D$ • anti-substitution rule: $D\theta \xrightarrow{\theta^{-1}} D$ から成る。ただし、 D は節、 A はアトム、 L はリテラル、 θ は代入である。この規則系 $\mathcal{R}$ は、与えられた節集合 $\mathcal{E}$ に対して $\mathcal{H} \vdash \mathcal{E}$ となるすべての節集合 $\mathcal{H}$ を求めることができるという意味で、完全である。

実際に $\mathcal{R}$ を適用しようとなると、case rule を適用する際のアトムの選択、anti-substitution rule を適用する際の代入の決定が難しい。しかし、背景知識と正例が与えられている場合には、 $\mathcal{R}$ を適用する際の指針としてそれらを用いることができる。

以下では $K \cup \mathcal{E}^+$ が無矛盾で $\forall e^- \in \mathcal{E}^- \ K \cup \mathcal{E}^+ \not\vdash e^-$ である節の集合 K と正の単一節の集合 $\mathcal{E}^+$, $\mathcal{E}^-$ が与えられている時に、

$$\forall e^+ \in \mathcal{E}^+ \exists \theta \text{ s.t. } A\theta = e^+, K \vdash B_1\theta, \dots, K \vdash B_n\theta \quad (1)$$

$$\forall e^- \in \mathcal{E}^- \nexists \theta \text{ s.t. } A\theta = e^-, K \vdash B_1\theta, \dots, K \vdash B_n\theta \quad (2)$$

が成り立つ確定節 $A \leftarrow B_1, \dots, B_n$ を求める場合について考察する。 K を背景知識、 $\mathcal{E}^+$ の要素を正例、 $\mathcal{E}^-$ の要素を負例と呼ぶ。

¹ 本稿ではアトムと正の単一節を同一視する。

この場合には背景知識を用いて証明可能なアトム全体の集合の部分集合の lgg (以下では証明可能なアトムの lgg と略す) を計算して、それらを $\mathcal{R}$ を適用する際の指針として用いることができる。

$A^1, \dots, A^m$ を正例とし A を $\{A^1, \dots, A^m\}$ の lgg とする。まず、 $A\theta^j = A^j$ となる代入 θ^j から出発し、証明可能なアトムの lgg $B_1, \dots, B_k, \dots$ を順次生成して $\mathcal{K} \vdash B_k \theta^j$ となるように θ^j を拡張できるか否かを調べる。しかも、 $A \leftarrow B_1, \dots, B_k$ が $\{A\theta^j \leftarrow B_1 \theta^j, \dots, B_k \theta^j \mid j = 1, \dots, m\}$ の lgg になることが望ましいので、 B_k に現れる異なる変数 X, Y に対して $X\theta^j \neq Y\theta^j$ となる代入 θ^j が存在するように拡張できるか否かを調べる。

もし、このように代入を拡張できるならば case rule を用いて $A\theta^j \leftarrow B_1 \theta^j, \dots, B_{k-1} \theta^j$ の本体に $B_k \theta^j$ を追加し、その後 anti-substitution rule を適用して $A \leftarrow B_1, \dots, B_k$ を導く。この節は (1) を満たすので、後はこうして導かれた節の中から (2) を満たすものを選び出せば良い。

これで $\mathcal{R}$ を用いて (1) と (2) を満たす節を導くために必要な、 $A^j (= A\theta^j)$ が頭部である確定節の本体に case rule で追加すべきアトム $B_k^j (= B_k \theta^j)$ と anti-substitution rule を適用すべき代入 θ^j が見つかったことになる。

簡単に言うと、この過程では証明可能なアトムの lgg の列 $B_1, \dots, B_k, \dots$ を順次生成して、それらを組み合わせて作られる節 $A \leftarrow B_1, \dots, B_k$ が (1) と (2) を満たすか否かを調べている。(1) を満たすか否かを代入を拡張しながら調べているのである。

この過程は正例が一つしかなくても目標とする節を導くことができる。条件 (1) を $\exists e^+ \in \mathcal{E}^+ \exists \theta \text{ s.t. } A\theta = e^+, \mathcal{K} \vdash B_1 \theta, \dots, \mathcal{K} \vdash B_n \theta$ に変えることができると言っても良い。

また、背景知識で与えられていない新しい述語記号を用いたアトムを容易に導入することができる。その場合には次の 2 つの条件

$$\exists e^+ \in \mathcal{E}^+ \exists \theta \text{ s.t. } A\theta = e^+, \mathcal{K} \vdash B_1 \theta, \dots, \mathcal{K} \vdash B_n \theta, F\theta \in \mathcal{H}' \quad (1')$$

$$\forall e^- \in \mathcal{E}^- \nexists \theta \text{ s.t. } A\theta = e^-, \mathcal{K} \vdash B_1 \theta, \dots, \mathcal{K} \vdash B_n \theta, F\theta \in \mathcal{H}' \quad (2')$$

を満たす確定節 $A \leftarrow B_1, \dots, B_n, F$ と F の具体例の集合 $\mathcal{H}'$ を求めることを目標とする。ただし、 F は背景知識、正例、負例に現れない述語記号を持つアトムである。

$A, B_1, \dots, B_n$ に現れる変数を $X_1, \dots, X_p$ とし、特に A に現れる変数を $X_1, \dots, X_q$ ($q \leq p$) とする。 aux を新しい述語記号、 θ を $A\theta \in \mathcal{E}^+, \mathcal{K} \vdash B_i \theta$ となる代入とする。 $F = aux(X_1, \dots, X_p), \mathcal{H}' = \{F\theta\}$ とすれば、(1') が成り立つ。また、 $\mathcal{E}^+ \cap \mathcal{E}^- \neq \emptyset$ だから (2') も成り立つ。つまり $A \leftarrow B_1, \dots, B_n, F$ が求めたい節である。

ただし、一般には aux の引数の集合 $\{X_1, \dots, X_p\}$ は不必要な変数を含んでおり、また引数の集合が $X_1, \dots, X_q$ をすべて含めば (1') (2') が成り立つのは自明である。従って $\{X_1, \dots, X_p\}$ の部分集合の中で、(1') (2') を満足するアトムを作ることができ、 $X_1, \dots, X_q$ のいずれかは含まないものの極小な集合を aux の引数の集合として用いる必要がある。それを求めるためには例えば DBC[3] のようなアルゴリズムを利用できる。

我々は上で述べた方法を用いて (1') (2') を満たす節を求める帰納推論システムを現在開発中である。その主要な部分は、背景知識を用いて証明可能なアトム全体の集合の部分集合の lgg とその variant を生成する部分である。次の節ではその方法について述べる。

3 lgg の生成方法

我々の帰納推論システムでは、背景知識を用いて証明可能なアトム全体の集合の部分集合の lgg を計算し、それらを組み合わせて節を生成する。

一般には証明可能なアトムの lgg は無数にあり、それらをすべて求めることは不可能である。ところが、システムが求めるべき節において使用可能な有限個の述語記号、関数記号、定数と関数の入れ子の深さの上限がわかっているならば、lgg は variant を同一視すると有限個である。従って何らかの方法ですべてを計算可能であると期待できる。

我々のシステムでは証明可能なアトムの lgg を次のように二段階に分けて計算する。

まず証明可能なアトムの lgg を variant を同一視して計算する。ただし、計算すべき lgg を有限個にするための制限として、使用可能な有限個の述語記号、関数記号、定数と関数の入れ子の深さの上限を与える。

次に lgg の variant で節の本体に追加すべきものを計算する。これらは特別な条件を与えなくても有限個しかなく、すべて計算することが可能である。

variant を同一視して lgg を計算する方法として、背景知識に含まれているアトム (正の単一節) を用いて計算する方法がある。証明可能なアトムは一般には無数にあるので、たとえ上で述べた制限の範囲内の lgg に限定しても、この方法で lgg をすべて求められる保証がない。しかし、背景知識にアトムが十分含まれていれば、証明可能なアトムの lgg のすべてをこの方法で計算可能であると期待できる。そのために重要なことは様々な形のアトムが背景知識に含まれていることであって、アトムが数多く含まれている必要はない。

アトムが十分に含まれていない場合には、背景知識を用いて証明可能なアトムを一旦生成して、それらを用いて lgg を計算できる。ただし、アトムが特別な形のものに偏らないように生成しなければならない。

証明可能なアトムの lgg をすべて確実に求めるためには、アトムを一旦生成することなく直接背景知識から lgg を計算する必要がある。しかし今のところ lgg の一部を計算する方法しか見つかっていない。

3.1 lgg を計算するアルゴリズム

ここではアトムの集合が与えられた時に、その部分集合の lgg を variant を同一視してすべて求めるアルゴリズムについて述べる。

単純な方法としては与えられたアトムの集合の部分集合をすべて求めて、それらの lgg を計算すれば良い。この方法はアトムの数が多くなると非常に効率が悪いため、以下では効率良く lgg を求めるアルゴリズムについて述べる。

まずアトム全体を述語記号 P_i で分類して、アトムの集合 $E(P_i) = \{e \mid e \text{ は述語記号が } P_i \text{ であるアトム}\}$ を作る。その後、次のアルゴリズムを用いて $lggs(E(P_i))$ を計算する。 $\bigcup_{P_i} lggs(E(P_i))$ がすべての lgg の集合である。

述語記号が同じアトムの集合 E が与えられた時に $lggs(E)$ を計算するアルゴリズム

1. $t = lgg(E)$ を計算する。 $V = \{v_i\}$ を t の変数の集合とする。
2. E の各要素 e_j に対して $t\theta_j = e_j$ となる代入 $\theta_j = \{t_i^j/v_i\}$ を計算する。
3. t の各変数 v_i に対して、 E の要素 e_j を項 t_i^j の関数記号 f_k 及び定数 c_k で分類して $E_{v_i, f_k} = \{e_j \in E \mid t_i^j = f_k(\dots)\}$ と $E_{v_i, c_k} = \{e_j \in E \mid t_i^j = c_k\}$ を作る。
4. t の変数の集合 V の部分集合で要素の数が 2 以上である各集合 $V' = \{v_{i_1}, \dots, v_{i_k}\}$ に対して、 E の要素 e_j の中から $t_{i_1}^j, \dots, t_{i_k}^j$ が同じものを集めて $E_{V'} = \{e_j \in E \mid t_{i_1}^j = \dots = t_{i_k}^j, v_{i_1}, \dots, v_{i_k} \in V'\}$ を作る。
5. $lggs(E) = \{t\} \cup \bigcup_{v_i, f_k} lggs(E_{v_i, f_k}) \cup \bigcup_{v_i, c_k} lggs(E_{v_i, c_k}) \cup \bigcup_{V'} lggs(E_{V'})$ を出力する。

例 $E = \{reverse([a], [a]), reverse([b], [b]), reverse([a, b], [b, a]), reverse([a, c], [c, a])\}$ の $lggs$ を計算する。

$$\begin{aligned}
lgg(E) &= reverse([X|Y], [Z|W]) \\
E_{X,a} &= \{reverse([a], [a]), reverse([a, b], [b, a]), reverse([a, c], [c, a])\} \quad E_{X,b} = \{reverse([b], [b])\} \\
E_{Y,[]} &= E_{W,[]} = \{reverse([a], [a]), reverse([b], [b])\} \\
E_{Y,cons} &= E_{W,cons} = \{reverse([a, b], [b, a]), reverse([a, c], [c, a])\} \\
E_{Z,a} &= \{reverse([a], [a])\} \quad E_{Z,b} = \{reverse([b], [b]), reverse([a, b], [b, a])\} \quad E_{Z,c} = \{reverse([a, c], [c, a])\} \\
lggs(E_{X,a}) &= \left\{ \begin{array}{l} reverse([a|X], [Y|Z]), reverse([a], [a]), reverse([a, X], [X, a]) \\ reverse([a, b], [b, a]), reverse([a, c], [c, a]) \end{array} \right\} \\
lggs(E_{X,b}) &= \{reverse([b], [b])\} \\
lggs(E_{Y,[]}) &= \{reverse([X], [X]), reverse([a], [a]), reverse([b], [b])\} \\
lggs(E_{Y,cons}) &= \{reverse([a, X], [X, a]), reverse([a, b], [b, a]), reverse([a, c], [c, a])\} \\
lggs(E_{Z,a}) &= \{reverse([a], [a])\} \\
lggs(E_{Z,b}) &= \{reverse([X|Y], [b|Z]), reverse([b], [b]), reverse([a, b], [b, a])\} \\
lggs(E_{Z,c}) &= \{reverse([a, c], [c, a])\} \\
E_{\{X,Y,Z,W\}} &= E_{\{X,Y,Z\}} = E_{\{X,Y,W\}} = E_{\{X,Z,W\}} = E_{\{Y,Z,W\}} = E_{\{X,Y\}} = E_{\{X,W\}} = E_{\{Y,Z\}} = E_{\{Z,W\}} = \phi \\
E_{\{X,Z\}} &= E_{\{Y,W\}} = \{reverse([a], [a]), reverse([b], [b])\} \\
lggs(E_{\{X,Z\}}) &= \{reverse([X], [X]), reverse([a], [a]), reverse([b], [b])\} \\
lggs(E) &= \left\{ \begin{array}{l} reverse([X|Y], [Z|W]), reverse([a|X], [Y|Z]), reverse([a], [a]) \\ reverse([a, X], [X, a]), reverse([a, b], [b, a]), reverse([a, c], [c, a]) \\ reverse([b], [b]), reverse([X], [X]), reverse([X|Y], [b|Z]) \end{array} \right\}
\end{aligned}$$

この例の中で $reverse([X], [X])$ を二度計算しているように、上のアルゴリズムでは同じ lgg を重複して計算してしまう。我々はこのアルゴリズムを改良して各 lgg を一度だけ計算するようにしたが、本稿ではこれについての説明を省略する。

lgg において使用可能な関数記号、定数記号、関数の入れ子の深さの条件が与えられている場合には、条件を満たすものだけ計算すれば良い。上の例において使用可能な定数が $[]$ (空リスト) の場合には a は使用できない定数なので $E_{X,a}$ などは計算する必要がなく、 $lggs(E) = \{reverse([X|Y], [Z|W]), reverse([X], [X])\}$ となる。

3.2 アトムの変数の生成

ここでは節とアトムが与えられた時に、節の本体に追加すべきアトムの変数の生成する方法について述べる。

節の本体に追加するために生成するので、アトムの変数のうちの節と共通な変数を持つものだけを生成すれば良い [1, 2, 4]。この根拠は次節で述べる。また、ある variant の変数のうちの節に現れないものを節に現れない別の変数に rename する必要はない。

従って m 個の変数が現れるアトム A の variant のうち n 個の変数 $X_1, \dots, X_n$ が現れる節に追加すべきものは次のようにして生成できる。まず A の変数を節に現れない変数 $Y_1, \dots, Y_m$ に rename しておく。後は $X_1, \dots, X_n$ と $Y_1, \dots, Y_m$ から k 個ずつ選んで変数の置換 $\theta = \{X_{i_1}/Y_{j_1}, \dots, X_{i_k}/Y_{j_k}\}$ を作り、これを用いて A の variant $A\theta$ を生成してゆけばよい。

重複がないように variant を数えあげると、その個数は $\sum_{k=1}^{\max\{n,m\}} n P_k \cdot m P_k / k!$ である。

例えば $\text{reverse}([X|Y], [Z|W])$ に $\text{reverse}(U, V)$ を追加する時には次の 20 個の variant

$\text{reverse}(X, V)$	$\text{reverse}(U, X)$	$\text{reverse}(X, Y)$	$\text{reverse}(X, Z)$	$\text{reverse}(X, W)$
$\text{reverse}(Y, V)$	$\text{reverse}(U, Y)$	$\text{reverse}(Y, X)$	$\text{reverse}(Y, Z)$	$\text{reverse}(Y, W)$
$\text{reverse}(Z, V)$	$\text{reverse}(U, Z)$	$\text{reverse}(Z, X)$	$\text{reverse}(Z, Y)$	$\text{reverse}(Z, W)$
$\text{reverse}(W, V)$	$\text{reverse}(U, W)$	$\text{reverse}(W, X)$	$\text{reverse}(W, Y)$	$\text{reverse}(W, Z)$

を生成すれば良い。

我々のシステムではこれらの variant は条件 (1') (2') を満たす節を作るために用いる。節の本体に variant を追加する毎にこれらの条件を満たすか否か調べ、無駄のないように variant の生成順序を考慮すると、上で述べた variant をすべて生成する必要はない。これについての説明は省略する。

4 帰納推論システムの概要

現在我々は証明可能なアトムの lgg を用いて探索すべき節空間を生成する帰納推論システムを開発中である。

我々のシステムの目標は背景知識 $\mathcal{K}$ 、正例の集合 $\mathcal{E}^+$ 、負例の集合 $\mathcal{E}^-$ が与えられた時に、次の条件

- (1) $A, B_1, \dots, B_n$ は証明可能なアトムの lgg
- (2) F は $\mathcal{K}, \mathcal{E}^+, \mathcal{E}^-$ に現れない述語記号をもつアトム
- (3) $\exists e^+ \in \mathcal{E}^+ \exists \theta \text{ s.t. } A\theta = e^+, \mathcal{K} \vdash B_1\theta, \dots, \mathcal{K} \vdash B_n\theta, F\theta \in \mathcal{H}'$
- (4) $\forall e^- \in \mathcal{E}^- \nexists \theta \text{ s.t. } A\theta = e^-, \mathcal{K} \vdash B_1\theta, \dots, \mathcal{K} \vdash B_n\theta, F\theta \in \mathcal{H}'$
- (5) C に現れる任意の 2 つの異なる変数 X, Y に対して $A\theta \in \mathcal{E}^+, \mathcal{K} \vdash B_1\theta, \dots, \mathcal{K} \vdash B_n\theta, F\theta \in \mathcal{H}'$, $X\theta \neq Y\theta$ となる代入 θ がある。
- (6) $C' \subseteq C$, C' は節 C に関する条件 (4) を満たす $\Rightarrow C' = C$

を満たす確定節 $C (= A \leftarrow B_1, \dots, B_n, F)$ と F の具体例の集合 $\mathcal{H}'$ をすべて求めることである。

これらの条件のうち (6) は節の既約性に関する条件である [4]。

2 つの節 C_1, C_2 が両方とも上の条件 (1) から (5) を満たし $C_1 \subseteq C_2$ が成り立つ場合には、帰納推論システムは C_1 の方だけ求めれば十分である。そのためには節の条件の中に (6') $C' \subseteq C$, C' は C に関する条件 (1) から (5) を満たす $\Rightarrow C' = C$ を加えれば良い。一方「 C が (1) (2) (3) (5) を満たし $C' \subseteq C$ ならば C' も (1) (2) (3) (5) を満たす」から、「 C は (1) から (5) と (6') を満たす $\Leftrightarrow C$ は (1) から (6) を満たす」が成り立つ。

以上の理由により、節の既約性の条件として (6) を加えてある。

ところで節 C が (1) から (6) を満たす場合には、 C のアトムを 2 つに分けて作られる節 C_1, C_2 ($\neq C$, $C_1 \cup C_2 = C$) には共通な変数がある。もしも共通な変数がないならば C_1, C_2 のうち C の頭部 A を含む方の節 $C_\alpha (= A \leftarrow B_{i_1}, \dots, B_{i_k})$ は C の条件 (4) を満たし、 C が (6) を満たすことに矛盾するからである。

このことから、 C が (1) から (6) を満たす場合には C の本体のアトムを並べかえて $\text{var}(A, B_1, \dots, B_{i-1})^2 \cap \text{var}(B_i) \neq \emptyset$ ($i = 1, \dots, n$) となるようにできることがわかる。従って、前節で述べたように節本体に追加するアトムは節と共通な変数を持つものに限ってよいことになる。

我々の帰納推論システムの開発は、試作版のシステムを CMU Lisp を用いてプログラムした段階である。この試作版には次の 5 つの制約がある。

² $\text{var}(X)$ は X に現れる変数全部からなる集合を表す。

- 正例と負例は同じ述語記号を用いた正の基底単一節である。
- 背景知識として正例のみを使用する。したがって $X \vdash A \Leftrightarrow A \in \mathcal{E}^+$ である。
- 節に現れる述語記号は、正例、負例と同じ記号及びシステムが新しく作った記号のみである。
- システムが新しく作った述語の具体例 $\mathcal{H}'$ は出力しない（内部では計算している）。
- 関数記号は節の頭部にしか現れない。ただし例外がある。

試作版のシステムは、入力として • 正例の集合 $\mathcal{E}^+$ • 負例の集合 $\mathcal{E}^-$ • 目標とする節に使用可能な定数の集合 $Const$ • 目標とする節の長さの上限 CL_{max} を与えられると、条件 (1) から (6) を満たす節のうち上の制限の範囲内のものをすべて生成・出力する。この過程の間はユーザーは一切介入しない。

試作版のシステムは次のアルゴリズムを用いている。

1. 背景知識を用いて証明可能なアトム全体の部分集合の lgg を計算する。それらのうち正例と負例の一般化になっているものを集めて集合 S をつくる。正例の一般化だが負例の一般化でないものをすべて出力する。正例の一般化でないものを集めて集合 LGG をつくる。
2. S から節 D を選ぶ。節がなければ終了する。
3. LGG からアトムを選び D に追加すべき variant B を作る。variant があれば 4. へ。variant がなければ新しい述語記号を用いたアトム F を D の本体に追加して F の具体例の集合 $\mathcal{H}'$ を計算する。 $C = D \vee \neg F$ が条件 (3) (4) (5) (6) を満足するように F を作ることができたら C を出力する。 D を S から取り除き 2. へ。
4. D の本体に B を追加した節 $C = D \vee \neg B$ が条件 (3) (4) (5) (6) を満足する場合、 C を出力する。
5. C の本体にアトムを追加する可否か決定する。追加する場合は C を S に加える。
6. 2. に戻る。

lgg と variant の計算には前節で述べたアルゴリズムを用いている。また、節 C にアトムを追加しない条件として • C が (3) または (5) を満たさない • C が (3) (4) (5) を満たすを用いている。

新しい述語記号を用いたアトム F を本体に追加して F の具体例の集合 $\mathcal{H}'$ を計算する時には、DBC[3] と同様なアルゴリズムを用いている。ただし、DBC は新しいアトムを作るために必要な変数の極小集合を一つしか求めないが、我々のシステムではすべての節を求めることを目標にしているので、変数の極小集合をすべて求めてアトムを作る。

試作版のシステムには付いていないが、出力しようとする節を少し一般化する機構が必要である。例えば、我々のシステムの完成版は適切な入力を与えられると $reverse([X|Y], [Z|W]) \leftarrow reverse(Y, U), concat(X, U, [Z|W])$ を作ることができるはずである。この節では変数 Z, W が項 $[Z|W]$ の一部としてしか現れないので、システムはこの節を出力する代わりに $[Z|W]$ を一般化した節 $reverse([X|Y], V) \leftarrow reverse(Y, U), concat(X, U, V)$ を出力した方がよい。このように節内部のある項に現れるすべての変数が同じ形の項にしか現れない場合、支障がなければその項を一般化するべきである。

また、例えばプログラムとして必要な最小限の節を選別するような、生成した節の中から目的に応じて必要な節を選別する機構が必要だが、試作版には付いていない。

実行例 1 入力

$$\mathcal{E}^+ = \left\{ \begin{array}{l} reverse([], []), reverse([a], [a]), reverse([b], [b]), reverse([c], [c]) \\ reverse([b, c], [c, b]), reverse([c, b], [b, c]), reverse([b, a], [a, b]), reverse([a, b], [b, a]) \\ reverse([c, a], [a, c]), reverse([a, c], [c, a]), reverse([a, b, c], [c, b, a]), reverse([a, c, b], [b, c, a]) \\ reverse([c, b, a], [a, b, c]), reverse([b, c, a], [a, c, b]) \end{array} \right\}$$

$$\mathcal{E}^- = \left\{ \begin{array}{l} reverse([], [a]), reverse([a], []), reverse([c], [b]) \\ reverse([b, c], [b, c]), reverse([b, c], [a, b]), reverse([a, b], [a]), reverse([a, b], [b]) \\ reverse([b, c], [c, b, a]), reverse([c, b, a], [b, c, a]), reverse(c, b, a), [b, c]), reverse([a, b], [b, c]) \end{array} \right\}$$

$$Const = \{\}\quad CL_{max} = \infty$$

とした場合の出力は

$$reverse([X, Y, Z], [Z, Y, X]) \quad reverse([X, Y], [Y, X]) \quad reverse([X], [X])$$

$$\begin{aligned}
&reverse(X, Y) \leftarrow reverse(Y, X) \\
&reverse([X|Y], [Z|W]) \leftarrow reverse(Y, U), aux1(X, Z, W, U) \\
&reverse([X|Y], [Z|W]) \leftarrow reverse(W, U), aux2(X, Y, Z, U) \\
&reverse([X|Y], [Z|W]) \leftarrow reverse(U, Y), aux3(X, Z, W, U) \\
&reverse([X|Y], [Z|W]) \leftarrow reverse(U, W), aux4(X, Y, Z, U)
\end{aligned}$$

である。初めの4つの節は通常プログラムとしては使用されないが、*reverse*の基本的な性質を表している節である。

後の4つの節に含まれている *aux1*(*X, Z, W, U*)、*aux2*(*X, Y, Z, U*)、*aux3*(*X, Z, W, U*)、*aux4*(*X, Y, Z, U*) はシステムが作った新しい述語である。システムの内部ではこれらの新しい述語の正例と負例を *reverse* に関する正例と負例から自動的に計算しているので、それらを入力として再びシステムを起動すると新しい述語に関する節を求めることができる。

例えば、システムが計算した *aux1* の正例と負例は

$$\begin{aligned}
\mathcal{E}^+ &= \left\{ \begin{array}{l} aux1(a, a, [], []), aux1(b, b, [], []), aux1(c, c, [], []) \\ aux1(b, c, [b], [c]), aux1(c, b, [c], [b]), aux1(b, a, [b], [a]), aux1(a, b, [a], [b]) \\ aux1(c, a, [c], [a]), aux1(a, c, [a], [c]), aux1(a, c, [b, a], [c, b]), aux1(a, b, [c, a], [b, c]) \\ aux1(c, a, [b, c], [a, b]), aux1(b, a, [c, b], [a, c]) \end{array} \right\} \\
\mathcal{E}^- &= \left\{ \begin{array}{l} aux1(c, b, [], []), aux1(b, b, [c], [c]), aux1(b, a, [b], [c]), aux1(a, a, [], [b]), aux1(a, b, [], [b]) \\ aux1(b, c, [b, a], [c]), aux1(c, b, [c, a], [a, b]), aux1(c, b, [c], [a, b]), aux1(a, b, [c], [b]) \end{array} \right\}
\end{aligned}$$

である。正例、負例としてこれらを用い、*Const* = {[]}、*CLmax* = 2 としてシステムを起動すると、*aux1* に関する節

$$\begin{aligned}
&aux1(X, Y, [X], [Y]) \quad aux1(X, X, [], []) \\
&aux1(X, Y, [Z|W], [Y|U]) \leftarrow aux1(X, Z, W, U)
\end{aligned}$$

などが出力される。このようにして求められる *reverse* と *aux1* に関する節と *reverse* の正例の中から

$$\begin{aligned}
&reverse([X|Y], [Z|W]) \leftarrow reverse(Y, U), aux1(X, Z, W, U) \\
&reverse([], []) \\
&aux1(X, Y, [Z|W], [Y|U]) \leftarrow aux1(X, Z, W, U) \\
&aux1(X, X, [], [])
\end{aligned}$$

を選別すると *reverse*(*X, Y*) の具体例を生成するプログラムになる。

実行例 2 入力

$$\begin{aligned}
\mathcal{E}^+ &= \left\{ \begin{array}{l} append([], [], []), append([], [a], [a]), append([], [b], [b]), append([], [b, a], [b, a]) \\ append([a], [], [a]), append([b], [], [b]), append([b], [a], [b, a]), append([c], [b], [c, b]) \\ append([c, b], [a], [c, b, a]), append([c], [b, c], [c, b, a]) \\ append([c], [b, a], [c, b, a]), append([c, b], [], [c, b]) \end{array} \right\} \\
\mathcal{E}^- &= \left\{ \begin{array}{l} append([b], [a], [a, b]), append([a], [b], [a]), append([a], [a], [a, b]), append([c], [a], [c, b, a]) \\ append([c, b], [a], [c, a]), append([c, b], [a], [b, a]), append([c, b], [a], [b, c, a]) \\ append([c], [b, a], [c, a]), append([c], [b, a], [c, a]), append([c], [b, a], [b, a]) \end{array} \right\} \\
Const &= \{[]\} \\
CLmax &= 4
\end{aligned}$$

の場合、約 5000 個の節を出力する。その中には

$$\begin{aligned}
&append(X, [], X) \quad append([X], Y, [X|Y]) \quad append([], X, X) \\
&append(X, Y, Z) \leftarrow append(X, W, U), append(W, V, Y), append(U, V, Z) \\
&append([X|Y], Z, [X|W]) \leftarrow append(Y, Z, W) \\
&append(X, Y, [Z]) \leftarrow append(Y, X, [Z])
\end{aligned}$$

など興味ある節が含まれている。

5 既存のシステムとの比較

我々が開発中の帰納推論システムは理論上は効率の良いシステムである。従って、システムが求める節をプログラムとして用いることができる節など特別な形のものに限定せずに、様々な節を求めることができる。

これに対して既存の ILP システムでは効率を良くするために様々なヒューリスティクスを用いている。特にシステムが求められる節をプログラムとして用いることができるものに限定しているのが現状である。そのために $reverse(X, Y) \leftarrow reverse(Y, X)$ や $append(X, V, W) \leftarrow append(X, Y, U), append(U, Z, W), append(Y, Z, V)$ など、述語の基本的な規則性を表す興味ある節を求めることができない。

以下では我々のシステムと代表的な ILP システムである MIS[8] 及び GOLEM[4] を比較する。

5.1 MIS との比較

我々が開発中の帰納推論システムと同様に、MIS も節空間探索型の帰納推論システムである。

MIS は 3 つの refinement オペレータ • 変数を具体化する • 変数どうしを単一化する • 節本体にアトムを一つ付ける を用いて生成される節の空間を探索する。

節の本体に使用できる述語として $reverse(X, Y)$ と $concat(X, Y, Z)$ を与えると、MIS は refinement オペレータを用いて次のように節を生成する。

$$reverse(X, Y) \mapsto \left\{ \begin{array}{l} reverse(X, X) \mapsto \dots \\ \\ reverse([X|Y], Z) \mapsto \left\{ \begin{array}{l} reverse([X|Y], Y) \mapsto \dots \\ reverse([X, Y|Z], W) \mapsto \dots \\ reverse([X|Y], [Z|W]) \mapsto \dots \\ reverse([X|Y], Z) \leftarrow reverse(Y, W) \mapsto \dots \\ reverse([X|Y], Z) \leftarrow concat(X, Y, W) \mapsto \dots \\ \vdots \end{array} \right. \\ \\ reverse(X, [Y|Z]) \mapsto \left\{ \begin{array}{l} reverse(X, [Y|X]) \mapsto \dots \\ reverse([X|Y], [Z|W]) \mapsto \dots \\ reverse(X, [Y, Z|W]) \mapsto \dots \\ \vdots \end{array} \right. \\ \\ \vdots \end{array} \right.$$

この方法には次の問題点がある。

- 具体化が不十分なアトムを生成する。 $reverse(X, Y)$ の具体例は 2 つの引数のうち片方が空でないリストの場合にはもう片方の引数も空でないリストである。しかし、上の例のように MIS は $reverse(X, Y)$ の変数を片方ずつ具体化して、具体化が不十分なアトム $reverse([X|Y], Z)$ 、 $reverse(X, [Y|Z])$ を生成する。これでは効率が悪いので両方の変数を同時に具体化して $reverse([X|Y], [Z|W])$ を生成するべきである。
- 具体例が存在しないアトムを生成する。 $reverse(X, Y)$ の具体例では 2 つの引数は同じ長さのリストであり、 $reverse([X|Y], Y)$ 、 $reverse(X, [Y|X])$ の具体例は存在しない。しかし上の例のように MIS はこれらのアトムを生成する。MIS は生成した節の具体例が存在するか否か確かめ、存在しない場合にはその節にオペレータを適用せず探索空間の生成には用いない。しかし、この例と同じアトムを含む別の節を繰り返し生成するので効率が悪い。
- 同じ節を繰り返し生成する。上の例では $reverse([X|Y], [Z|W])$ が 2 回生成されている。これはオペレータの適用順序や組み合わせ方を十分考慮すれば避けることができるはずである。しかし実際にはそのような考慮は困難であり、MIS やそれを改良したシステム [2, 3] でもそのような考慮は払われていない。

我々のシステムでは、まず証明可能なアトムの lgg を計算して、それらを組み合わせて節を生成する。そのおかげで MIS がかかえている問題点を次のように克服している。

- 十分具体化されたアトムだけを扱う。 lgg はその名の通り最小限に一般化されたアトム、つまり十分具体化されたアトムである。例えば $reverse(X, Y)$ の具体例の lgg の中には $reverse([X|Y], [Z|W])$ は含まれているが $reverse([X|Y], Z)$ や $reverse(X, [Y|Z])$ は含まれていない。この様に我々のシステムが生成する節のアトムは十分具体化されたものに限られる。

- 具体例が存在するアトムだけを扱う。lgg には必ず具体例があるので、あらかじめ計算した lgg の中に $reverse([X|Y], Y)$, $reverse(X, [Y|X])$ のような具体例が存在しないアトムが含まれることはない。従って我々のシステムが生成する節のアトムは具体例が存在するものに限られる。
- 重複なく節を生成するのが易しい。節の本体に追加するためにアトムの variant を作ることをアトム間の変数どうしの単一化とみなせば、我々のシステムは ● アトム間の変数どうしの単一化 ● 節の本体にアトムを付ける の 2 種類のオペレータを用いて節を生成しているとみなすことができる。つまり ● 変数の具体化 ● アトム内の変数どうしの単一化 のオペレータがないので、生成する節の空間の構造が単純になり、重複なく節を生成するようにオペレータを適用していくことは MIS の場合よりも容易である。実際前節で述べたように、まず variant を同一視して lgg を計算し、次に必要な variant を作れば重複なくアトムを生成できるので、それらを組み合わせて重複なく節を生成できる。

5.2 GOLEM との比較

我々が開発中の帰納推論システムと同様に、GOLEM も lgg を利用した帰納推論システムである。

GOLEM は、まずあらかじめ決められた計算量の限度内で証明可能な基底アトムを計算し、その結果得られた基底アトム全部を本体に持ち正例を頭部に持つ確定節を作る。次にそれらの節の lgg[6] を計算し、lgg の本体から余分なアトムを取り除いて、目標の条件を満たす節を求める。

目標の条件を満たす節をできるだけ確実に求めるためには、初めにできるだけ多く基底アトムを計算する必要がある。文献 [4] で述べられている GOLEM の実行例では、最初に計算した基底アトムの数は結果として求められた節の本体の数の 20 倍から 90 倍である。

基底アトムの数が多いと本体が長い確定節の lgg を計算することになる。本体のアトムの個数が m と n の確定節の lgg を計算すると、一旦 $m \times n$ 個のアトムを本体に持つ確定節ができる。lgg を何度も計算する必要があるので、基底アトムの個数が多いとそれだけ計算量が増える。

余分なアトムを取り除く作業の計算量は節本体の長さの影響を強く受ける。この作業は本体全体の集合の部分集合の中から与えられた条件を満たす極小部分集合を探索する作業なので、節本体が長いと極端に計算量が多くなる。

このように GOLEM で用いている方法は、直観的には効率が悪いと思われる。一旦非常に長い節を生成してそれを短くするだけでなく、その計算量は少ないとは言い難いからである。

ところで、本来この方法では無数にある証明可能な基底アトムをすべて計算して、それらを本体に持つ無限の長さの確定節の lgg を計算しなければならない。この作業は不可能なので、GOLEM では基底アトムの計算を途中で打ち切り、そこまでで計算できた有限個の基底アトムを用いて lgg を計算する。

本来必要な計算を途中で打ち切るので、システムの能力が制限されていると考えられる。しかし実際にどの程度制限されていて、どのような節ならば確実に求められるのか理解するのが困難である。

また、lgg の本体から余分なアトムを取り除く作業を特別な工夫をしないで行なうと、その作業に必要な計算量は本体の長さに対して指数関数的である。計算量の爆発を避けるために GOLEM では 3 つのヒューリスティックスを用いている。その結果システムが求められる節の形が制限されているが、その制限は直観的には不明瞭である。

このようにシステムの能力を直観的に理解できないのも GOLEM の問題点の一つである。

我々のシステムでは証明可能なアトムの lgg を計算し、それらを組み合わせて探索すべき節空間を生成する。本体が短い節を先に調べることができるので、比較的短い節は効率良く求めることができる。

比較的長い節も求めたい場合には、lgg の組み合わせの数が爆発的に増えないように探索空間を制御できると都合が良い。我々のシステムでは節の記述に使用可能な述語記号、関数記号、定数と関数の入れ子の深さの上限で lgg の計算を制限している。この制限によって計算される lgg の数が少なくなるので、組み合わせの数を押えることができ比較的長い節も探索できる。

経験的には、興味ある節では現れる関数の数は少なく関数の入れ子の深さは小さい。既存の ILP システムで求められた節 [5] のほとんどには関数は現れない。現れてもほとんどの場合関数は 1 つで入れ子の深さは 1 である。従って上の制限は不都合なものではないと考えられる。しかもこの制限は直観的に理解しやすい。

ところで、現在の我々のシステムではアトムの集合を用いて証明可能なアトムの lgg を計算している。従って、それらのすべてを計算している保証はなく、システムが求められる節が制限されている可能性がある。

lgg を計算するためには様々なアトムが背景知識に含まれていれば良く、少数のアトムがあれば十分である。しかも lgg の計算に制限を加えるから、その制限の範囲内にあるすべての lgg を計算している可能性は高い。従って lgg の計算方法によるシステムの能力の制限はほとんどないと考えられる。

また、計算されていない lgg があっても、計算された lgg を組み合わせた形の節は必ず探索する。つまり、システムが最初に計算した lgg を調べることにより、システムが確実に求められる節の範囲を直観的に理解しやすい形で知ることができる。

以上のように我々のシステムは能力を直観的に理解しやすいという点で扱いやすいシステムである。

6 まとめ

本報告では我々が現在開発中の帰納推論システムについて述べた。

我々のシステムで最も重要なことは、背景知識を用いて証明可能なアトム全体の集合の部分集合の lgg を計算することである。これによって効率が良く直観的に理解しやすいシステムになっている。

このシステムは完全な帰納推論規則系 $\mathcal{R}$ に基づいており、完全性をできるだけ損なわないように開発されている。特に、システムの効率が十分良いので、既存の ILP システムのようにヒューリスティックスを用いる必要がなく、述語の規則性を表す節も求めることが可能である。

今後の課題としては、まず本稿で述べた試作版の制限をはずしてシステムを完成させなければならない。また、背景知識としてプログラムが与えられた場合など、背景知識にアトムが十分含まれていない場合にも lgg を計算できなければならない。

現在のシステムは新しい述語を可能な限り導入して節を生成している [3]。この方法でも *reverse* の例では特に問題は起こらない。しかし、*append* の場合は不必要な節を多数生成してしまう。これを防ぐために新しい述語を導入するための基準を明らかにする必要がある。

我々のシステムは目標とする節を特別な形のものに制限せずに、広い範囲の節を求めることができるシステムである。今後は例えばプログラムのような制限された形の節を効率良く求めるための仕組みを追加して、目的に応じた節を求められるようにする必要がある。既存のシステムで用いられている手法のほとんどは我々のシステムにも適用できると考えられ、それらを用いることで効率の良いシステムを構築できるはずである。

参考文献

- [1] Arimura, H. and Shinohara, T.: Polynomial Time Inference of Unions of Tree Pattern Languages, *Proc. of the 2nd International Workshop on Algorithmic Learning Theory*, pp. 105-114 (1991).
- [2] Kijssirikul, B., Numao, M. and Shimura, M.: Efficient Learning of Logic Programs with Non-determinate, Non-discriminating Literals, in Muggleton, S. (ed.), *Inductive Logic Programming*, Academic Press, pp. 361-372 (1992).
- [3] ブンサーム・キッスリクン, 沼尾 正行, 志村 正道: 弁別に基づく構成的帰納学習, 人工知能学会誌, Vol. 7, No. 6, pp. 1027-1036 (1992).
- [4] Muggleton, S. and Feng, C.: Efficient Induction of Logic Programming, *Proc. of the 1st International Workshop on Algorithmic Learning Theory*, pp. 368-381 (1990).
- [5] Muggleton, S. (ed.): *Inductive Logic Programming*, Academic Press (1992).
- [6] Plotkin, G.D.: A Note on Inductive Generalization, in Meltzer, B. and Michie, D. (eds.), *Machine Intelligence 5*, Edinburgh University Press, pp. 153-163 (1969).
- [7] Sato, T.: A Complete Set of Rules for Inductive Inference, *Electrotechnical Laboratory report TR-92-44* (1992).
- [8] Shapiro, E.Y.: *Algorithmic Program Debugging*, MIT Press (1982).