ICOT Technical Memorandom: TM-1182

TM-1182

International Workshop on Inductive Logic Programming (ILP92)

by
S. Muggleton & K. Furukawa

July, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

INTERNATIONAL CONFERENCE ON
FIFTH GENERATION COMPUTER SYSTEMS 1992

INTERNATIONAL WORKSHOP ON
INDUCTIVE LOGIC PROGRAMMING
(ILP92)

June 6-7, 1992 Tokyo, Japan

PROCEEDINGS

Institute for New Generation Computer Technology

Program Chairman

Stephen Muggleton
Turing Institute
George House, 36 North Hanover Street
Glasgow G1 2AD, U.K.

Local Chairman

Koichi Furukawa
Institute for New Generation Computer Technology (ICOT)
Mita Kokusai Bldg. 21F
4-28 Mita 1-chome, Minato-ku
Tokyo 108, Japan

Contents

Theory

Implementations, Experiments and Applications

Constraint-directed generalization for learning spatial relations
Mizoguchi F., Ohwada H.

Automated debugging of logic programs via theory revision
Mooney R., Richards B.

Correcting multiple faults in the concept and subconcepts by learning and abduction
Tangkitvanich S., Numao M., Shimura M.

Learning optimal KRK strategies
Bain M.

Drug design by machine learning
King R., Muggleton S., Lewis R., Sternberg M.

Protein secondary structure prediction using logic
Muggleton S., King R., Sternberg M.

Testing the applicability of ILP systems
Morales E.

Automatically constructing control systems by observing human behaviour
Sammut C.

Theory

A Theory of Predicate Invention

Kiyoshi Akama
Department of Information Engineering
Faculty of Engineering, Hokkaido University
kita-ku, Sapporo, 060 Japan

Predicate invention is a key procedure of inductive learning. But it does not have enough theoretical basis and its applicability is restricted to a few data structures and a few knowledge representation systems. In this paper we propose a theory of predicate invention, which has the following significant features.

- There are two methods of predicate invention. One is Common Structure Procedure (CSP) and the other is Common Component Procedure (CCP). The intra-construction operator by Muggleton and Buntine corresponds to CSP, but not to CCP.
- Predicate invention in this paper is separated from hypothesis generation, and is more accurately formalized by theorems of equivalent transformation of programs.
- Simple hypothesis formation associated with predicate invention can give plausible hypotheses which may be generated by the truncation and absorption operators.
- The theory in this paper is based on GLP theory, therefore it gives a unified theory of predicate invention for diverse knowledge representation systems and data structures, especially logic programs with terms (pure Prolog), logic programs with strings and constraint logic programs.

1 Introduction

Predicate invention is a key procedure of inductive learning. In this paper we propose a theory of predicate invention. There are two major problems in the existing methodologies of predicate invention.

One problem is that they do not give enough theoretical basis to predicate invention. In the paper of Muggleton and Buntine [12], predicate invention is based on the principle of inverse resolution. Inverse resolution is basically a framework for hypothesis generation. Predicate invention is associated with a single operator of intra-construction. Other operators such as absorption are not related to predicate invention. In the process of creating these operators, inverse resolution depends on what is called simplifying assumptions.

But they seem to play more roles than the true meaning of the word and their roles are not so clear, which makes it more difficult to understand fully what predicate invention is.

Different view of predicate invention has been adopted in the research of inductive learning systems of LS/0 [1], LS/1 [2] and LS/2 [3]. In all these systems, predicate invention is also one of the most important procedures to improve their knowledge. More accurately, in these systems, predicate invention plays a more fundamental role in structuralizing the knowledge than CIGOL. New predicates are invented in LS/i (i=0,1,2) in terms of equivalent transformation of knowledge. Hypothesis rules which may be obtained by using the truncation or absorption operator in CIGOL can be generated in LS/i (i=0,1,2) by adding simple hypotheses to the results of predicate invention.

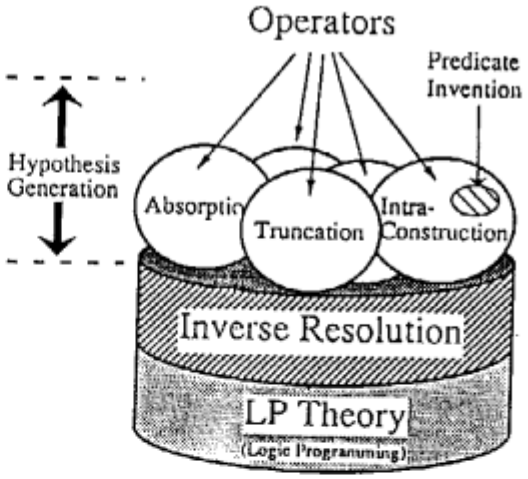


Figure 1: Old framework

The theory of predicate invention in this paper is a formalization of the methodology used in LS/i (i=0,1,2). It has the following characteristics.

- There are two methods of predicate invention. One is Common Structure Procedure (CSP) and the other is Common Component Procedure (CCP). The intra-construction operator by Muggleton and Buntine corresponds to CSP, but not to CCP.
- Predicate invention in this paper is separated from hypothesis generation, and is more accurately formalized by theorems of equivalent transformation of programs.
- Simple hypothesis formation associated with predicate invention can give plausible hypothesis rules which may be generated by the truncation and absorption operators.

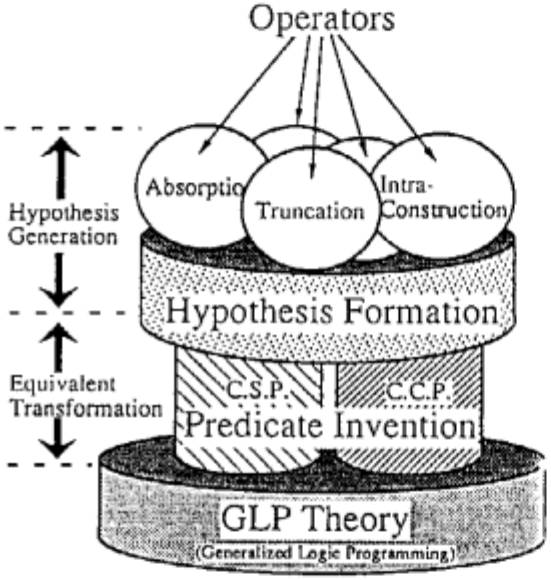


Figure 2: New framework

The other problem for the existing methodologies of predicate invention is the lack of wide applicability of the theory. Because there has been no theory which can effectively unify many knowledge representation systems and data structures, the research on learning has had to produce many slightly different theories each of which deals with one kind of knowledge representation system or data structure. There is a similar problem in the case of inductive learning and predicate invention [1, 13, 2, 11, 12, 10]. This problem is overcome in this paper by the theory of generalized logic programs (GLP theory) [4, 5], because

- GLP theory is general enough to unify diverse knowledge representation systems and data structures, including logic programs with terms (pure Prolog), logic programs with strings (including context free grammers) and constraint logic programs [8].
- GLP theory is not too abstract to effectively explain important common structures of inductive learning algorithms. Based on the common theory we can construct specialized theories and effective algorithms by using specific knowledge of each data structure and knowledge representation system.

In this paper we first give in section 2 an illustrative example of the application of the theory in this paper. Next we review GLP theory. We introduce the definition of specialization systems in section 3 and give some examples of specialization systems in section 4. We summerize the declarative semantics of generalized logic programs on specialization systems in section 5. We introduce basic specialization systems and show theorems for inference rules in section 6. We omit the procedural semantics of generalized logic programs on specialization systems because it is not necessary for the discussion in this paper.

We give a theory of predicate invention in section 7. The theory is applied to the domain of terms in section 8 and to the domain of strings in section 9. Section 10 is the conclusion.

All the proofs [1] are omitted here. We use notations from [4, 5]. Especially, $powerset(X)$ denotes the powerset of a set X. $map(X, Y)$ denotes the set of all mappings from a set X to a set Y. $map(X)$ is equivalent to $map(X, X)$. $partial\_map(X, Y)$ denotes the set of all partial mappings from a set X to a set Y. $partial\_map(X)$ is defined to be $partial\_map(X, X)$. Composition of partial mappings are denoted by $\circ$, that is, for two partial mappings f and g

$$f \circ g = \{(x, z) \mid (x, y) \in g, (y, z) \in f\}$$

2 An Illustrative Example

2.1 Rule Generation by Absorption Operator

The following example was used to illustrate the absorption operator in [12].

```
Input:
  member(A,[A|B]).                    (1)
  member(A,[B,A|C]).                  (2)
Output:
  member(A,[B|C]) :- member(A,C).     (3)
```

We will explain the outline of our theory by using this example.

2.2 Formalization Using Generalized Logic Programs

In our theory the atom [2] `member(A,[A|B])` is mathematically represented by the pair (member,

[1] For detailed discussion and proofs, see [6, 4, 5].
[2] Atomic formulas are called simply atoms in this paper.

(A,[A|B])). The first fact (1) in the input is represented in our theory by

(member, (A,[A|B])) ←

Therefore, we consider that the input of our problem is a program $P_a = P \cup \{C_1^A, C_2^A\}$, where C_1^A and C_2^A are the following clauses

C_1^A = (member, (A,[A|B])) ←

C_2^A = (member, (A,[B,A|C])) ←

and P is the rest of the program which is not mentioned explicitly in the above example.

Similarly the output clause (3) obtained by the absorption operator is represented by a clause C^Z,

C^Z = (member, (A,[B|C])) ← (member, (A,C)).

We deal with the generation of the rule (3) in two steps [3].

1. The program P_a is equivalently transformed to a program P_b.
2. The program P_b becomes a program P_c by adding hypothesis clauses.

From the program P_c we can deduce C^Z, that is, $P_c \models C^Z$.

2.3 Equivalent Transformation

Our first subgoal is to find a new program P_b which is equivalent [4] to the input program P_a and includes invented predicates. One such program is shown bellow,

$P_b = P \cup \{C_1^C, C_2^C\} \cup \{C_1^D, C_2^D\}$

where

C_1^C = (member, (X,Y)) ← (v098, (X,Y))

C_2^C = (member, (Z,[B|W])) ← (v099, (Z,W))

C_1^D = (v098, (A,[A|C])) ←

C_2^D = (v099, (A,[A|C])) ←

v098 and v099 are newly invented predicates which are not used in P_a. Note that two predicates are invented at the same time, while the absorption operator invents no predicate and the intra-construction operator invents only one predicate.

Next we show that P_a and P_b are equivalent from a declarative point of view. For any program P_r we denote the declarative meaning [5] of a program P_r by $REP(P_r)$. In this case, we find

$$REP(P_b) = REP(P_a) \cup REP(\{C_1^D\}) \cup REP(\{C_2^D\})$$

Let R be the set of all predicates used in the program P_a and let $\mathcal{G}_t$ be the set of all pairs of ground terms, then,

member $\in R$

v098 $\notin R$

v099 $\notin R$

$REP(\{C_1^D\}) \cap (R \times \mathcal{G}_t) = \emptyset$

$REP(\{C_2^D\}) \cap (R \times \mathcal{G}_t) = \emptyset$

$$REP(P_a) \cap (R \times \mathcal{G}_t) = REP(P_b) \cap (R \times \mathcal{G}_t) \quad (4)$$

When (4) holds, we say that P_a and P_b are R-equivalent. In this sense P_a is equivalently transformed to P_b, inventing two predicates.

2.4 Hypothesis Formation

Next we demonstrate that the predicate invention described above leads us naturally to the output clause (3) which is obtained by the absorption operator. First we introduce the following two hypothesis clauses.

C_{12}^H = (v098, (X,Y)) ← (v099, (X,Y))

C_{21}^H = (v099, (X,Y)) ← (v098, (X,Y))

Adding these hypotheses is natural [6] because both v098 and v099 represent the same relation at this point [7], that is,

$\{ g \mid (\text{v098}, g) \in REP(\{C_1^D\}) \}$

$= \{ g \mid (\text{v099}, g) \in REP(\{C_2^D\}) \}$.

Similarly we can also add a hypothesis clause

C^H = (v098, (X,Y)) ← (member, (X,Y))

which is the inverse clause of C_1^C. It is clear that from C_2^C, C_{21}^H and C^H we can deduce the following clause.

C^Z = (member, (A,[B|C])) ← (member, (A,C)).

which is equivalent to the result (3) which is obtained by the absorption operator.

2.5 Flexibility of the Output

The absorption operator returns a clause (3) which is equivalent to C^Z, but when the clause (3) turns out to be inappropriate it is difficult to modify the clause to get a new improved clause.

The framework in this paper returns a program

$P_c = P \cup \{C_1^C, C_2^C, C_1^D, C_2^D, C_{12}^H, C_{21}^H, C^H\}$

from which we can deduce C^Z. We can modify P_c by changing some of the hypothesis clauses in P_c without retracting C_1^C, C_2^C, C_1^D and C_2^D which include invented predicates.

In this sense our framework is more flexible than the inverse resolution in CIGOL.

3 Specialization Systems

GLP theory is important for the theory in this paper because it enables us to develop a unified theory of predicate invention in later sections. Here we review specialization systems on which GLP theory is constructed. See [5] for details.

3.1 Specialization Systems

By generalizing the structure of terms, grond terms and substitutions, we get a more abstract structure

[3] The intra-construction operator has only one step.

[4] Strictly they are R-equivalent, which will be defined soon.

[5] The declarative meaning of a program P is defined in section 5.2.

[6] We need not add hypotheses if we invent only one predicate instead of two. But hypothesis change is less flexible than the case of two predicates.

[7] When we get other information about the unknown predicate member later, two predicates may turn out to be a different relation. In this case we should change the hypotheses C_{12}^H and C_{21}^H. This is not possible for the case of inventing only one predicate instead of two.

called specialization systems.

Definition 1 A *specialization system* is a 4-tuple $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ that satisfies the following conditions.

(1) $\mu : \mathcal{S} \to partial\_map(\mathcal{A})$

(2) $\forall s_1, s_2 \in \mathcal{S}, \exists s \in \mathcal{S} : \mu(s) = \mu(s_2) \circ \mu(s_1)$

(3) $\exists s \in \mathcal{S}, \forall a \in \mathcal{A} : \mu(s)(a) = a$

(4) $\mathcal{A} \supset \mathcal{G}$

Elements of $\mathcal{A}$ are called objects or atoms. $\mathcal{G}$ is called the interpretation domain of Γ. Elements of $\mathcal{S}$ are called *specializations*. The specializations that satisfy (3) are called *identity specializations*, and any one of them [8] is denoted by ϵ. A specialization $s \in \mathcal{S}$ is said to be applicable to $a \in \mathcal{A}$ iff there exists $b \in \mathcal{A}$ such that $(a, b) \in \mu(s)$.

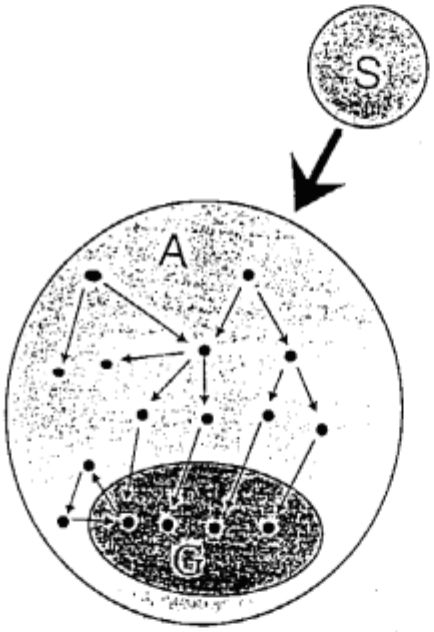


Figure 3: A Specialization System

From the definition, μ is a subset of $\mathcal{S} \times partial\_map(\mathcal{A})$. As each element in $partial\_map(\mathcal{A})$ is a subset of $\mathcal{A} \times \mathcal{A}$, μ is a subset of $\mathcal{S} \times powerset(\mathcal{A} \times \mathcal{A})$. We often regard μ as a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$, because $\mu \subset \mathcal{S} \times powerset(\mathcal{A} \times \mathcal{A})$ and $\nu \subset \mathcal{S} \times \mathcal{A} \times \mathcal{A}$ determine each other uniquely by the following equations.

$$\mu = \{(s, M) \mid M = \{(a, b) \mid (s, a, b) \in \nu\}\}$$
$$\nu = \{(s, a, b) \mid (s, M) \in \mu, (a, b) \in M\}$$

When there is no danger of confusion, we regard elements in $\mathcal{S}$ as partial mappings over $\mathcal{A}$, and use the following notational convention. Each element in $\mathcal{S}$ which is identified as a partial mapping on $\mathcal{A}$ is denoted by a Greek letter such as θ, and the application of such a partial mapping is represented by postfix notation. For example, $s \in \mathcal{S}, \mu(s)(a)$ and $\mu(s_n) \circ \mu(s_{n-1}) \circ \cdots \circ \mu(s_1)$ are denoted respectively by $\theta \in \mathcal{S}, a\theta$ and $\theta_1 \circ \theta_2 \circ \cdots \circ \theta_n$. The composition operator $\circ$ is often omited.

In the following discussion we fix a specialization system $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$.

Definition 2 A mapping $rep : \mathcal{A} \to powerset(\mathcal{G})$ is defined by $rep(a) = \{g \mid g = a\theta \in \mathcal{G}, \theta \in \mathcal{S}\}$.

4 Examples of Specialization Systems

We give three basic specialization systems and two specialization construction schemata. We also introduce two specializations systems which will be used in later sections.

4.1 Domain of Symbols

Domain of symbols form a simple specialization system, where symbols are constants in the sense that there is no specialization except the identity specialization ϵ which does not change symbols.

Proposition 1 Let R be any set and μ_R be a mapping from $\{\epsilon\}$ to $map(R)$ such that $\forall r \in R : \mu_R(\epsilon)(r) = r$. Then $< R, R, \{\epsilon\}, \mu_R >$ is a specialization system.

The specialization system which is constructed from an arbitrary set R by proposition 1 is denoted by Γ_R.

4.2 Domain of Terms

Let V, K and F be mutually disjoint sets. Each element of V, K and F is called respectively, a *variable*, a *constant* and a *function*. Each function in F is associated with a positive integer called *arity*. *Terms* and *substitutions* are defined as usual. The set of all terms over V, K and F is denoted by $Term(V, K, F)$. The set of all terms over K and F is denoted by $Term(\emptyset, K, F)$. The set of all substitutions over V, K and F is denoted by $Subst_t(V, K, F)$ [9]. Application of a substitution $\theta \in Subst_t(V, K, F)$ to terms defines a mapping M_θ over $Term(V, K, F)$. The mapping, $\mu_t : Subst_t(V, K, F) \to map(Term(V, K, F))$ is also defined to give such a mapping M_θ for each substitution θ.

The following proposition shows that terms and substitutions form a specialization system.

Proposition 2 Let $\mathcal{A}_t$ be $Term(V, K, F)$, $\mathcal{G}_t$ be $Term(\emptyset, K, F)$ and $\mathcal{S}_t$ be $Subst_t(V, K, F)$. Then the 4-tuple $< \mathcal{A}_t, \mathcal{G}_t, \mathcal{S}_t, \mu_t >$ is a specialization system.

The specialization system which is constructed in proposition 2 is denoted by Γ_t in this paper.

[8] There may be more than one identity specializations.

[9] The subscript t means terms.

4.3 Domain of Strings

Let V, K be mutually disjoint sets. Each element of V and K is called respectively, a *variable* and a *constant*. *Strings* and *substitutions* are defined as usual. The set of all strings over V and K is denoted by $String(V \cup K)$. The set of all strings over K is denoted by $String(K)$. The set of all substitutions over V and K is denoted by $Subst_s(V, K)$ [10]. Application of a substitution $\theta \in Subst_s(V, K)$ to strings defines a mapping M_θ over $String(V \cup K)$. The mapping, $\mu_s : Subst_s(V, K) \rightarrow map(String(V \cup K))$ is also defined to give such a mapping M_θ for each substitution θ.

The following proposition shows that strings and substitutions form a specialization system.

Proposition 3 Let $\mathcal{A}_s$ be $String(V \cup K)$, $\mathcal{G}_s$ be $String(K)$ and $\mathcal{S}_s$ be $Subst_s(V, K)$. Then the 4-tuple $< \mathcal{A}_s, \mathcal{G}_s, \mathcal{S}_s, \mu_s >$ is a specialization system.

The specialization system which is constructed in proposition 3 is denoted by Γ_s in this paper.

4.4 Specialization Construction Schemata

We can construct new specialization systems from more basic ones.

Proposition 4 Let $\Gamma_1 =< \mathcal{A}_1, \mathcal{G}_1, \mathcal{S}_1, \mu_1 >$ and $\Gamma_2 =< \mathcal{A}_2, \mathcal{G}_2, \mathcal{S}_2, \mu_2 >$ be specialization systems. Let also $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$, $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2$, $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$ and $\mu : \mathcal{S} \rightarrow partial\_map(\mathcal{A})$ be defined by [11]

$$\mu = \{((s_1, s_2), (a_1, a_2), (b_1, b_2)) \mid (s_1, a_1, b_1) \in \mu_1, (s_2, a_2, b_2) \in \mu_2\}.$$

Then $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ is a specialization system.

The specialization system which is constructed from Γ_1 and Γ_2 by proposition 4 is denoted by $\Gamma_1 \times \Gamma_2$.

Definition 3 Let X be any set. $Tuple(X)$ is defined to be the set of all tuples of elements in X, that is,

$$Tuple(X) = X^0 + X^1 + X^2 + \cdots$$

For example, when $X = \{a, b\}$,

$$Tuple(X) = \{(), (a), (b), (a, a), (a, b), (b, b), \ldots\}.$$

Proposition 5 Let $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ be a specialization system. Then

$$< Tuple(\mathcal{A}), Tuple(\mathcal{G}), \mathcal{S}, \nu >$$

is a specialization system, where $\nu : \mathcal{S} \rightarrow partial\_map(Tuple(\mathcal{A}))$ is defined by [12]

$$\nu = \nu_0 \cup \nu_1 \cup \nu_2 \cup \cdots$$

$$\nu_n = \{(s, (a_1, a_2, \ldots, a_n), (b_1, b_2, \ldots, b_n)) \mid \forall i \in \{1, 2, \ldots, n\} : (s, a_i, b_i) \in \mu\}$$

[10] The subscript s represents strings.

[11] Using one to one correspondence mentioned in section 3 we regard μ, μ_1 and μ_2 respectively as a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$, $\mathcal{S}_1 \times \mathcal{A}_1 \times \mathcal{A}_1$ and $\mathcal{S}_2 \times \mathcal{A}_2 \times \mathcal{A}_2$.

[12] Using one to one correspondence mentioned in section 3 we regard μ and ν respectively as a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{A}$ and $\mathcal{S} \times Tuple(\mathcal{A}) \times Tuple(\mathcal{A})$.

The specialization system which is constructed from Γ by proposition 5 is denoted by $Tuple(\Gamma)$.

4.5 Specialization Systems used in Examples

Let R be a set of symbols such as `member`, `arch`, `tr`, `v189`, `v002`, `v-root`, `v561`, `v248` and so on. We use elements of R as predicates. Using basic specializations (Γ_R, Γ_t and Γ_s) and specialization construction schemata ($\Gamma_1 \times \Gamma_2$ and $Tuple(\Gamma)$) introduced above, we can construct two specialization systems, $\Gamma_R \times Tuple(\Gamma_t)$ and $\Gamma_R \times Tuple(\Gamma_s)$.

$\Gamma_R \times Tuple(\Gamma_t)$ is a specialization system which can be used to represent atoms in Prolog. For example a Prolog atom `member(X,[2 3 4])` is mathematically dealt with [13] in this paper as `(member, (X,[2,3,4]))` which is an element in $\Gamma_R \times Tuple(\Gamma_t)$, where `[2,3,4]` is an abbreviation of `cons(2,cons(3,cons(4,nil)))` which consists of `cons` $\in F$, `nil` $\in K$ and numbers in K.

$\Gamma_R \times Tuple(\Gamma_s)$ is a specialization system which is used to represent atoms with string arguments. For example

```
tr([little girl],[CHIISANA SHOUJO])
```

is an atom with string arguments [14], which means that `[CHIISANA SHOUJO]` is a Japanese translation of `[little girl]`. In the following theoretical discussions in this paper, this atom is dealt with as

```
(tr, ([little girl],[CHIISANA SHOUJO]))
```

which is an element in $\Gamma_R \times Tuple(\Gamma_s)$, where `[little girl]` and `[CHIISANA SHOUJO]` are constant strings.

5 Declarative Semantics of Logic Programs on Specialization Systems

In this section we give the outline of declarative semantics of logic programs on specialization systems. Due to space limitations, we omit most of the definitions. See [4].

5.1 Logic Programs on Specialization Systems

Using logical connectives ($\neg$, $\wedge$ and $\vee$) and quantifiers ($\forall$ and $\exists$), we can define formulas on an arbitrary set. Classifying formulas into subclasses, we can introduce closed formulas, clauses, program clauses on the given set. Especially, the set of all program clauses on an arbitrary set X is denoted by $PC(X)$, that is,

$$PC(X) = \{H \leftarrow B_1, \ldots, B_n \mid H, B_1, \ldots, B_n \in X\}$$

[13] Symbol atoms which begin with capital letter represent variables in the case of terms.

[14] `[little girl]` is not a list of words but a sequence of words. Such structures are called strings from the theoretical viewpoint.

Let $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ be a specialization system. Formulas, closed formulas, clauses and program clauses on $\mathcal{A}$ are also called, respectively, formulas, closed formulas, clauses and program clauses on Γ.

Definition 4 A *logic program* on a specialization system $< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ is a (possibly infinite) set of program clauses on $\mathcal{A}$.

A logic program on a specialization system is often called a *generalized logic program* (GLP) in order to stress the distinction between ordinary logic programs and generalized ones.

5.2 Declarative Semantics of Logic Programs on Specialization Systems

In order to discuss semantics of closed formulas on specialization systems, we define interpretations, which determine the truth or falsity of all closed formulas. Models and logical consequences are also defined in terms of interpretations.

Let $Program(\Gamma)$ be the set of all logic programs on a specialization system Γ, and let $REP : Program(\Gamma) \to powerset(\mathcal{G})$ be a mapping which determines, for each element P in $Program(\Gamma)$, any one [15] of

1. the minimal model M_P of P,
2. the least fixpoint of K_P [16] , and
3. $K_P \uparrow \omega$ [17] .

Then $< Program(\Gamma), \mathcal{G}, REP >$ is a representation system in the sense that a program P represents $REP(P)$ which is a subset of $\mathcal{G}$. $REP(P)$ is also called the meaning of the program P.

6 Basic Specialization Systems

6.1 Basic Specialization Systems

In usual logic, the inference rule of resolution [18] is always sound. But it is not always sound in the case of generalised logic programs. We have already proposed a sufficient condition for the safe application of the resolution rule:

Theorem 1 Let Γ be a *safe specialization system*. If C_1 and C_2 be *homogeneous clauses* on Γ and
$C_1 = (H \leftarrow A_1, \ldots, A_i, \ldots, A_q)$
$C_2 = (K \leftarrow B_1, \ldots, B_n)$
$\theta \in S$ be applicable to $H, A_1, \ldots, A_i, \ldots, A_q$.
$\sigma \in S$ be applicable to $K, B_1, \ldots, B_n$.
$A_i\theta = K\sigma$
$C_3 = (H\theta \leftarrow A_1\theta, \ldots, A_{i-1}\theta, B_1\sigma, \ldots, B_n\sigma, A_{i+1}\theta, \ldots, A_q\theta)$
then C_3 is also a homogeneous clause on Γ and $\{C_1, C_2\} \models C_3$.

To improve the readability, we do not define here *safe specialization systems* and *homogeneous clauses* in the theorem 1. Instead of explaining rather complicated definitions for theorem 1, we introduce a small class of specialization systems and show a less general but more easily understandable theorem.

Definition 5 A specialization system $\Gamma = < \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ is *basic* iff

(1) $\forall a \in \mathcal{A}, \forall \theta \in \mathcal{S} : \theta$ is applicable to a.

(2) $\forall g \in \mathcal{G}, \forall \theta \in \mathcal{S} : g\theta = g$

(3) $\forall a \in \mathcal{A}, \exists \theta \in \mathcal{S} : a\theta \in \mathcal{G}$

Specialization systems for logic programs with terms and logic programs with strings are *basic* because all specializations are applicable to all atoms, their interpretation domains are sets of ground atoms, ground atoms are not changed by specializations and all atoms can be grounded by some specializations.

Specialization systems for logic programs with terms and strings are formally dealt with, respectively, by $\Gamma_R \times Tuple(\Gamma_t)$ and $\Gamma_R \times Tuple(\Gamma_s)$.

6.2 Inference Rules for Basic Specialization Systems

We give inference rules for basic specialization systems.

Theorem 2 Let Γ be a basic specialization system and assume that
$C_1 = (H \leftarrow A_1, \ldots, A_i, \ldots, A_q)$
$C_2 = (A_i \leftarrow B_1, \ldots, B_n)$
$C_3 = (H \leftarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_q)$
are clauses on Γ. Then, $\{C_1, C_2\} \models C_3$.

Theorem 3 Let Γ be a basic specialization system and assume that
$C_1 = (H \leftarrow B_1, \ldots, B_n)$
$C_2 = (H\theta \leftarrow B_1\theta, \ldots, B_n\theta)$
are clauses on Γ. Then, $C_1 \models C_2$.

These theorems lead us to the following familier rule of resolution.

Resolution: Let Γ be a basic specialization system.
Let C_1 and C_2 be clauses on Γ and
$C_1 = (H \leftarrow A_1, \ldots, A_i, \ldots, A_q)$
$C_2 = (K \leftarrow B_1, \ldots, B_n)$
Let (θ, σ) be a unifier of A_i and K, that is, $A_i\theta = K\sigma$, and
$C_3 = (H\theta \leftarrow A_1\theta, \ldots, A_{i-1}\theta, B_1\sigma, \ldots, B_n\sigma, A_{i+1}\theta, \ldots, A_q\theta)$
Then $\{C_1, C_2\} \models C_3$.

[15] They are equal.
[16] $K_P \in map(powerset(\mathcal{G}))$ is the *knowledge-increasing transformation* of P, which is defined as the sum of T_P *(one-step-inference transformation)* and I_d *(the identity mapping)*.
[17] $K_P \uparrow \omega = lub\{K^n(\emptyset) \mid n \geq 0\}$.
[18] See 6.2.

7 Theory of Predicate Invention

We give a theory of predicate invention based on GLP theory.

7.1 Basic Definitions

We assume that a specialization system $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$ is given.

Definition 6 $\theta \in \mathcal{S}$ is *applicable* to $W \subset \mathcal{A}$ iff θ is applicable to all elements in W. $\theta \in \mathcal{S}$ is *applicable* to a clause C iff θ is applicable to all atoms in C. When C is a clause $H \leftarrow B_1, \ldots, B_n$ and θ is applicable to C, $C\theta$ is defined as $H\theta \leftarrow B_1\theta, \ldots, B_n\theta$.

Definition 7 A specialization $\rho \in \mathcal{S}$ is a *renaming specialization* (or simply a *renaming*) iff there is an identity specialization ϵ such that $\exists \tau \in \mathcal{S} : \rho\tau = \epsilon$.

Definition 8 Let $X, x \in \mathcal{A}$ and $C^X, C^x \in PC(\mathcal{A})$. Then, $(X : x) \rhd (C^X : C^x)$ iff

$$[\forall \tau \in \mathcal{S} : [(X\tau \in rep(x), C^X\tau \in PC(\mathcal{G})) \rightarrow (\exists \rho \in \mathcal{S} : C^x\rho = C^X\tau)]]$$

For example $(\mathtt{X} : 5) \rhd (\mathtt{f(X,Y)} \leftarrow : \mathtt{f(5,Z)} \leftarrow)$.

Definition 9 Let P be a program on Γ. $Head(P)$ is the set of all heads of clauses in P. $Body(P)$ is the set of all atoms in bodies of clauses in P. $Atom(P)$ is the set of all atoms in clauses in P.

Definition 10 Let $D \subset \mathcal{G}$. $REP(P; D)$ is defined as $REP(P) \cap D$.

We also assume that two specialization systems $\Gamma_x =< \mathcal{A}_x, \mathcal{G}_x, \mathcal{S}_x, \mu_x >$ and $\Gamma_y =< \mathcal{A}_y, \mathcal{G}_y, \mathcal{S}_y, \mu_y >$ are given.

Definition 11 Let $C^x = H^x \leftarrow B^x_1, \ldots, B^x_n$ be a clause on Γ_x and $C^y = H^y \leftarrow B^y_1, \ldots, B^y_n$ be a clause on Γ_y. $C^x \times C^y$ is defined as $(H^x, H^y) \leftarrow (B^x_1, B^y_1), \ldots, (B^x_n, B^y_n)$, which is a clause on $\Gamma_x \times \Gamma_y$.

7.2 Two procedures for Predicate Invention

Let $\mathcal{R}$ be a set of predicates and Γ_R be a specialization system $< \mathcal{R}, \mathcal{R}, \{\epsilon\}, \mu_R >$ which was defined in proposition 1. Also let $\Gamma =< \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu >$. From proposition 4 we get a specialization system $\Gamma_R \times \Gamma$. As there is one-to-one correspondence between $\{\epsilon\} \times \mathcal{S}$ and $\mathcal{S}$, $(\epsilon, \theta) \in \{\epsilon\} \times \mathcal{S}$ is often referred to by θ in $\mathcal{S}$. In the following we deal with programs on $\Gamma_R \times \Gamma$.

Definition 12 Let $R \subset \mathcal{R}$. A program P is on R iff all atoms in P are elements in $R \times \mathcal{A}$.

Definition 13 $REP(P \mid R)$ is defined as $REP(P; R \times \mathcal{G})$. A program P_1 and a program P_2 are R-equivalent iff $REP(P_1 \mid R) = REP(P_2 \mid R)$.

7.2.1 Common Structure Procedure

We give a procedure for predicate invention, called Common Structure Procedure (CSP). The intra-construction operator for predicate invention corresponds to CSP.

Procedure 1 Given a program P_a on R which consists of a program P on R and clauses

$C^A_i = (C^r \times C^a_i) \quad (i \in I)$

where I is a non-empty finite set. First find C^b, W, w_i $(i \in I)$, θ_i $(i \in I)$ and ρ_i $(i \in I)$ such that

$C^b\theta_i\rho_i = C^a_i \quad (i \in I)$

ρ_i $(i \in I)$ are renamings

$(W : w_i) \rhd (C^b : C^a_i) \quad (i \in I)$

$W\theta_i = w_i \quad (i \in I)$

Then select r such that $r \in \mathcal{R} - R$ and let

$C^C = (C^r \Leftarrow r) \times (C^b \Leftarrow W)$

$C^D_i = (r \leftarrow) \times (w_i \leftarrow) \quad (i \in I)$

Then change the program

$P_a = P \cup \{C^A_i \mid i \in I\}$

to

$P_b = P \cup \{C^C\} \cup \{C^D_i \mid i \in I\}$.

Theorem 4 By the procedure 1 the given program P_a is R-equivalently transformed into P_b, that is,

$REP(P_a \mid R) = REP(P_b \mid R)$.

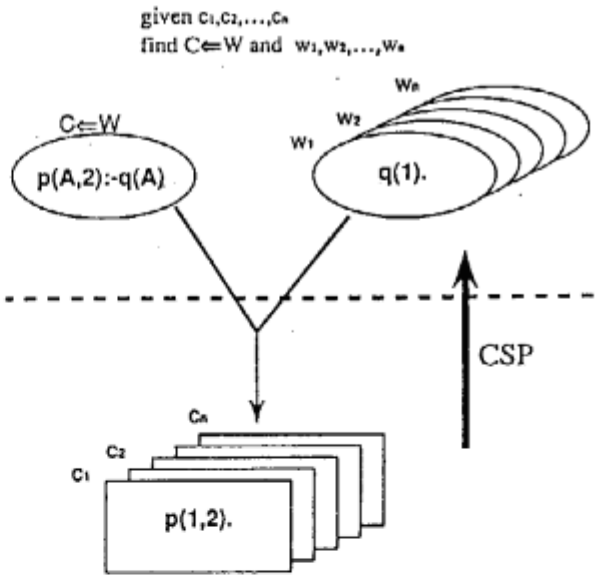


Figure 4: Problem for CSP

7.2.2 Common Component Procedure

We introduce another procedure, called Common Component Procedure (CCP).

Procedure 2 Given a program P_a on R which consists of a program P on R and clauses

$C^A_i = (C^r_i \times C^a_i) \quad (i \in I)$

where I is a non-empty finite set. First find C^b_i $(i \in I)$, W_i $(i \in I)$, w, θ_i $(i \in I)$ and ρ_i $(i \in I)$

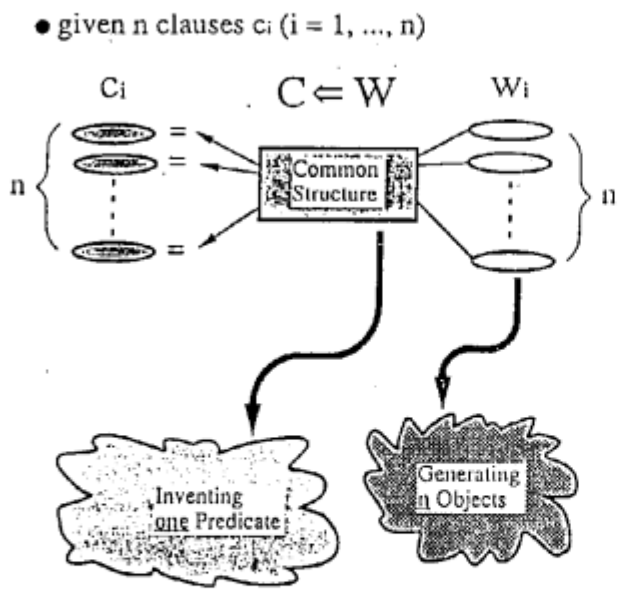


Figure 5: Common Structure Procedure

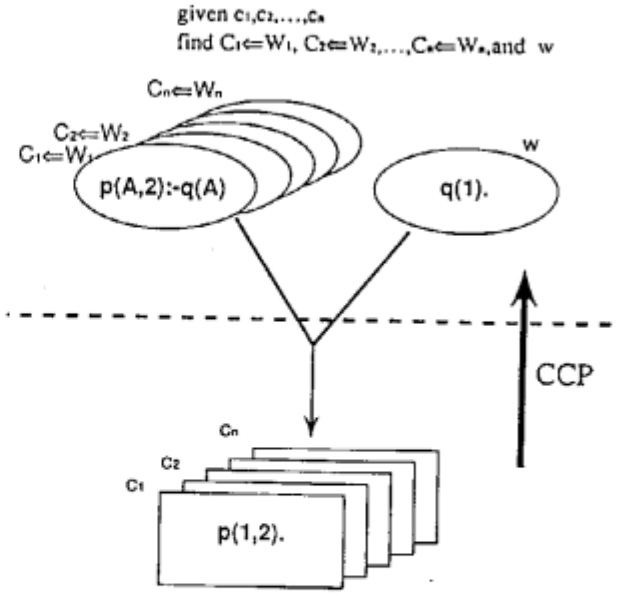


Figure 6: Problem for CCP

such that
$C_i^b\theta_i\rho_i = C_i^a \quad (i \in I)$
ρ_i $(i \in I)$ are renamings
$(W_i : w) \rhd (C_i^b : C_i^a) \quad (i \in I)$
$W_i\theta_i = w \quad (i \in I)$
Then select r_i $(i \in I)$ such that $r_i \in \mathcal{R} - R$ $(i \in I)$ and r_i $(i \in I)$ are different from each other. Let
$C_i^D = (r_i \leftarrow) \times (w \leftarrow) \quad (i \in I)$
$C_i^C = (C_i^r \Leftarrow r_i) \times (C_i^b \Leftarrow W_i) \quad (i \in I)$
Then change the program
$P_a = P \cup \{C_i^A \mid i \in I\}$
to
$P_b = P \cup \{C_i^C \mid i \in I\} \cup \{C_i^D \mid i \in I\}.$

Theorem 5 By the procedure 2 the given program P_a is R-equivalently transformed into P_b, that is,
$REP(P_a \mid R) = REP(P_b \mid R)$.

7.3 Two Problems for Predicate Invention

In the procedures 1 and 2 we find two important problems.

Problem for CSP Given $C_i^a \quad (i \in I)$, where I is a non-empty finite set. Find C^b, W, w_i $(i \in I)$, θ_i $(i \in I)$ and ρ_i $(i \in I)$ such that
$C^b\theta_i\rho_i = C_i^a \quad (i \in I)$
ρ_i $(i \in I)$ are renamings
$(W : w_i) \rhd (C^b : C_i^a) \quad (i \in I)$
$W\theta_i = w_i \quad (i \in I)$

In the Common Structure Procedure (CSP), the common part (C^b) of given clauses $(C_i^a$ for $i \in I)$ forms the main structure and the different parts $(w_i$ for $i \in I)$ are found to be new objects.

Problem for CCP Given clauses $C_i^a \quad (i \in I)$, where I is a non-empty finite set. Find C_i^b $(i \in I)$, W_i $(i \in I)$, w, θ_i $(i \in I)$ and ρ_i $(i \in I)$ such that
$C_i^b\theta_i\rho_i = C_i^a \quad (i \in I)$
ρ_i $(i \in I)$ are renamings
$(W_i : w) \rhd (C_i^b : C_i^a) \quad (i \in I)$
$W_i\theta_i = w \quad (i \in I)$

In the Common Component Procedure (CCP), the common part (w) is extracted as a component and different parts $(C_i^b \quad$ for $i \in I)$ are found to be new contexts. CCP is the "dual" procedure to CSP for inventing predicates.

8 Examples for the Domain of Terms

Here we use examples in the paper [12] of S. Muggleton and W. Buntine.

8.1 Intra-construction

The following example was used to illustrate the intra-construction operator. v189 is an invented predicate.

```
Input:
  arch([],beam,[]).
  arch([block],beam,[block]).
  arch([brick],beam,[brick]).
  arch([block,brick],beam,[block,brick]).
Output:
  arch(A,beam,A) :- v189(A).
  v189([]).
  v189([block]).
  v189([brick]).
  v189([block,brick]).
```

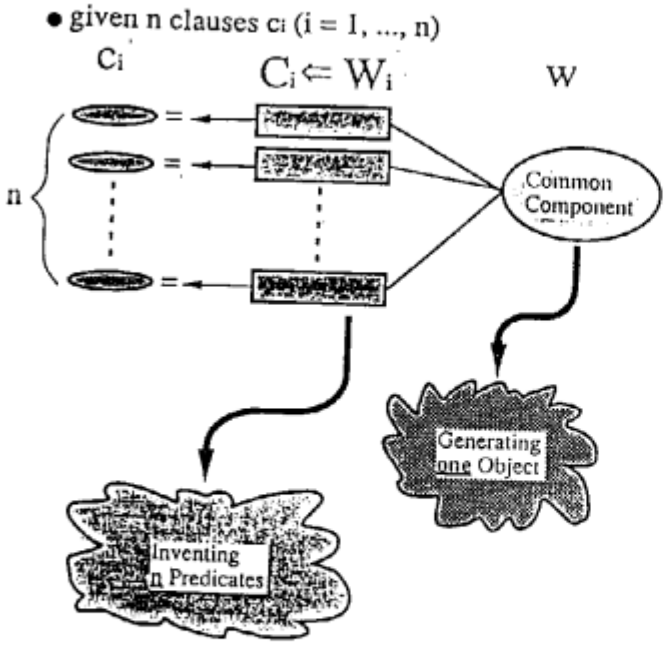


Figure 7: Common Component Procedure

We will explain this example by using the theory in this paper. Assume that we are given a program $P_a = P \cup \{C_1^A, C_2^A, C_3^A, C_4^A\}$, where

C_1^A = (arch, ([],beam,[])) ←
C_2^A = (arch, ([block],beam,[block])) ←
C_3^A = (arch, ([brick],beam,[brick])) ←
C_4^A = (arch, ([block,brick], beam,[block,brick])) ←

Let R be the set of all predicates in P_a, then arch $\in R$. We assume v189 $\in \mathcal{R} - R$. From C_1^A, C_2^A, C_3^A and C_4^A we know

C^r = arch ←
C_1^a = ([],beam,[]) ←
C_2^a = ([block],beam,[block]) ←
C_3^a = ([brick],beam,[brick]) ←
C_4^a = ([block,brick],beam,[block,brick]) ←

As one of the least general generalizations of C_1^a, C_2^a, C_3^a and C_4^a is (A,beam,A) ←, we can select $C^b, W, w_1, w_2, w_3, w_4, \theta_1, \theta_2, \theta_3, \theta_4, \rho_1, \rho_2, \rho_3$ and ρ_4 as follows.

C^b = (A,beam,A) ←
W = (A)
w_1 = ([])
w_2 = ([block])
w_3 = ([brick])
w_4 = ([block,brick])
θ_1 = {A/[]}
θ_2 = {A/[block]}
θ_3 = {A/[brick]}
θ_4 = {A/[block,brick]}
ρ_1 = { }
ρ_2 = { }
ρ_3 = { }
ρ_4 = { }

We can select r = v189, then the new program P_b is

$P \cup \{C^C\} \cup \{C_1^D, C_2^D, C_3^D, C_4^D\}$,

where

C^C = (arch, (A,beam,A)) ← (v189, (A))
C_1^D = (v189, ([])) ←
C_2^D = (v189, ([block])) ←
C_3^D = (v189, ([brick])) ←
C_4^D = (v189, ([block,brick])) ←

It is clear that P_a is R-equivalent to P_b. The transformation from P_a to P_b corresponds to what is done by the intra-construction operator.

8.2 Truncation

The following example was used to illustrate the truncation operator.

```
Input:
  member(blue,[blue]).
  member(eye,[eye,nose,throat]).
Output:
  member(A,[A|B]).
```

We will explain this example by using the theory in this paper. Assume that we are given a program $P_a = P \cup \{C_1^A, C_2^A\}$, where C_1^A and C_2^A are the following facts.

C_1^A = (member, (blue,[blue])) ←
C_2^A = (member, (eye,[eye,nose,throat])) ←

Let R be the set of all predicates in P_a, then member $\in R$. We assume v002 $\in \mathcal{R} - R$. From C_1^A and C_2^A we know

C^r = member ←
C_1^a = (blue,[blue]) ←
C_2^a = (eye,[eye,nose,throat]) ←

As one of the least general generalizations of C_1^a and C_2^a is (A,[A|B]) ←, we can select

C^b = (A,[A|B]) ←
W = (A,B)
w_1 = (blue,[])
w_2 = (eye,[nose,throat])
θ_1 = {A/blue,B/[]}
θ_2 = {A/eye,B/[nose,throat]}
ρ_1 = { }
ρ_2 = { }

We can select r = v002, then the new program $P_b = P \cup \{C^C\} \cup \{C_1^D, C_2^D\}$, where

C^C = (member, (A,[A|B])) ← (v002, (A,B))
C_1^D = (v002, (blue,[])) ←
C_2^D = (v002, (eye,[nose,throat])) ←

It is clear that P_a is R-equivalent to P_b. Furthermore, when we add a hypothesis clause

C^H = (v002, (X,Y)) ← (v_{all}, (X,Y))

where v_{all} is one of the built-in predicates [19] which can be represented by infinite [20] clauses of the form:

C_j^{all} = (v_{all}, ($X_1, X_2, \ldots, X_j$)) ←

It is clear that from C^C, C^H and C_2^{all} we can deduce the following clause

C^T = (member, (A,[A|B])) ←

which is equivalent to the result obtained by the truncation operator.

[19] All the built-in predicates are assumed to be included in P.

[20] Note that infinite number of clauses can be included in a program in GLP theory.

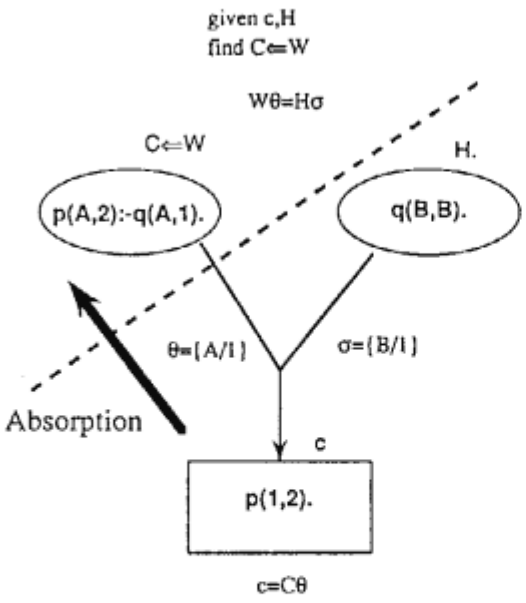


Figure 8: Problem for Absorption

8.3 Absorption

We will explain the following example by using the theory in this paper.

```
Input:
  less(A, s(A)).
  less(B, s(s(B))).
Output:
  less(X, s(Y)) :-  less(X, Y).
```

One of the clauses which the absorption operator generates is shown above. Assume that we are given a program $P_a = P \cup \{C_1^A, C_2^A\}$, where C_1^A and C_2^A are the following clauses.

C_1^A = (less, (A,s(A))) ←
C_2^A = (less, (B,s(s(B)))) ←

Let R be the set of all predicates in P_a, then less $\in R$. We assume v098 and v099 are included in $\mathcal{R}-R$. From C_1^A and C_2^A we know

C_1^r = less ←
C_2^r = less ←
C_1^a = (A,s(A)) ←
C_2^a = (B,s(s(B))) ←

we can select $C_1^b, C_2^b, W_1, W_2, w, \theta_1, \theta_2, \rho_1$ and ρ_2 as follows.

C_1^b = (X,Y) ←
C_2^b = (Z,s(W)) ←
W_1 = (X,Y)
W_2 = (Z,W)
w = (A,s(A))
θ_1 = {X/A,Y/s(A)}
θ_2 = {Z/A,W/s(A)}
ρ_1 = { }
ρ_2 = {A/B}

We can select r_1 = v098 and r_2 = v099. From these we get a new program $P_b = P \cup \{C_1^C, C_2^C\} \cup \{C_1^D, C_2^D\}$, where

C_1^C = (less, (X,Y)) ← (v098, (X,Y))
C_2^C = (less, (Z,s(W))) ← (v099, (Z,W))
C_1^D = (v098, (A,s(A))) ←
C_2^D = (v099, (A,s(A))) ←

P_a and P_b are R-equivalent. It is easy to obtain the following two hypothesis clauses.

C_{12}^H = (v098, (X,Y)) ← (v099, (X,Y))
C_{21}^H = (v099, (X,Y)) ← (v098, (X,Y))

When we further add to $P_b \cup \{C_{12}^H, C_{21}^H\}$ a hypothesis clause

C^H = (v098, (X,Y)) ← (less, (X,Y))

which is the inverse clause of C_1^C. It is clear that from C_2^C, C_{21}^H and C^H we can deduce the following clause.

C^T = (less, (X,s(Y))) ← (less, (X,Y))

which is equivalent to the result obtained by the absorption operator. While the absorption operator returns a clause which is equivalent to C^T, the procedure in this paper returns a set of clauses $P_c = \{C_1^C, C_2^C, C_1^D, C_2^D, C_{12}^H, C_{21}^H, C^H\}$ which can deduce C^T. We can modify P_c by changing some of the hypothesis clauses in P_c without retracting invented predicates. But the result of absorption is more difficult to modify. In this sense our framework is more flexible than CIGOL.

9 Examples for the Domain of Strings

Here we use examples of translation learning [2] by the domain independent inductive learning system LS/1.

9.1 Using Common Structure Procedure

Assume that we are given a program $P_a = P \cup \{C_1^A, C_2^A\}$, where C_1^A and C_2^A are the following facts.

C_1^A = (v-root, ([I am a little girl],
[WATASHI HA CHIISANA SHOUJO DA])) ←
C_2^A = (v-root, ([I am a boy],
[WATASHI HA SHOUNEN DA])) ←

Let R be the set of all predicates in P_a. Then v-root $\in R$. We assume v561 $\in \mathcal{R} - R$. From C_1^A and C_2^A we know

C^r = v-root ←
C_1^a = ([I am a little girl],
[WATASHI HA CHIISANA SHOUJO DA]) ←
C_2^a = ([I am a boy],
[WATASHI HA SHOUNEN DA]) ←

we can select $C^b, W, w_1, w_2, \theta_1, \theta_2, \rho_1$ and ρ_2 as follows [21].

C^b = ([I am a $1], [WATASHI HA $2 DA]) ←
W = ([$1], [$2])
w_1 = ([little girl], [CHIISANA SHOUJO])
w_2 = ([boy], [SHOUNEN])
θ_1 = {$1/[little girl],
$2/[CHIISANA SHOUJO]}

[21] In this case, symbol atoms which begin with dollar represent variables.

θ_2 = {$1/[boy], $2/[SHOUNEN]}
ρ_1 = { }
ρ_2 = { }

We can select r = v561. Then the new program is $P_b = P \cup \{C^C\} \cup \{C_1^D, C_2^D\}$, where

C^C = (v-root, ([I am a $1], [WATASHI HA $2 DA])) ← (v561, ([$1], [$2]))
C_1^D = (v561, ([little girl], [CHIISANA SHOUJO])) ←
C_2^D = (v561, ([boy], [SHOUNEN])) ←

It is clear that P_a is R-equivalent to P_b.

9.2 Using Common Component Procedure

Assume that we are given a program $P_a = P \cup \{C_1^A, C_2^A\}$, where C_1^A and C_2^A are the following facts.

C_1^A = (v248, ([WATASHI NO RINGO], [my apple])) ←
C_2^A = (v248, ([RINGO], [an apple])) ←

Let R be the set of all predicates in $P \cup \{C_1^A, C_2^A\}$. Then v248 $\in R$. We assume v198, v199 $\in \mathcal{R} - R$. From C_1^A and C_2^A we know C_1^a and C_2^a.

C_1^a = ([WATASHI NO RINGO], [my apple]) ←
C_2^a = ([RINGO], [an apple]) ←

We can select clauses, atoms and specializations as follows.

C_1^b = ([WATASHI NO $481], [my $482]) ←
C_2^b = ([$483], [an $484]) ←
W_1 = ([$481], [$482])
W_2 = ([$483], [$484])
w = ([RINGO], [apple])
θ_1 = {$481/[RINGO], $482/[apple]}
θ_2 = {$483/[RINGO], $484/[apple]}
ρ_1 = { }
ρ_2 = { }

We can select r_1 = v198 and r_2 = v199. From these we get a new program $P_b = P \cup \{C_1^C, C_2^C\} \cup \{C_1^D, C_2^D\}$, where

C_1^C = (v248, ([WATASHI NO $481], [my $482])) ← (v198, ([$481], [$482]))
C_2^C = (v248, ([$483], [an $484])) ← (v199, ([$483], [$484]))
C_1^D = (v198, ([RINGO], [apple])) ←
C_2^D = (v199, ([RINGO], [apple])) ←

It is clear that P_a is transformed to P_b R-equivalently. It is easy to add the following hypothesis clauses to P_b.

C_{12}^H = (v198, ([$X], [$Y])) ← (v199, ([$X], [$Y]))
C_{21}^H = (v199, ([$X], [$Y])) ← (v198, ([$X], [$Y]))
C_1^H = (v198, ([$481], [$482])) ← (v248, ([WATASHI NO $481], [my $482]))
C_2^H = (v199, ([$483], [$484])) ← (v248, ([$483], [an $484]))

Then P_b becomes P_c.

$P_c = P \cup \{C_1^C, C_2^C, C_1^D, C_2^D, C_{12}^H, C_{21}^H, C_1^H, C_2^H\}$

From P_c we can deduce the following clauses

C_{12}^Z = (v248, ([WATASHI NO $X], [my $Y])) ← (v248, ([$X], [an $Y]))
C_{21}^Z = (v248, ([$X], [an $Y])) ← (v248, ([WATASHI NO $X], [my $Y]))

which might be obtained by an absorption operator for the string domain.

10 Conclusion

Predicate invention is the key procedure of inductive learning. The theory of predicate invention proposed in this paper has the following significant features.

- There are two methods of predicate invention. One is Common Structure Procedure (CSP), where the common part of given clauses forms the main structure and the different parts are found to be new objects. The other is Common Component Procedure (CCP), where the common part is extracted as a component and different parts are found to be new contexts. CCP is the "dual" procedure of CSP for inventing predicates. Muggleton and Buntine proposed only one operator (the intra-construction operator) for predicate invention, which corresponds to CSP.

- Predicate invention in this paper is separated from hypothesis generation. When a new predicate is made, the semantic meaning [22] of the given program is not altered but it may later be effectively used in forming hypotheses.

- By adding a simple hypothesis formation procedure, we can generate hypothesis clauses, which increases the semantic meaning of the given programs. Simple hypothesis formation associated with predicate invention can give plausible hypotheses which may be generated by the truncation and absorption operators.
  CSP + SHF = truncation
  CCP + SHF = absorption
  where SHF means Simple Hypothesis Formation.

- The theory in this paper can be applied to homogeneous generalized logic programs on safe specialization systems. Logic programs with terms (Prolog) and logic programs with strings are safe specialization systems and all clauses on these specialization systems are homogeneous. Constraint logic programs form safe specialization systems and all clauses used in constraint logic programs are homogeneous. By constructing appropriate safe specialization systems, many knowledge representations can be regarded as homogeneous generalized logic programs on them. Therefore the theory in this paper can be applied to diverse knowledge representations and data structures.

[22] Semantic meaning of a program P is $REP(P)$.

References

[1] Akama,K. and Ichikawa,A. : A Basic Model for Learning Systems, Proc. of 6th IJCAI, (1979)

[2] Akama,K. : Learning of Translation by the Inductive Learning System LS/1 (in Japanese), *J. Japan Soc. Artif. Intell.*,Vol.2 No.3, pp.341-349 (1987)

[3] Akama,K. : Construction of Learning Systems based on GLP theory 1 (in Japanese), *Preprints Work. Gr. for Knowledge Engineering and Artificial Intelligence, IPSJ*,64-3, pp.21-30 (1989)

[4] Akama,K. : Declarative Semantics of Logic Programs on Parameterized Representation Systems, *Hokkaido University Information Engineering Technical Report*, HIER-LI-9001 (1990)

[5] Akama,K. : Generalized Logic Programs on Specialization Systems and SLDA Resolution, *Hokkaido University Information Engineering Technical Report*, HIER-LI-9002 (1990)

[6] Akama,K. : Predicate Invention based on the Theory of Generalized Logic Programs, *Hokkaido University Information Engineering Technical Report*, HIER-LI-9101 (1991)

[7] Dietterich, T.G and Michalsky,R.S.: Inductive Learning of Structural Descriptions, *Artificial Intelligence*,16, pp.257-294 (1981)

[8] Jaffar,J. and Lassez,J.L.: Constraint Logic Programming, Technical Report, Department of Computer Science, Monash University, June (1986)

[9] Laird,P. : EBG and Term-Rewriting Systems, Proc. of ALT '90, pp.425-440 (1990)

[10] Liu,S. and Hagiya,M. : Model Inference of Constrained Recursive Figures, Proc. of ALT '90, pp.355-367 (1990)

[11] Muggleton,S.: Duce, an oracle based approach to constructive induction, IJCAI 10, pp.287-292 (1987)

[12] Muggleton,S. and Buntine,W. : Machine invention of First-order Predicates by inverting resolution, Proc. 5th International conference on Machine learning pp.339-351 (1988)

[13] Sato,S. and Nagao,M. : A Learning Method for Translation Grammar based on Grammatical Inference (in Japanese), Research Report of Information Processing Society of Japan, SIGAI, 46-11 (1986)

[14] Shapiro,E : Logic Program with Uncertainties : A tool for Implementing Rule-Based Systems, Proc. 8th IJCAI, pp.529-532 (1983)

A theoretical framework for predicate invention

Stephen Muggleton
The Turing Institute,
36 North Hanover Street,
Glasgow G1 2AD,
UK.

April 24, 1992

Abstract

A number of authors within the Inductive Logic Programming literature have investigated the problem of introducing novel terms into the learner's vocabulary. This process is known as *predicate invention*. There is a growing need for a theoretical framework for predicate invention. We describe a first attempt at such a framework in an attempt to answer when predicate invention is necessary and from which universe these predicates are chosen. The framework uses the notion of a lattice of predicate utility. Some results of an initial implementation are given.

Inverting Implication

Stephen Muggleton
The Turing Institute,
36 North Hanover Street,
Glasgow G1 2AD,
UK.

April 27, 1992

Abstract

All generalisations within logic involve inverting implication. Yet, ever since Plotkin's work in the early 1970's methods of generalising first-order clauses have involved inverting the clausal subsumption relationship. However, even Plotkin realised that this approach was incomplete. Since inversion of subsumption is central to many Inductive Logic Programming approaches, this form of incompleteness has been propagated to techniques such as Inverse Resolution and Relative Least General Generalisation. A more complete approach to inverting implication has been attempted with some success recently by Lapointe and Matwin. In the present paper the author derives general solutions to this problem from first principles. It is shown that clausal subsumption is only incomplete for self-recursive clauses. Avoiding this incompleteness involves algorithms which find "nth roots" of clauses. Completeness and correctness results are proved for a non-deterministic algorithms which constructs nth roots of clauses. It is shown how this algorithm can be used to invert implication in the presence of background knowledge. In conclusion the relationship between these results and Hoare's logical definition of programming from specifications is discussed.

1 Introduction

Plotkin [19] was the first to show that θ-subsumption and implication between clauses are not equivalent. The difference between the two is important since almost all inductive algorithms which generalise first-order clauses invert θ-subsumption rather than implication. This inevitably leads to a form of incompleteness in these algorithms. In this paper methods of constructing the inverse implicants of clauses are explored. In section 7 it is shown how methods developed in earlier sections can be extended to the problem of inverting implication in the presence of background knowledge. First the difference between Plotkin's θ-subsumption and implication

between clauses will be reviewed. The reader is referred to Appendix A for the usual definitions in Logic Programming and Inductive Logic Programming (ILP).

Clause C θ-subsumes clause D whenever there exists a substitution θ such that $C\theta \subseteq D$. Clause C implies clause D, or $C \rightarrow D$, whenever every model of C is a model of D. Whenever clause C θ-subsumes clause D it also implies D. However the converse does not hold. For instance Plotkin shows that with clauses

$$\begin{aligned} C &= p(f(X)) \leftarrow p(X) \\ D &= p(f(f(X))) \leftarrow p(X) \end{aligned}$$

C implies D, since D is simply C self-resolved. However C does not θ-subsume D. In discussing this problem Niblett [18] proves various general results. For instance he shows that implication between Horn clauses is decidable and also that there is not always a unique least generalisation under implication of an arbitrary pair of clauses. For instance, the clause D above and the clause $E = p(f(f(f(X)))) \leftarrow p(X)$ have both C and the clause $p(f(X)) \leftarrow p(Y)$ as least generalisations.

Gottlob [8] also proves a number of properties concerning impliction between clauses. Notably let C^+, C^- be the positive and negative literals of C and D^+, D^- be the same for D. Now if $C \rightarrow D$ then C^+ θ-subsumes D^+ and C^- θ-subsumes D^-.

2 Sub-unification

The problem of inverting implication is discussed in a recent paper by Lapointe and Matwin [11]. They note that inverse resolution [15, 14, 22, 24] is incapable of reversing SLD derivations in which the hypothesised clause is used more than once. In fact Plotkin [19] showed that the same problem appears in the use of relative least general generalisation of clauses. Lapointe and Matwin go on to describe sub-unification, a process of matching sub-terms. They demonstrate that sub-unification is able to construct recursive clauses from fewer examples than would be required by ILP systems such as Golem [16] and FOIL [20]. For instance, given the atoms $append([], X, X)$ and $append([a, b, Y], [1, 2], [a, b, Y, 1, 2])$ sub-unification can be used to construct the recursive clause

$$append([U|V], W, [X|Y]) \leftarrow append(V, W, Y)$$

Unlike the approach taken originally with inverse resolution [15], Lapointe and Matwin do not derive sub-unification from resolution. Instead sub-unification is based on a definition of most general sub-unifiers. Although the operations described by Lapointe and Matwin are shown to work on a number of examples it is not clear how general the mechanism is.

In this paper a general approach to inverting implication is developed. The approach taken involves a new form of inverting resolution which is derived from first principles.

3 Implication and resolution

In this section the relationship between resolution and implication between clauses is investigated. Below a definition equivalent to Robinson's [21] resolution closure is given. The function $\mathcal{L}$ below contains only the linear derivations of Robinson's function $\mathcal{R}$ (see Appendix A.3). However, the closure is equivalent up to renaming of variables given that linear derivation (as opposed to input derivation) is known to be complete.

Definition 1 (Resolution closure) *Let T be a set of clauses. The function $\mathcal{L}$ is recursively defined as*

$$\begin{aligned} \mathcal{L}^1(T) &= T \\ \mathcal{L}^n(T) &= \{C : C_1 \in \mathcal{L}^{n-1}(T), C_2 \in T, C \text{ is the resolvent of } C_1 \text{ and } C_2\} \end{aligned}$$

the resolution closure $\mathcal{L}^*(T)$ is $\mathcal{L}^1(T) \cup \mathcal{L}^2(T) \cup \ldots$

Lee [12] first proved the subsumption theorem, a reproof of which can be found in Bain and Muggleton [1]. The theorem can be stated as follows.

Theorem 2 (Subsumption theorem) *Let T be a set of clauses and C be a non-tautological clause. $T \models C$ if and only if there exists D in $\mathcal{L}^*(T)$ and substitution θ such that $D\theta \subseteq C$.*

In order to apply this to the case of implication between clauses Theorem 2 can be applied to the special case in which T is a single clause.

Corollary 3 (Implication between clauses using resolution) *Let C be an arbitrary clause and D be a non-tautological clause. $C \models D$, or $C \rightarrow D$, if and only if there exists a clause E in $\mathcal{L}^*(\{C\})$ and substitution θ such that $E\theta \subseteq D$.*
Proof. *Follows directly as a special case of theorem 2.*

Restating corollary 3, $C \rightarrow D$ whenever one of the following conditions holds

1. D is a tautology.
2. C θ-subsumes D.
3. E θ-subsumes D where E is constructed by repeatedly self-resolving C.

The first two conditions are somewhat trivial. The third condition demonstrates the significance of self-recursive clauses in this problem. It is clearly no coincidence that Plotkin's example (Section 1) and the clauses investigated by Lapointe and Matwin (Section 2) are self-recursive.

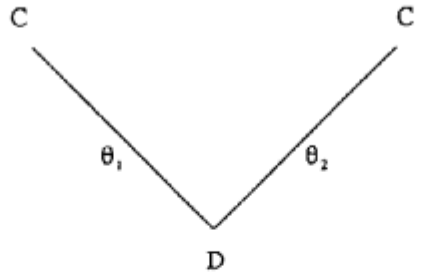


Figure 1: Squaring a clause

4 Nth powers and nth roots of clauses

The set of clauses constructed by self-recursing C, $\mathcal{L}^*(\{C\})$, is partitioned into levels by the function $\mathcal{L}$. By viewing resolution as a product operation Muggleton and Buntine [15] (see '$\cdot$' operator in A.3) stated the problem of finding the inverse resolvent of a pair of clauses as that of finding the set of quotients of two clauses. Following the same analogy the set $C^2 = \mathcal{L}^2(\{C\})$ might be called the squares of the clause C and $C^3 = \mathcal{L}^3(\{C\})$ the cubes of C. The following definition captures this idea.

Definition 4 (nth powers of a clause) *Let C and D be clauses. For $n \geq 1$, D is an nth power of C if and only if D is an alphabetic variant of a clause in $\mathcal{L}^n(\{C\})$.*

Taking the analogy a bit further one might also talk about the nth roots of a clause.

Definition 5 (nth roots of a clause) *Let C and D be clauses. D is an nth root of C if and only if C is an nth power of D.*

Corollary 3 can now be restated in terms of nth roots of a clause.

Corollary 6 (Implication between clauses in terms of nth roots) *Let C be an arbitrary clause and D be a non-tautological clause. $C \rightarrow D$ if and only if for some positive integer n, C is an nth root of a clause E which θ-subsumes D.*

It is fairly straightforward to enumerate the set of clauses which θ-subsume a given clause. Therefore the problem of finding the set of clauses which imply a given clause C reduces to that of enumerating the set of nth roots of clauses which θ-subsume C. The special case of clauses which immediately θ-subsume C occurs with $n = 1$.

5 Constructing the square roots of a clause

Before attempting the harder problem of constructing arbitrary nth roots of clauses let us consider the simpler problem of constructing the square roots of a clause. Figure 1 shows the self resolution of the clause C to give D, where $D \in C^2$.

Assume that this resolution involves the complementary pair $\langle l\theta_1, \overline{l'}\theta_2 \rangle$ where l is a positive literal in C and $\overline{l'}$ is a negative literal in C and $\theta_1\theta_2$ is the most general unifier (mgu) of l and l'. From the definition of a complementary pair (Appendix A.1)

$$l\theta_1 = l'\theta_2$$

where the domains of θ_1 and θ_2 are subsets of $vars(l)$ and $vars(l')$ respectively. Clause C can be written as

$$l \leftarrow l' \wedge B \tag{1}$$

where B is a conjunction of literals. If C is a definite clause and l is an atom then B will be a conjunction of atoms. D, any self-resolvent of C, has the general form

$$l\theta_2 \leftarrow l'\theta_1 \wedge B\theta_1 \wedge B\theta_2 \tag{2}$$

The problem now is how to construct C (clause 1) from D (clause 2). To do so l, l' and B need to be reconstructed. A simple non-deterministic and incomplete first-cut approach to this problem would seem to be as follows.

Algorithm 1 (a simple, flawed square root algorithm)

1. *Choose from D a pair of literals $\langle l\theta_2, \overline{l'\theta_1} \rangle$ with the same predicate symbol.*
2. *Partition the clause $D - \{l\theta_2, \overline{l'\theta_1}\}$ into two equal cardinality conjuncts $B\theta_1$ and $B\theta_2$ which are both instances of a conjunct B.*
3. *Construct θ_1 and θ_2 by matching B to $B\theta_1$ and $B\theta_2$.*
4. *Invert the substitution θ_1 on $l'\theta_1$ and θ_2 on $l\theta_2$ to get l' and l respectively.*
5. *Return $l \leftarrow l' \wedge B$*

The following example, which uses Prolog-like notation, demonstrates Algorithm 1 at work.

Example 7 (Trace of Algorithm 1)
Let clause C be

$$lt(F, G) \leftarrow succ(F, H),\ lt(H, G)$$

and D be

$$lt(I, J) \leftarrow succ(I, K),\ succ(K, L),\ lt(L, J)$$

The steps in Algorithm 1 are followed below to reconstruct C from D.

1. *Let $l\theta_2 = lt(I, J)$ and $l'\theta_1 = lt(L, J)$.*
2. *Let $B\theta_2 = succ(I, K)$ and $B\theta_1 = succ(K, L)$, which are both instances of $B = succ(M, N)$.*
3. *$\theta_1 = \{M/K, N/L\}$ and $\theta_2 = \{M/I, N/K\}$.*

4. $l = lt(M, J)$ *and* $l' = lt(N, J)$.

5. *Return* $lt(M, J) \leftarrow lt(N, J), succ(M, N)$

which is an alphabetic variant of C.

Algorithm 1 has a number of shortcomings. Firstly, it is non-deterministic. This could be overcome by constructing all possible solutions and returning those which self-resolve to give alphabetic variants of C. Secondly, Example 7 demonstrates that the substitutions θ_1 and θ_2 constructed in step 3 of Algorithm 1 can be incomplete. In Example 7 neither θ_1 nor θ_2 contain a substitution for the variable J. Plotkin's clauses in Section 1 provide an extreme example of the incomplete construction of θ_1 and θ_2. In Plotkin's example B is empty and therefore step 3 will fail to extract θ_1 and θ_2. Thirdly, the inversion of the substitutions in step 4 is not straightforward. If the use of inverse substitutions described in [15] (defined also in Appendix A.2) is followed then there can be a multiplicity of possible inverse substitutions of a particular substitution θ applied to a given literal l. Many of these problems can be avoided by first flattening the clause D, constructing its square roots and then unflattening the results.

5.1 Flattening clauses

Rouveirol and Puget [23, 22] describe operations called *flattening* and *unflattening* to simplify inverse resolution. This form of operation is well-known within the literature of integrating logic programming and functional programming (see for instance [7, 4]). Rouveirol and Puget's flattening operation transforms clauses with function symbols into clauses in a function-free form. Unflattening a clause transforms it back to its original form. The approach taken in this paper to flattening clauses differs from Rouveirol and Puget in that only equality literals are introduced rather than introducing new predicates. For the purposes of finding the square roots of a clause, flattened clauses will be used with a particular goal in mind. The goal is to ensure that the *mgu* involved in self-resolving a clause is a special kind of substitution known as a *renaming*. Since renamings are easy to invert they help solving some of the problems with Algorithm 1.

5.2 Renaming

Lloyd [13] defines a renaming substitution, or a renaming for short, as follows (see Appendix A.1 for the definition of vars(F)).

Definition 8 (Renamings) *Let* F *be a well-formed formula and* $\theta = \{u_1/v_1, .., u_n/v_n\}$ *be a substitution.* θ *is a renaming of* F *if and only if* $u_1, .., u_n$ *are all distinct variables,* $v_1, .., v_n$ *are all distinct variables and* $(vars(F) - \{u_1, .., u_n\}) \cap \{v_1, .., v_n\} = \emptyset$.

Renamings are easy to invert because they are one-to-one mappings. The inverse of a renaming is defined as follows.

Definition 9 (Inverse renaming) *Let $\theta = \{u_1/v_1, .., u_n/v_n\}$ be a renaming of the formula F. θ^r, the inverse of θ for F, is the substitution $\{v_1/u_1, .., v_n/u_n\}$.*

The following theorems about renamings can now be shown.

Lemma 10 (Renaming composed with its inverse is identity function) *Let θ be a renaming of the well-formed formula F. $F\theta\theta^r = F$.*
Proof. *Each variable u in the domain of θ is mapped to a distinct variable v by θ. v is then mapped back to u using θ^r.*

Lemma 11 (Composition of renamings is a renaming) *Let θ be a renaming of the formula F and τ be a renaming of $F\theta$. The substitution $\sigma\tau$ is a renaming of F.*
Proof. *Let u be mapped to v in θ. If v is in the domain of τ and v is mapped to w in τ then u is mapped uniquely to w in $\sigma\tau$. Otherwise u is mapped uniquely to v in $\sigma\tau$.*

5.3 Flattening using equalities

As stated earlier the goal is to use flattening to ensure that the *mgu* involved in self-resolving a clause C is a renaming of C. This can be done by flattening the clause so that the only terms in the two recursing literals are sets of distinct variables.

Example 12 (Flattening, squaring and unflattening a clause)
Let clause C be

$$member(G, [H|I]) \leftarrow member(G, I)$$

C is flattened to to

$$member(G, J) \leftarrow member(G, I),\ J = [H|I]$$

in which the only terms in the two recursing literals are sets of distinct variables. Self-resolve C involves resolving it with $C\sigma$ where σ is a renaming of all the variables in C. Thus $C\sigma$ might be

$$member(G', J') \leftarrow member(G', I'),\ J' = [H'|I']$$

The head of C can be resolved away with the member atom in the body of $C\sigma$. The mgu involved is the renaming $\theta = \{G'/G, I'/J\}$ of C and the resolvent is

$$member(G, J') \leftarrow member(G, I),\ J' = [H'|J],\ J = [H|I]$$

Unflattening this clause gives

$$member(G, [H', H|I]) \leftarrow member(G, I)$$

which is the square of the clause C.

Flattening and unflattening of clauses need now to be formally defined. First the function unflat is defined as follows.

Definition 13 (Unflattening) *Let C be the clause $D \vee \overline{E}$ where D contains no equality literals and E is the conjunction $s_1 = t_1 \wedge s_2 = t_2 \wedge .. \wedge s_n = t_n$. $unflat(C) = D\epsilon_1\epsilon_2..\epsilon_n$ where ϵ_i is the mgu of s_i and t_i for $1 \leq i \leq n$.*

Unflattening is equivalent to resolving away all equality literals in the body of a clause using the single axiom equality theory

$$X = X$$

The set of flattened clauses is defined as follows.

Definition 14 (Flattening) *Let C and D be clauses. $D \in flat(C)$ if and only if C is an alphabetic variant of $unflat(D)$.*

5.4 Canonical flattening

Next a canonical flattening will be defined to capture the method of flattening applied in Example 12.

Definition 15 (Canonical flattening) *Let C and $D = F \vee \overline{E}$ be clauses in which F contains no equality literals and E is a conjunction of atoms. D is a canonical flattening of C, or $D \in cf(C)$, if and only if $D \in flat(C)$ and every literal in F has the form $p(v_1, .., v_n)$ or $\underline{p(v_1, .., v_n)}$ in which $v_1, .., v_n$ are distinct variables and every atom in E has the form $x = y$ or $x = f(y_1, .., y_m)$.*

Since it is intended to use canonical flattening to improve Algorithm 1 it is necessary to show the following.

Theorem 16 (Mgu of squaring a canonical flattening is a renaming) *Let C be a clause and D be a canonical flattening of C. If D is self-resolved to give E then the mgu involved in the resolution is a renaming of D.*

Proof. *Let $D = l \leftarrow l' \wedge B$, let $D\sigma$ be D standardised apart using the renaming σ of D and let $\langle l, \overline{l'\sigma} \rangle$ be the pair of literals involved in the resolution of D and $D\sigma$. Letting $l = p(u_1, .., u_n)$ and $l'\sigma = p(v_1, .., v_n)$ the mgu involved in the resolution is $\theta = \{v_1/u_1, .., v_n/u_n\}$. The variables $u_1, .., u_n$ and $v_1, .., v_n$ are all distinct since both D and $D\sigma$ are canonical flattenings of C (see Definition 15). The sets $\{u_1, .., u_n\}$ and $\{v_1, .., v_n\}$ are disjoint and none of the variables $\{v_1, .., v_n\}$ appear in D since D and $D\sigma$ have been standardised apart. Therefore, by Definition 8, θ is a renaming of D.*

Next it is necessary to be assured that flattening a pair of clauses, resolving them and then unflattening the resolvent gives the same result as resolving the original clauses.

Lemma 17 (Unflattening distributes over resolution) *Let C_1 and C_2 be clauses and D_1 and D_2 be flattenings of C_1 and C_2 respectively. The clause F is the resolvent of D_1 and D_2 only if $unflat(F)$ is the resolvent of C_1 and C_2.*

Proof. *Let $D_1 = l \leftarrow B_1 \wedge E_1$ and $D_2 = l_2 \leftarrow l' \wedge B_2 \wedge E_2$ where E_1 and E_2 are*

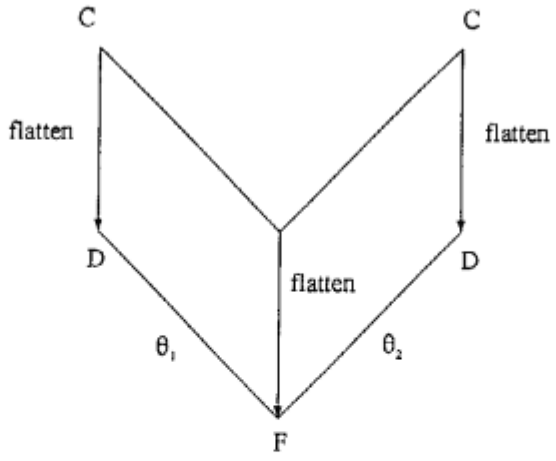


Figure 2: Flattening and squaring is equivalent to squaring and flattening

the set of all equality atoms in D_1 and D_2. Let $C_1 = unflat(D_1) = (l \leftarrow B_1)\epsilon_1$ and $C_2 = unflat(D_2) = (l_2 \leftarrow l' \wedge B_2)\epsilon_2$ where ϵ_1 and ϵ_2 are the substitutions produced by resolving away E_1 and E_2 respectively. Let $\epsilon = \epsilon_1\epsilon_2$. Let D_1 and D_2 resolve on the pair of literals $\langle l, \overline{l'}\rangle$ to give F. Then $unflat(F)$ is equivalent to

$$unflat(l_2 \leftarrow B_1 \wedge B_2 \wedge E_1 \wedge E_2 \wedge (l = l')) \tag{3}$$

since unflat will unify and remove l and l'. In the same way resolving C_1 and C_2 using the pair of literals $\langle l\epsilon, \overline{l'\epsilon}\rangle$ is equivalent to

$$unflat((l_2 \leftarrow B_1 \wedge B_2 \wedge (l = l'))\epsilon) \tag{4}$$

Formula 4 can be derived from formula 3 by resolving away $E_1 \wedge E_2$. This completes the proof.

As a corollary flattening, squaring and unflattening a clause must be equivalent to squaring the original clause.

Corollary 18 (Flattening, squaring and unflattening) *Let C be a clause and D be a flattening of C. The clause F is a square of D only if $unflat(F)$ is a square of C.*

Proof. *Let the renamings σ and τ be used to standardise apart C and D respectively. The corollary is now simply a restatement of Lemma 17 with $C_1 = C$, $C_2 = C\sigma$, $D_1 = D$ and $D_2 = D\tau$.*

This corollary applies to squaring a clause whereas the intention is to canonically flatten a clause, extract its square root and unflatten the result. Figure 2 illustrates the following theorem.

Theorem 19 (Canonical flattening and squaring) *Let C be a clause and D be a canonical flattening of C. The clause F is a square of D only if F is a canonical flattening of a square of C.*

Proof. *Given Corollary 18 it is only necessary to show that squaring a canonical flattening produces a canonical flattening. Let* $D = l \leftarrow l' \wedge B \wedge E$ *where* E *is conjunction of all equality atoms. Let* σ *standardises* $D\sigma$ *apart from* D. *Let* D *and* $D\sigma$ *resolve with complementary pair* $\langle l\theta_1, \overline{l'\sigma}\theta_2\rangle$ *to give* $F =$

$$l\sigma\theta_2 \leftarrow l'\theta_1 \wedge (B \wedge E)\theta_1 \wedge (B \wedge E)\sigma\theta_2$$

According to Lemma 11, since σ *is a renaming of* D *and* θ_2 *is a renaming of* $D\sigma$ *their composition* $\sigma\theta_2$ *is a renaming of* D. *Every subcomponent of* F *is a renaming of a canonical flattening. Therefore* F *is a canonical flattening.*

5.5 An improved square root algorithm

Theorem 19 suggests the following nondeterministic algorithm for extracting the square roots of a clause.

Algorithm 2 (Square root algorithm)

1. *Canonically flatten the input clause* G *to give clause* F.
2. *Choose from* F *a pair of literals* $\langle l\theta_2, \overline{l'\theta_1}\rangle$ *with the same predicate symbol.*
3. *Partition the clause* $F - \{l\theta_2, \overline{l'\theta_1}\}$ *into two equal cardinality conjuncts* $B\theta_1$ *and* $B\theta_2$ *from which* B *is constructed by taking the least generalisation of corresponding pairs of literals which are alphabetic variants.*
4. *Construct the renamings* θ_1 *and* θ_2 *of* D *by matching* B *to* $B\theta_1$ *and* $B\theta_2$.
5. *Apply* θ_1^r *to* $l'\theta_1$ *and* θ_2^r *to* $l\theta_2$ *to get* l' *and* l *respectively (see Lemma 10).*
6. *Return* $C = unflat(l \leftarrow l' \wedge B)$ *if* G *is the square of* C.

This can now be applied to Plotkin's example from Section 1.

Example 20 (Trace of Algorithm 2)
Let clause G *be* $p(f(f(X))) \leftarrow p(X)$.
The steps in Algorithm 2 are followed below.

1. F *is* $p(Y) \leftarrow p(X),\ Y = f(Z),\ Z = f(X)$.
2. *Let* $l\theta_2 = p(Y)$ *and* $l'\theta_1 = p(X)$.
3. *Let* $B\theta_2$ *be* $(Y = f(Z))$. $B\theta_1$ *be* $(Z = f(X))$ *and* B *be* $(U = f(V))$.
4. θ_1 *is the renaming* $\{U/Z, V/X\}$ *of* D *and* θ_2 *is the renaming* $\{U/Y, V/Z\}$ *of* D.
5. θ_1^r *is* $\{Z/U, X/V\}$, θ_2^r *is* $\{Y/U, Z/V\}$, $l = l\theta_2\theta_2^r = p(U)$ *and* $l' = l'\theta_1\theta_1^r = p(V)$.
6. *Return* $C = unflat(p(U) \leftarrow p(V),\ U = f(V))$ *which is* $p(f(V)) \leftarrow p(V)$. G *is the square of* C.

The following theorem shows that Algorithm 2 is complete and correct in a non-deterministic sense.

Theorem 21 (Completeness and correctness of Algorithm 2) *Let C be a clause and G be a square of C. When Algorithm 2 is presented with G there is a set of choices made in steps 1, 2 and 3 which will construct an alphabetic variant of C.*

Proof. *Algorithm 2 is correct since step 6 guarantees that any solution returned will be a square root of C. Therefore it is necessary to show it is complete in the sense that an alphabetic variant of every square root of G can be constructed given appropriate choices for the non-deterministic steps 1, 2 and 3. This is guaranteed by Theorem 19 since there is a canonical flattening F of G which has the form*

$$l\sigma\theta_2 \leftarrow l'\theta_1 \wedge (B \wedge E)\theta_1 \wedge (B \wedge E)\sigma\theta_2$$

if $D = l \leftarrow l' \wedge B \wedge E$ and $C = (l \leftarrow l' \wedge B)\epsilon$ (see proof of Theorem 19).

5.6 Problems with Algorithm 2

Despite Theorem 21, applying Algorithm 2 is not without problems. The problems are to do with generating flattened clauses in step 1. Consider again the canonically flattened clause $D =$

$$l \leftarrow l' \wedge B \wedge E$$

and its square $F =$

$$l\theta_2 \leftarrow l'\theta_1 \wedge B\theta_1 \wedge B\theta_2 \wedge E\theta_1 \wedge E\theta_2$$

Certain terms and literals will not appear in $unflat(F)$ under the following conditions.

- Literals m and m' appear in $B \wedge E$ for which $m\theta_1 = m'\theta_2$. Only one instance will therefore appear in F.
- The equality $v = t$ appears in $E\theta_1 \wedge E\theta_2$ and v does not appear in any other literal in D. The term t will therefore not appear in $unflat(F)$.
- The equalities $v = s$ and $v = t$ appear in $E\theta_1 \wedge E\theta_2$ and $s \neq t$. Only the term formed by unifying s and t appears in $unflat(F)$.

Restrictions could be devised for the clausal language to which Algorithm 2 can be straightforwardly applied. However this approach will not be followed up in this paper.

Original variables	u_1	v_1	..	u_n	v_n
θ_2	x_1	w_1	..	x_n	w_n
θ_1	w_1	y_1	..	w_n	y_n

Figure 3: Initialisation of substitution table

5.7 Tabulated substitutions

Steps 3 and 4 of Algorithm 2 cannot be wholly separated. From a programming point of view it makes sense to build up the substitutions θ_1 and θ_2 at the same time as matching literals in step 3. Only literals which lead to θ_1 and θ_2 being renamings should be matched. A simple way to do this is to build up a three row table of variables. The first row represents variables in the flattened version of C. The second and third rows represent the unique mappings of these variables in θ_2 and θ_1. When initialising this table it is possible to take advantage of a constraint related to the unification of the recursing literals l and l'. Suppose that $l = p(u_1, .., u_n)$ and $l' = p(v_1, .., v_n)$. These must unify to give a literal $p(w_1, .., w_n)$ where $l\theta_2 = p(x_1, .., x_n)$ and $l'\theta_1 = p(y_1, .., y_n)$. This constraint can be represented in the initialised substitution table shown in Figure 3.

The following example demonstrates the use of such a substitution table in Algorithm 2 for the predicate *split* which breaks a list into two approximately equal lengthed sublists.

Example 22 (Use of substitution table)
Let G be $split([H, I|J], [H|K], [I|L]) \leftarrow split(J, K, L)$. The steps in Algorithm 2 are followed below.

1. *F is $split(M, N, O) \leftarrow split(J, K, L), M = [H|P], P = [I|J], N = [H|K], O = [I|L]$.*

2. *Let $l\theta_2 = split(M, N, O)$ and $l'\theta_1 = split(J, K, L)$.*

3. *The substitution table is initialised to*

Original variables	Q	R	S	T	U	V
θ_2	M	w_1	N	w_2	O	w_3
θ_1	w_1	J	w_2	K	w_3	L

Let $B\theta_2$ be $(M = [H|P], N = [H|K])$, $B\theta_1$ be $(P = [I|J], O = [I|L])$ and B be $(Q = [W|R], S = [W|V])$. The final substitution table is

Original variables	Q	R	S	T	V	W
θ_2	M	P	N	O	K	H
θ_1	P	J	O	K	L	I

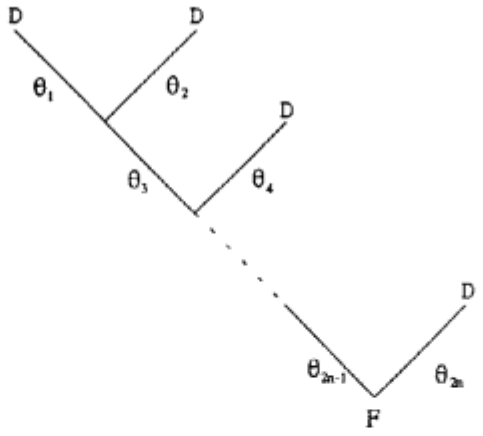


Figure 4: Self-resolving a clause n times

(Note that that the constraint described in Figure 3 led to the merging of columns of original variables T and U).

4. *θ_1 is the renaming $\{P/Q, J/R, O/S, K/T, L/V, I/W\}$ and $\theta_2$is the renaming $\{M/Q, P/R, N/S, O/T, K/V, H/W\}$.*

5. *θ_1^r is the renaming $\{Q/P, R/J, S/O, T/K, V/L, W/I\}$, $\theta_2^r$is the renaming $\{Q/M, R/P, S/N, T/O, V/K, W/H\}$, $l = l\theta_2\theta_2^r = split(Q, S, T)$ and $l' = l'\theta_1\theta_1^r = split(R, T, V)$.*

6. *Return $C = unflat(split(Q, S, T) \leftarrow split(R, T, V), Q = [W|R], S = [W|V])$ which is $split([W|R], [W|V], T) \leftarrow split(R, T, V)$. G is the square of C.*

6 Constructing the nth roots of a clause

In this section the construction of nth roots of clauses is investigated. Assume that D is a canonical flattening of the clause C . Figure 4 shows D resolved n times against itself to give the clause F, where $F \in D^n$. Suppose that this self-resolution always involves instances of the literals l and $\overline{l'}$ from D. This is a simplifying assumption since D need not always self-resolve using instances of the same pair of literals. If clause D is

$$l \leftarrow l' \wedge B$$

then clause F is

$$\begin{aligned} l\theta_{2n} \leftarrow l'\theta_1\theta_3..\theta_{2n-1} \wedge\ & B\theta_1\theta_3..\theta_{2n-1} \wedge \\ & B\theta_2\theta_3\theta_5..\theta_{2n-1} \wedge \\ & \ldots \wedge \\ & B\theta_{2n-2}\theta_{2n-1} \wedge \\ & B\theta_{2n} \end{aligned}$$

Note that F contains n instances of B. One of these instances has the same substitution as the instance of l (θ_{2n}). Another instance of B has the same substitution as the instance of l' ($\theta_1\theta_3..\theta_{2n-1}$). In the above it was assumed that self-resolution always involved instances of the literals l and $\overline{l'}$ from D. Somewhat surprisingly, the phenomenon of corresponding substitutions occurs for all clauses in D^n.

Theorem 23 *Let D be a clause, n be an integer greater than 1 and F be a clause in D^n. F has the form $l\gamma_1 \vee \overline{l'\gamma_2} \vee B\gamma_1 \vee B\gamma_2 \vee .. \vee B\gamma_n$ where l and l' are atoms in D with the same predicate symbol and $B = D - \{l, \overline{l'}\}$.*

Proof. *The proof is by mathematical induction on k. The base case, $n = 2$, is true since if D self-recurses with complementary pair $(l\theta_1\overline{l'}\theta_2)$ then F is $l\theta_2 \leftarrow l'\theta_1 \vee B\theta_1 \vee B\theta_2$. Assume the theorem is true for all integers from 2 to k and prove it follows for $k+1$. Let F in D^k be $l\gamma_1 \vee \overline{l'\gamma_2} \vee B\gamma_1 \vee B\gamma_2 \vee .. \vee B\gamma_k$ and D be $l \vee \overline{l'} \vee B$. Assume F and D resolve with complementary pair $(l\gamma_1\alpha, \overline{l'}\beta)$ and mgu $\alpha\beta$ where the domains of α and β are subsets of $vars(l\gamma_1)$ and $vars(l')$ respectively. The resolvent of F and D in D^{k+1} is $l\beta \vee \overline{l'\gamma_2\alpha} \vee B\gamma_1\alpha \vee B\gamma_2\alpha \vee .. \vee B\gamma_k\alpha \vee B\beta$. This clause fulfills the condition and completes the proof.*

By extending the arguments of the previous section it can be shown that F must be a canonical flattening of a clause G which is an nth power of C.

Theorem 24 (Canonical flattening and nth powers) *Let C' be a clause and D be a canonical flattening of C. The clause F is an nth power of D only if F is a canonical flattening of an nth power of C.*

Proof. *The proof is by mathematical induction on k. The base case, $n = 1$, is true since D is a canonical flattening of C. Assume it is true for all n up to k and prove for $k+1$. F in D^{k+1} is constructed by resolving D with F' in D^k. From the inductive hypothesis and Lemma 17, F is a flattening of a clause in D^{k+1}. Since the mgu θ involved in constructing F is simply a composition of renamings it follows from Lemma 11 that θ is a renaming. The set difference between F and F' is a renaming of a subset of D and thus F is a canonical flattening of a clause in D^{k+1}. This completes the proof.*

This allows a slight variant of Algorithm 2 to be used for constructing the nth roots of a given clause. In the following we assume that n is given and that F contains the literals $l\beta$ and $\overline{l'\alpha}$.

Algorithm 3 (nth root algorithm)

1. *Canonically flatten the input clause G to give clause F.*
2. *Choose from F a pair of literals $(l\beta, \overline{l'\alpha})$ with the same predicate symbol.*
3. *Choose two equal cardinality conjuncts of literals $B\alpha$ and $B\beta$ from clause $H = (F - \{l\beta, \overline{l'\alpha}\})$ each of which have an nth of the cardinality of H from which B is constructed by taking the least generalisation of corresponding pairs of literals which are alphabetic variants.*

4. *Construct renamings α and β by matching B to $B\alpha$ and $B\beta$.*

5. *Apply α^r to $l\alpha$ and β^r to $l'\beta$ to get l and l' respectively.*

6. *Return $C = unflat(l \leftarrow l' \wedge B)$ if G is the square of C.*

The following example demonstrates Algorithm 3 on Plotkin's example from Section 1 with $n = 4$.

Example 25 (Trace of Algorithm 3)
Let clause G be $p(f(f(f(f(X))))) \leftarrow p(X)$.
The steps in Algorithm 3 are followed below.

1. *F is $p(H) \leftarrow p(X)$, $H = f(I)$, $I = f(J)$, $J = f(K)$, $K = f(X)$.*

2. *Let $l\alpha = p(H)$ and $l'\beta = p(X)$.*

3. *Let $B\alpha$ be $(H = f(I))$, $B\beta$ be $(K = f(X))$ and B be $(L = f(M))$.*

4. *α is the renaming $\{L/H, M/I\}$ and β is the renaming $\{L/K, M/X\}$.*

5. *α^r is the renaming $\{H/L, I/M\}$, β^r is the renaming $\{K/L, X/M\}$, $l = l\alpha\alpha^r = p(L)$ and $l' = l'\beta\beta^r = p(M)$.*

6. *Return $C = unflat(p(L) \leftarrow p(M), L = f(M))$ which is $p(f(M)) \leftarrow p(M)$. G is the fourth power of C.*

The following theorem shows the completeness and correctness of Algorithm 3.

Theorem 26 (Completeness and correctness of Algorithm 3) *Let C be a clause and G be an nth power of C. When Algorithm 3 is presented with G there is a set of choices made in steps 1, 2 and 3 which will construct an alphabetic variant of C.*

Proof. *Algorithm 2 is correct since step 6 guarantees that any solution returned will be a square root of C. Therefore it is necessary to show its completeness. This is guaranteed by Theorems 23 and 24.*

6.1 Problems with Algorithm 3

The primary problem with applying Algorithm 3 is deciding the value of n for any given clause. One clue here comes from the form of the canonically flattened clause shown in Section 6. Since clause $H = (F - \{l\alpha, \overline{l'\beta}\})$ contains n instances of B, the number of occurrences of every predicate and function symbol in H must be a multiple of n. From this fact there should be a small finite number of candidate values for n for any given clause. However, it should be noted that literals and terms can be lost in the three ways listed in Section 5.6.

Note also that Corollary 3 says that in order to construct clause C which implies clause D, it is necessary to first construct a clause E which subsumes D. But how should E be chosen? One way is to drop literals from a canonical flattening of D until appropriate numbers of occurrences of predicate and function symbols remain in E. Then take the nth root to give C.

7 Implication and background knowledge

In the normal setting of Inductive Logic Programming [14] generalisation is carried out in the presence of background knowledge. In this section the solution to inverting implication between clauses is extended to the case in which background knowledge is present.

Assume a background clausal theory T and a clause (or example) C which is not entailed by T. Assume that there is a single clause D such that

$$T \wedge D \models C$$

This problem can be transformed to one involving implication between single clauses as follows.

$$\begin{aligned} T \wedge D &\models C \\ D &\models (T \rightarrow C) \\ &\models D \rightarrow (T \rightarrow C) \\ &\models D \rightarrow \overline{(T \wedge \overline{C})} \\ &\models D \rightarrow \overline{(l_1 \wedge l_2 \wedge \ldots)} \end{aligned}$$

In the last line $(T \wedge \overline{C})$ is replaced with a conjunction of all ground literals which can be derived from $(T \wedge \overline{C})$. This can be viewed as replacing the formula with a model of the formula. Since $(l_1 \wedge l_2 \wedge \ldots)$ is a conjunction of literals, the last line above represents implication between two clauses. The clause $(\overline{l_1} \vee \overline{l_2} \vee \ldots)$ can be constructed to be of finite length if T is generative (see [16]) and elements of the model are only constructed to a finite depth of resolution. This clause can then be used to construct D using the methods described in previous sections.

8 Conclusion

In this paper the general problem of inverting implication is discussed. This problem is at the heart of research into Inductive Logic Programming and Machine Learning in general since all forms of generalisation involve inverting implication. The methods and algorithms described in this paper are derived from a first principles approach to the problem and extend previous approaches such as those using inverse resolution [15, 14, 22, 24] and relative least general generalisation [19, 2, 16].

Although a first attempt has been made at this problem in a previous paper by Lapointe and Matwin [11] the author believes the approach taken in this paper to be more general and comprehensive. Various remaining problems with the square root and nth root algorithms are described in Sections 5.6 and Section 6.1. The problems of time complexity of these algorithms is not discussed here. Also no implementation of the approach described in this paper has been made.

As Lapointe and Matwin noted, the advantages of extending the generalisation techniques beyond those of inverse resolution [14] lie in the fact that fewer examples are required to learn recursive clauses. Recursive clauses have not been vital for the

success of several real world applications of Inductive Logic Programming [17, 10, 6, 5]. However, they are of central interest within problems involving construction of arbitrary programs. Traditionally this area has involved deductive techniques within the area known as formal methods. According to a recent paper by Hoare [9]

> Given specification S, the task is to find a program P which satisfies it, in the sense that every possible observation of every possible behaviour of the program P will be among the behaviours described (and therefore permitted by) the specification of S. In logic, this can be assured with mathematical certainty by a proof of the simple implication
>
> $$\vdash P \rightarrow S.$$

Note that this problem is encompassed by the discussion in this paper. However, the requirements for Inductive Logic Programming are slightly weaker than those described by Hoare, since if the specification S is an incomplete set of examples then not all behaviours of P are defined. However, there is nothing in the present discussion to stop one making use of arbitrary (non-ground) formulae instead of examples. Such formulae could comprise a specification in the sense that "every possible observation of every possible behaviour of the program P will be among the behaviours described". The background knowledge referred to in Inductive Logic Programming maps to the the set of abstract data types and data operations used in formal methods approaches. Note also that using the approach in this paper $P \rightarrow S$ should be ensured by construction and is similar in that way to the approach of transformational programming introduced first by Burstall and Darlington [3].

Acknowledgements.
The author would like especially to thank Ashwin Srinivasan for useful input during this research. Thanks are also due to Wray Buntine, Stuart Russell and members of the Turing Institute Inductive Logic Programming group for helpful and interesting discussions on the topics in this paper. This work was supported partly by the Esprit Ecoles project 3059 and an SERC Post-graduate fellowship held by the author.

Appendix

A Definitions from logic

A.1 Formulae in first order predicate calculus

A variable is represented by an upper case letter followed by a string of lower case letters and digits. A function symbol is a lower case letter followed by a string of lower case letters and digits. A predicate symbol is a lower case letter followed by

a string of lower case letter and digits. The negation of F is $\overline{F}$. A variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when f, g and h are function symbols and X is a variable. A predicate symbol immediately followed by a bracketted n-tuple of terms is called an atomic formula, or atom. Both l and $\overline{l}$ are literals whenever l is an atomic formula. In this case l is called a positive literal and $\overline{l}$ is called a negative literal. The literals l and $\overline{l}$ are said to be each others complements and form, in either order, a complementary pair. A finite set (possibly empty) of literals is called a clause. The empty clause is represented by $\square$. A clause represents the disjunction of its literals. Thus the clause $\{l_1, l_2, .. \overline{l_i}, \overline{l_{i+1}}, ...\}$ can be equivalently represented as $(l_1 \vee l_2 \vee .. \overline{l_i} \vee \overline{l_{i+1}} \vee ...)$ or $l_1, l_2, .. \leftarrow l_i, l_{i+1}, ...$ A Horn clause is a clause which contains at most one positive literal. A definite clause is a clause which contains exactly one positive literal. The positive literal in a definite clause is called the head of the clause while the negative literals are collectively called the body of the clause. A set of clauses is called a clausal theory. The empty clausal theory is represented by $\blacksquare$. A clausal theory represents the conjunction of its clauses. Thus the clausal theory $\{C_1, C_2, ...\}$ can be equivalently represented as $(C_1 \wedge C_2 \wedge ...)$. A set of Horn clauses is called a logic program. Apart from representing the empty clause and the empty theory, the symbols $\square$ and $\blacksquare$ represent the logical constants *False* and *True* respectively. Any clause, such as $l \leftarrow l$, which is equivalent to $\blacksquare$ is said to be a *tautology*. Literals, clauses and clausal theories are all well-formed-formulae (wff's). Let E be a wff or term. $vars(E)$ denotes the set of variables in E. E is said to be ground if and only if vars$(E) = \emptyset$.

A.2 Models and substitutions

A set of ground literals which does not contain a complementary pair is called an interpretation. Let M be an interpretation, C be a clause and $\mathcal{C}$ be the set of all ground clauses obtained by replacing the variables in C by ground terms. M is a model of C if and only if each clause in $\mathcal{C}$ contains at least one literal found in M. M is a model for clausal theory T if and only if M is a model for each clause in T. Let F_1 and F_2 be two wff's. F_1 semantically entails F_2, or $F_1 \models F_2$ if and only if every model of F_1 is a model of F_2. F_1 is said to syntactically entail F_2 using I, or $F_1 \vdash_I F_2$, if and only if F_2 can be derived from F_1 using the set of deductive inference rules I. The set of inference rules I is said to be deductively sound and complete if and only if $F_1 \vdash_I F_2$ whenever $F_1 \models F_2$. In this case the subscript can be dropped and one can merely write $F_1 \vdash F_2$. Let F_1 and F_2 be two wff's. F_1 is said to be more general than F_2 if and only if $F_1 \vdash F_2$. A wff F is satisfiable if there is a model for F and unsatisfiable otherwise. F is unsatisfiable if and only if $F \models \square$. The deduction theorem states that $F_1 \wedge F_2 \models F_3$ if and only if $F_1 \models F_2 \rightarrow F_3$.

Let $\theta = \{v_1/t_1, .., v_n/t_n\}$. θ is said to be a substitution when each v_i is a variable and each t_i is a term, and for no distinct i and j is v_i the same as v_j. The set $\{v_1, .., v_n\}$ is called the domain of θ, or dom(θ), and $\{t_1, .., t_n\}$ the range of θ, or rng(θ). Lower case Greek letters are used to denote substitutions. Let E be a well-formed formula or a term and $\theta = \{v_1/t_1, .., v_n/t_n\}$ be a substitution.

The instantiation of E by θ, written $E\theta$, is formed by replacing every occurrence of v_i in E by t_i. Let F be a well-formed formula and $\theta = \{u_1/v_1, .., u_n/v_n\}$ be a substitution. θ is a renaming of F if and only if $u_1, .., u_n$ are all distinct variables, $v_1, .., v_n$ are all distinct variables and $(vars(F) - \{u_1, .., u_n\}) \cap \{v_1, .., v_n\} = \emptyset$. Every sub-term within a given term or literal l can be uniquely referenced by its *place* within l. Places within terms or literals are denoted by n-tuples of natural numbers and defined recursively as follows. The term at place $\langle i \rangle$ within $f(t_0, .., t_m)$ is t_i. The term at place $\langle i_0, .., i_n \rangle$ within $f(t_0, .., t_m)$ is the term at place $\langle i_1, .., i_n \rangle$ in t_{i_0}. Let t be a term found at place p in literal l, where l is a literal within clause C. The place of t in C is denoted by the pair $\langle l, p \rangle$. Let E be a clause or a term and $\theta = \{v_1/t_1, .., v_n/t_n\}$ be a substitution. The corresponding inverse substitution θ^{-1} is $\{\langle t_1, \{p_{1,1}, .., p_{1,m_1}\}\rangle/v_1, .., \langle t_n, \{p_{n,1}, .., p_{1,m_n}\}\rangle/v_n\}$. An inverse substitution is applied by replacing all t_i at places $p_{i,1}, .., p_{i,m_i}$ within E by v_i. Clearly $E\theta\theta^{-1} = E$. Note that an inverse substitution is not strictly a substitution but rather a rewrite. Let C and D be clauses. It is said that C θ-subsumes D if and only if there exists a substitution θ such that $C\theta \subseteq D$.

A.3 Resolution

Let F_1 and F_2 be two wff's and θ be a renaming of F_1. $F_1\theta$ and F_2 are said to be standardised apart whenever there is no variable which occurs in both $F_1\theta$ and F_2. If θ is used to standardise apart formula F and $F\theta$ then F and $F\theta$ are said to be alphabetic variants. The substitution θ is said to be the unifier of the atoms l and l' whenever $l\theta = l'\theta$. μ is the most general unifier (mgu) of l and l' if and only if for all unifiers γ of l and l' there exists a substitution δ such that $(l\mu)\delta = l\gamma$. $((C - \{l\}) \cup (D - \{\overline{l'}\}))\theta$ is said to be the resolvent of the clauses C and D whenever C and D are standardised apart, $l \in C$, $\overline{l'} \in D$, θ is the mgu of l and l'. That is to say that $\langle l\theta, \overline{l'}\theta \rangle$ is a complementary pair. The resolvent of clauses C and D is denoted $(C \cdot D)$ when the complementary pair of literals is unspecified. The '$\cdot$' operator is commutative, non-associative and non-distributive.

Let T be a clausal theory. Robinson [21] defined the function $\mathcal{R}^n(T)$ recursively as follows. $\mathcal{R}^0(T) = T$. $\mathcal{R}^n(T)$ is the union of $\mathcal{R}^{n-1}$ and the set of all resolvents constructed from pairs of clauses in $\mathcal{R}^{n-1}(T)$. Robinson showed that T is unsatisfiable if and only if there is some n for which $\mathcal{R}^n(T)$ contains the empty clause ($\square$).

References

[1] M. Bain and S. Muggleton. Non-monotonic learning. In D. Michie, editor, *Machine Intelligence 12*. Oxford University Press, 1991.

[2] W. Buntine. Generalised subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24:44–67, 1977.

[4] B. Carlson, F. Kant, and Wünsche. A scheme for functions in logic programming. UPMAIL 57, Uppsala Programming Methodology and Artificial Intelligence Laboratory, Uppsala, Sweden, 1989.

[5] B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[6] C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S.H. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[7] E. Giovannetti and C. Moiso. Some aspects of the integration between logic programming and functional programming. In P Jorrand and V. Sgurev, editors, *Artificial Intelligence II. Methodology, Systems, Applications*, pages 69–79. North-Holland, 1987.

[8] G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, 1987.

[9] C.A.R. Hoare. Programs are predicates. In *Proceedings of the Final Fifth Generation Conference*, Tokyo, 1992. Ohmsha.

[10] R. King, S. Muggleton, and M.J.E. Sternberg. Drug design by machine learning. *Proceedings of the National Academy of Sciences*, 1992. To appear.

[11] S. Lapointe and S. Matwin. Sub-unification: a tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Machine Learning Conference*, Los Altos, 1992. Morgan Kaufmann.

[12] C. Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms.* PhD thesis, University of California, Berkeley, 1967.

[13] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[14] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[15] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Kaufmann, 1988.

[16] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.

[17] S. Muggleton, R. King, and M. Sternberg. Predicting protein secondary structure using inductive logic programming, 1991. submitted to Proteins.

[18] T. Niblett. A study of generalisation in logic programs. In *EWSL-88*, London, 1988. Pitman.

[19] G.D. Plotkin. *Automatic Methods of Inductive Inference.* PhD thesis, Edinburgh University, August 1971.

[20] R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[21] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.

[22] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In S.H. Muggleton, editor, *Inductive Logic Programming.* Academic Press, London, 1992.

[23] C. Rouveirol and J-F Puget. A simple and general solution for inverting resolution. In *EWSL-89*, pages 201–210, London, 1989. Pitman.

[24] R. Wirth. Completing logic programs by inverse resolution. In *EWSL-89*, pages 239–250, London, 1989. Pitman.

Confirmation Theory and Machine Learning

by DONALD GILLIES (EMAIL d.gillies @ uk.ac.kcl.cc.elm)

Centre for Logic and Probability in Information Technology,
King's College London,
University of London,
Strand,
London WC2R 2LS, UK*

1 INTRODUCTION

So far machine learning has been developed without taking a great deal of account of what is known as confirmation theory. In this paper I want first to argue that confirmation theory is in fact needed for machine learning. I will then go on to give a survey of various approaches to confirmation theory, and will try to show how they can be related to machine learning.

In very abstract and schematic terms, the aim of machine learning is to write programs which will derive generalisations, or rules, or hypotheses, or theories from data. Typically a program is fed with some initial or training data, and obtains a hypothesis which agrees with this data. The hypothesis is then tested out against some further data. Let us suppose that the hypothesis (H say) accords with this further data. We then have a situation in which H successfully explains a body of data (E say) comprising both the training and the test data. Before we can use H in practice, however, we need to form an estimate of how well H is confirmed (or corroborated) by E. Only if the confirmation is high, would we be justified in using H in some practical application. We thus need to form an estimate of the degree of confirmation of H by E, which is written C(H,E). Now confirmation theory is quite simply the investigation of the function C(H,E), and we can therefore see why confirmation theory is needed in machine learning.

Confirmation theory has been studied by philosophers of science and by statisticians, who naturally deal with the case

in which H is a statistical hypothesis. Unfortunately the subject has proved difficult, and rather intractable. There is in fact no generally agreed way of defining the confirmation function C(H,E). Nonetheless some progress has been made. A number of different approaches have been suggested and developed, and I will try in this paper to explain some of these and show how they might be applied to machine learning. One initial point is worth making at this stage. I have so far given the confirmation function in the form C(H,E), but, in reality, the evidence E is always taken in conjunction with some assumed background knowledge (B say). We should, therefore, strictly speaking, write C(H, E&B) rather than C(H,E). For simplicity I will usually follow the standard convention of not writing B explicitly; but the presence of B should not be forgotten, and sometimes we shall have to take conscious account of B.

Bayesianism is one of the leading approaches to confirmation theory. It involves two assumptions. The first is that the confirmation function obeys the ordinary axioms of probability. This is written C(H,E) = Pr(H|E), which can be read as stating that confirmation is a probability function. The second is that changes in confirmation can be calculated using Bayes' Theorem. This process is known as Bayesian conditionalisation. Now the equation C(H,E) = Pr(H|E) may at first sight seem trivially true, since in ordinary language we tend to use expressions like E confirms H, and E renders H probable more or less interchangeably. It is important to realize, however, that this ordinary language argument is very far from establishing Bayesianism. The claim that the function C(H,E) should satisfy a very specific set of axioms (the probability axioms) is a substantial claim, and one which might not be true. In fact there are non-Bayesian approaches to confirmation theory developed by Popper and others in which the confirmation function C(H,E) does not satisfy the axioms of probability.

My own preference is for non-Bayesian confirmation theory. I have criticized Bayesianism in detail in my [1988], and will present one of the main arguments against Bayesianism at the beginning of section 3. This will lead to a development based on the Turing-Good Weight of Evidence Function. Then in section 4 I will describe the SES measure of confirmation which is non-Bayesian in character. This is developed using Popper's measure of the severity of a test, but differs from Popper's own confirmation function. It is introduced here for the first time.

These general considerations about confirmation theory will however become rather abstract and unsatisfactory unless they are related to a specific method for machine learning. For this purpose I have selected the approach to machine learning introduced by Stephen Muggleton in his [1988], and at present being developed by Stephen Muggleton and his colleagues at the Turing Institute in Glasgow, UK[1]. This approach has roots in the formal theory of algorithmic complexity. Naturally the particular ideas developed should be applicable in other approaches to machine learning. It should be added, however,

that my investigation does lend support to the Turing Institute's new approach to machine learning. As we shall see, this approach depends crucially on the notion of k-bit compression, and, as I will show particularly in sections 3 and 4, this concept agrees very well with concepts from confirmation theory (e.g. the Turing-Good weight of evidence function) which were developed in contexts quite different from machine learning. Theoretical convergence of this kind is a hopeful sign of being on the right track.

Let us begin by looking at those features of Muggleton's system which we shall need to use. Suppose that we have a set of positive instances E^+, and a machine learning system which induces theories (which may be thought of as programs) from these instances. The system has induced a theory P which successfully explains the positive instances, so that $P \vdash E^+$. The question now arises as to whether P is trustworthy, or whether its apparent success is just due to chance. To tackle this problem Muggleton supposes that we can define a reference Turing Machine T_r ([1988], p. 126):

'.. which, given an encoded version of P as input generates an encoded version of E^+ as output.
Thus

$$T_r(I(P)) = O(E^+)$$

where $I(P)$ is an input tape encoding of P, $O(E^+)$ is an output tape encoding of E^+ and $I(P)$ is k bits shorter that $O(E^+)$. ... We will use X_k to denote the statement that there exists such a k-bit compressed explanation $I(P)$ of $O(E^+)$.'

So, given that P provides a k-bit compressed explanation of the positive instances, we have to consider what support or confirmation this gives to P. I will attempt to tackle this problem by considering in turn some standard approaches to confirmation theory. For completeness I will begin with classical statistics, although, as will become apparent, this is purely falsificationist and does not introduce a confirmation function.

2 CLASSICAL STATISTICS

On this approach we set up a null hypothesis, and see if it is refuted by the evidence. Muggleton suggests ([1988], p. 126) that 'we might take the null hypothesis to be that the examples E' were produced by tossing a coin.' Muggleton denotes this null hypothesis by $\overline{\mathcal{H}}$, and shows that, from an earlier result in his paper, we have

$$Pr(X_k \mid \overline{\mathcal{H}}) \leq 2^{-k} \qquad - (1)$$

In the standard approach of classical statistics we try to refute the null hypothesis by setting up a rejection class whose size (probability) is suitably small (usually less than 5%). The null hypothesis is then regarded as falsified if the observed result lies in this rejection class, and confirmed otherwise.

In the present instance the rejection class should obviously be of the form

$$X_r \cup X_{r+1} \cup \ldots \cup X_n \qquad - (2)$$

so that we regard the null hypothesis as falsified if $r \leq k \leq n$ i.e. if the compression is sufficiently high. Now

$$Pr(X_r \cup X_{r+1} \cup \ldots \cup X_n) \leq 2^{-r} + 2^{-r-1} + \ldots + 2^{-n}$$

$$= 2^{-r+1} - 2^{-n} \qquad - (3)$$

For large n, this is approximately equal to 2^{-r+1}. So, if $r-1 = 5$ i.e. $r = 6$, Pr(Rejection Class) < 3%. Hence if we regard the null hypothesis as falsified when compression $k \geq 6$, we will be wrong in less than 3% of cases.

It should be noted, however, that 97% is <u>not</u> used in classical statistics as a measure of confirmation. One of the main points of this section is to stress that this would be an erroneous interpretation. In fact classical statistics has a purely falsificationist approach. Hypotheses are set up, and tested to see whether they are refuted or falsified at some level of significance, but no attempt is made to estimate the degree of confirmation of a hypothesis which passes a number of tests. To obtain a measure of degree of confirmation, we need to look at some of the other approaches.

3 WEIGHT OF EVIDENCE FUNCTIONS

We will begin this section by describing a well-known general difficulty with Bayesianism. Suppose that a particular individual (Mr A say) assigns a probability Pr(X) to X. This surely implies that, if asked to bet on whether X is true, Mr A will be prepared to do so at the rate p = Pr(X). Now consider the case in which X is a universal hypothesis, say 'All Ravens are Black'. If Mr A bets that H is true, he might lose since a non-Black Raven might be discovered, but he could never win because it can never be definitely established that a universal hypothesis is true. However many Black Ravens we have observed, it always remains possible that a non-Black Raven will appear in the future, showing that the universal generalisation is false. It follows that the only reasonable rate at which Mr A can bet on H is 0 - a rate which is really equivalent to Mr A's refusing to bet. If Mr A bets at any rate p > 0 on H being true, he might lose, but cannot win. This argument seems to establish that Pr(H) = 0 for any universal hypothesis H. But now consider Bayes' Theorem

$$Pr(H|E) = \frac{Pr(E|H)\ Pr(H)}{Pr(E)} \qquad - (4)$$

If Pr(H) = 0, it follows that Pr(H|E) = 0. So it seems that on the Bayesian position, C(H,E) = 0 for any universal hypothesis H and any evidence E. This is obviously completely counter-intuitive.

There have been some suggestions for resolving this problem (see Gillies [1988], pp. 192-9, and Cussens [1991], Chs.2 and 3, pp. 20-49). Moreover in many machine learning situations we are dealing with hypotheses of finite extent rather than universal hypotheses. Yet even if Pr(H) can be taken as non-zero, it is often not clear what its value should be. Bayesianism needs prior probabilities, but prior probabilities are characteristically difficult to evaluate in a way which is not arbitrary. This suggests that it is worth exploring non-Bayesian approaches to confirmation theory which avoid the use of prior probabilities altogether. These are best approached by a consideration of weight of evidence functions.

To introduce the concept of <u>weight of evidence</u>, we must begin by writing in the background evidence B explicitly. C(H,E&B) stands for the total confirmation given to H by both E and B. Weight of evidence W(H,E,B) is a 3-place function[2], and represents the contribution made to the total confirmation by the individual item of evidence E against a background B. Weight of evidence functions can be either <u>additive</u> or <u>multiplicative</u>. For the additive case, we have

$$C(H,E\&B) = W(H,E,B) + C(H,B) \qquad - (5)$$

This shows that if the prior confirmation can be neglected, we can use W(H,E,B) as our confirmation function. In general, however, this will lead to a non-Bayesian confirmation function. For the multiplicative case

$$C(H,E\&B) = W(H,E,B)\ C(H,B) \qquad - (6)$$

Bayes' Theorem (4) with the background knowledge made fully explicit takes the form

$$Pr(H|E\&B) = \frac{Pr(E|H\&B)}{Pr(E|B)}\ Pr(H|B) \qquad - (7)$$

Comparing the two equations (6) and (7), we see that the Bayesian weight of evidence function (BW say) is multiplicative, and given by

$$BW(H,E,B) = \frac{Pr(E|H\&B)}{Pr(E|B)} \qquad - (8)$$

Now if we want to convert this multiplicative function to an additive one, we can do so simply by taking logs. If we do so,and revert to the convention of not writing in the background knowledge explicitly, we obtain the following additive weight of evidence function

$$W(H,E) = \log Pr(E|H) - \log Pr(E) \qquad - (9)$$

This is in fact the Turing-Good Weight of Evidence Function for the case in which Pr(H) = 0. This function was introduced by Turing and Good in their cryptanalytic work during the Second World War, and developed by Good after the War. For historical details and discussions see Gillies [1990], Good [1979], [1983] Ch. 15, p. 159 and [1985], and Hodges [1983].

Applying the Turing-Good Weight of Evidence Function to Muggleton's machine learning situation, we get

$$W(P,E^+) = \log Pr(E^+|P) - \log Pr(E^+) \qquad - (10)$$

Since E^+ is generated by the program P, we have $Pr(E^+|P) = 1$, and so $\log Pr(E^+|P) = 0$. The evaluation of $\log Pr(E^+)$ is , however, a little more difficult, since it involves another principle of confirmation theory (the principle of explanatory surplus) which we must next describe.

A particular case of the principle of explanatory surplus arises in the following situation (cf. Gillies [1989] pp. 377-9). Suppose we have n facts $f_1, \ldots, f_n$, and these are explained using s theoretical assumptions $T_1, \ldots, T_s$. The principle of explanatory surplus states that $T_1, \ldots, T_s$ are supported not by all the facts which they explain, but only by that fraction of the facts which can be considered an explanatory surplus. The simplest and most straightforward way of estimating

the size of the explanatory surplus is to subtract the number of theoretical assumptions used from the number of facts explained, giving the value n-s.

The principle of explanatory surplus was developed to explicate the role of confirmation in the history of science, and in Gillies [1989] it is applied to the case of Newton's theory, taken as comprising the 3 laws of motion , and the law of gravity. In historical cases of this sort, the division of the theory into s theoretical assumptions, and of the evidence into n facts presents problems, and inevitably involves some arbitrariness. The situation is better in artificial intelligence, where the use of formalised languages usually creates a natural division into discrete assumptions or facts.

Returning now to Muggleton's machine learning, it is natural to measure both the number of theoretical assumptions in the program P, and the number of facts in the evidence E^+ by the number of bits in their respective codings. If there are n bits in E^+, then, since we have k-bit compression, there are n-k bits in P. Thus applying the principle of explanatory surplus, we regard only n - (n-k) = k bits of the evidence as actually supporting P. It is reasonable to assign these k bits of evidence the prior probability 2^{-k}, and so taking logs to the base 2, the value of the Turing-Good Weight of Evidence Function in the Muggleton machine learning case turns out to be

$$W(P,E^+) = k \qquad - (11)$$

This result is both striking and encouraging. The Turing-Good Weight of Evidence Function and the Principle of Explanatory Surplus were developed in contexts completely different from that of machine learning. Yet when they are applied to Muggleton's machine learning approach, it turns out that the weight of evidence is equal to k, the key parameter in the approach. This theoretical convergence can, we think, be taken as indicating that Muggleton's approach may be on the right lines.

We must next point out that so far we have considered only the special case in which the observed evidence E^+ is all derived from the program P. It may however be worth considering the more general case in which the program generates some, but not all of the observed evidence E^+. Let us suppose that $E^+ = E^{(1)} \,\&\, E^{(2)}$, where $E^{(1)}$ is successfully generated by the program P, but where the program generates something different from $E^{(2)}$. Let us suppose further that $E^{(1)}$ is r bits long in its coding, and $E^{(2)}$ is s bits long in its coding, where r + s = n. In this situation $Pr(E^+|P) = 0$, so that $\log Pr(E^+|P) = -\infty$. The Turing-Good Weight of Evidence Function thus becomes always negatively infinite. This corresponds to the fact that the hypothesis represented by P has been refuted by the negative evidence $E^{(2)}$. There may, however, be a practical situation in which our program P generates most of the observed evidence E^+, but fails in some

cases. Strictly speaking we should simply regard P as refuted, but we may in practice want to regard the cases of failure simply as exceptions or noise, and P as still quite well confirmed. The Turing-Good Weight of Evidence Function cannot represent confirmation in this sense. In the next section, therefore, we will present another measure of confirmation - the SES measure - which has been developed from Popper's measure of the severity of a test, and which gives intuitively satisfactory results in the case just described.

4. THE SES MEASURE OF CONFIRMATION

Popper's fundamental approach to the theory of corroboration is contained in what could be called the principle of severe testing. Popper formulates this principle as follows ([1934], p. 267):

' ... it is not so much the number of corroborating instances which determines the degree of corroboration as the severity of the various tests to which the hypothesis in question can be, and has been, subjected.'

But how do we measure the severity of a test? Suppose the test is of a hypothesis H, and yields evidence E. Then Popper considers a quantity Q defined by

$$Q = Pr(E|H\&B) - Pr(E|B) \qquad - (12)$$

Let us first consider the case in which Q is positive. Q is large if Pr(E|H&B) is large, i.e. if E is highly probable given H&B, and if Pr(E|B) is small, i.e. if E is highly improbable given B but not H. In other words Q is large if E is improbable without H, but probable with H. If E satisfies these conditions, then it is reasonable to say that the corresponding test is severe, and Q may be taken as a measure of the severity of the test.

Let us next consider the case in which Q is negative. (It turns out to be convenient to deal with the positive and negative cases separately.) |Q| is now large if Pr(E|B) is large, and Pr(E|H&B) is small. In other words |Q| is large if E is very probable given the background knowledge alone, but becomes very improbable if H is added to the background evidence. In these circumstances it is reasonable to take |Q| as measuring the extent to which E undermines the hypothesis H.

It should be noted that Popper's measure Q is closely related both to the Bayesian Weight of Evidence Function (8), and to the case of the Turing-Good Weight of Evidence Function given in equation (9).

We will now describe a confirmation function which we will call the SES measure of confirmation. SES is an acronym for Severity and Explanatory Surplus. The measure is formed by

combining Popper's Q function with the principle of explanatory surplus. The SES measure is thus partly based on Popper's ideas, but it should be noted that Popper's own corroboration function is different from the SES measure. An account of Popper's own corroboration function is to be found in his [1959] and [1983], Ch. IV, pp. 217-55.

The SES measure of confirmation is an ordered pair $\langle +c,-u \rangle$, where c and u are both positive. We express the evidence E as $E^{(1)}\&E^{(2)}$ where $E^{(1)}$ supports the hypothesis to degree c, and $E^{(2)}$ undermines it to degree u. The calculation of c and u is carried out as follows. We first of all consider the evidence E as giving the results of a number (n say) of tests of the hypothesis H.[3] We then consider those tests for which Popper's Q is positive, and sum the values of Q over these tests to give c'. We then estimate the fraction (f say) of the results of the positive tests which may be considered an explanatory surplus, and set $c = c'f$. To obtain -u, we simply sum the values of Q for the tests for which Q is negative. The SES measure is implicitly setting the confirmation or disconfirmation of the hypothesis H on background knowledge equal to zero, and it might be desirable in some cases to correct the measure to take explicit account of the confirmation or disconfirmation of H on B. One advantage of the SES measure of confirmation is that the numbers have a simple intuitive meaning. Suppose that $C(H,E) = \langle +c,-u \rangle$. This means that H has passed the equivalent of c tests of maximum severity, and failed the equivalent of u maximally undermining tests.

We will now apply the SES measure of confirmation to Muggleton's machine learning in the case considered at the end of section 3 in which the evidence E^* is divided into $E^{(1)}\&E^{(2)}$. In this approach we have to consider the evidence E^* as the result of a number of tests of the hypothesis P. Since E^* consists of n bits, we can take each bit as representing a test.[4] Consider a particular evidential bit (EBIT say). Its values can be coded as 0 and 1, so that $Pr(EBIT|B) = 1/2$. If P generates the observed value EBIT i.e. P passes this test, we have $Pr(EBIT|P\&B) = 1$, so that Q of EBIT = 1 - 1/2 = 1/2. If P generates a value different from EBIT i.e. P fails the test, we have $Pr(EBIT|P\&B) = 0$. so that Q of EBIT = -1/2.

Let us now consider $E^{(1)}$, which, by assumption, contains r bits. We first sum Q over these r bits to obtain $c' = r/2$. We have next to estimate the fraction f of $E^{(1)}$ which can be considered as an explanatory surplus. P has k-bit compression, and so consists of n-k bits. Thus the explanatory surplus is $r - (n-k) = r+k-n$ if $r+k>n$, or = 0 if $r+k \leq n$. Thus $c = c'f$, where $f = r+k-n/r$ if $r+k>n$, or = 0 if $r+k<n$. Turning now to $E^{(2)}$, which, by assumption, contains s bits where $r+s = n$, we simply sum Q over these s bits to obtain $u = -s/2$.

So the SES measure of confirmation in this case is as follows:

$$C(P,E^+) = \langle r+k-n/2, -s/2 \rangle \quad \text{if } r+k>n$$
$$= \langle 0, -s/2 \rangle \quad \text{if } r+k \leq n \qquad - (13)$$

It is interesting to observe the value of C in the case in which all the evidence is positive, i.e. $E^+ = E^{(1)}$, r=n, and s=0. We have

$$C(P,E^+) = \langle k/2, 0 \rangle \qquad - (14)$$

In this case, then, the positive confirmation is half the value of the Turing-Good Weight of Evidence Function, and, once again, the parameter k is of crucial importance.

FOOTNOTES

* This paper is part of the Rule-Based Systems project, and I would like to acknowledge the support of a Science and Engineering Research Council grant: GR/G 29854. Earlier versions of the paper were read at project meetings, and I am most grateful for comments and suggestions from other members of the project, particularly James Cussens, Dov Gabbay, Tony Hunter, Steve Muggleton, and Ashwin Srinivasan.

[1]For a more recent account, incorporating further developments, see Muggleton, Srinivasan, and Bain [1992].

[2]The importance of weight of evidence, and the fact that it is a 3-place function are rightly stressed by Good (cf. his [1983], Ch. 15, para 4, pp. 159-60).

[3]The division of evidence into discrete tests may be somewhat arbitrary, just as we saw earlier that there is an arbitrariness in the division of a theory into a number of theoretical assumptions, or of evidence into a number of facts. The use of formalized languages in machine learning, combined with considerations of context, should help to resolve this problem.

[4]Another approach might be to divide E' into 'examples' and take each example as a test. In general each example will need several bits to encode it, so that this would give somewhat different results. The meaning of the various calculations remains clear provided it is clearly understood what is being taken as a single test.

REFERENCES

CUSSENS, J. [1991]: Interpretations of Probability, Nonstandard Analysis and Confirmation Theory. PhD Thesis. King's College, London.

GILLIES, D.A. [1988]: 'Induction and Probability', in An Encyclopaedia of Philosophy, Edited by G.H.R.Parkinson, Routledge, Ch. 9, pp. 179-204.

GILLIES, D.A. [1989]: 'Non-Bayesian Confirmation theory, and the Principle of Explanatory Surplus', PSA 1988, Vol. 2, pp. 373-80.

GILLIES, D.A. [1990]: 'The Turing-Good Weight of Evidence Function and Popper's Measure of the Severity of a Test', British Journal for the Philosophy of Science, 41, pp. 143-6.

GOOD, I.J. [1979]: 'A.M.Turing's Statistical Work in World War II', Biometrika, 66, pp. 393-6.

GOOD, I.J. [1983]: Good Thinking. The Foundations of Probability and its Applications. University of Minnesota Press, Minneapolis.

GOOD, I.J. [1985]: 'Weight of Evidence: A Brief Survey' in J.M.Bernardo, M.H.De Groott, D.V.Lindley, A.F.M.Smith (Eds.) Bayesian Statistics 2, Proceedings of the Second Valencia International Meeting, September 6/10, 1983, North-Holland.

HODGES, A. [1983]: Alan Turing. The Enigma of Intelligence, Unwin Paperbacks, 1987.

MUGGLETON, S.H. [1988]: 'A strategy for constructing new predicates in first order logic', in Proceedings of the Third European Working Session of Learning, Pitman, pp. 123-30.

MUGGLETON, S.H., SRINIVASAN, A., AND BAIN, M. [1992]: 'Compression, Significance and Accuracy', Forthcoming

POPPER, K.R. [1934]: The Logic of Scientific Discovery, 6th Impression, Hutchinson, 1972.

POPPER, K.R. [1959]: 'Corroboration, the Weight of Evidence, and Statistical tests', Appendix *ix of POPPER [1934], pp. 387-419.

POPPER, K.R. [1983]: Realism and the Aim of Science, Hutchinson.

Towards Inductive Generalisation in Higher Order Logic

Cao Feng, Stephen Muggleton
The Turing Institute, George House, 36 North Hanover Street,
Glasgow G1 2AD, UK

Abstract

In many cases, higher order (Horn) clauses are more suited to express certain concepts when relations between predicates exist. However, to date there has been no appropriate higher order logic within which efficient inductive generalisation can be carried out. This paper describes inductive generalisation in M_λ – a higher order logic which not only retains the expressiveness of λ-calculus but also provides for effective and efficient inductive generalisation. The main strength of M_λ is twofold: it is a higher logic extension of the (clausal) first order logic and it can be mechanised in a way similar to the first order case in Horn clause form. For nonredundant M_λ normal terms, their least general generalisation (LGG) is unique, and so is their most general unification. Inductive generalisation in M_λ is implemented in the algorithm HOLGG. This algorithm has been applied to some interesting induction problems in the induction of higher order rule templates and automatic program transformation.

1 Introduction

Generalisation forms the basis of most inductive learning systems. In first order logic, generalisation has been well-understood [17, 18, 21], and many algorithms have been devised based on these principles [13]. Using a first order inductive tool [13], we can obtain the clauses:

$$\forall X \forall Y \forall Z. ancestor(X, Y) \leftarrow ancestor(X, Z), ancestor(Z, Y). \quad (1)$$

$$\forall X \forall Y \forall Z. less\_than(X, Y) \leftarrow less\_than(X, Z), less\_than(Z, Y). \quad (2)$$

which are the generalisation respectively of the facts: $less\_than(1,3), less\_than(2,4), \ldots$ and $ancestor(john, steve), ancestor(steve, mike), \ldots$

The clauses in (1) and (2) are very similar in the sense that (1) can be obtained from (2) by substituting the predicate symbol *ancestor* for *less_than*, and vice versa. If we allow for a special variable P, it is not difficult to see that both clauses in (1) and (2) are "substitution instances" of the clause:

$$\forall X \forall Y \forall Z. P(X, Y) : -P(X, Z), P(Z, Y). \quad (3)$$

with P being substituted by *ancestor* for (1) and *less_than* for (2). As P is a predicate variable, this clause needs a higher order language that goes beyond the first order predicate logic used in present machine learning research and applications.

In general, a term E is more general than another term F whenever there is a substitution θ for which $E\theta = F$. Therefore the formula in (3) is more general than both in (1) and (2). Because the clause in (3) contains the higher order variable "P", its meaning is not clear before it is made precise by formal semantic and syntactic definitions. Thus we need to consider the problem of how P, X, Y and Z are interpreted in (1), (2) and (3). This

has been partially tackled by logic programming in higher order Horn clauses [14], which incorporates new methods to deal with higher-order terms.

Obviously, the higher order clause is a more powerful representation both in terms of expressiveness and efficiency. Conversely, if we are given the higher order clause such as the clause in (3) we hope that we can more quickly find the rules (1) and (2) with respect to some given facts (perhaps in higher order).

2 Motivation

In order to introduce a higher order language we need to address several issues pertinent in inductive learning. The main concerns are:

- **Expressiveness.** Meta-level information about inductive problems can, in many cases, only be expressed adequately with higher order terms. The induction of program transformation rules is [7] is a case in point.

- **Induction.** Inasmuch machine learning is concerned, we need to consider how (3) can be obtained, especially from the given formulae such as (1) and (2). This gives rise to the need of the induction of (3). Similar to the case in first-order logic [17, 18, 13], this may be solved by generating the least general generalisation (LGG) of terms in this language.

- **Efficiency.** To achieve practical efficiency, deduction and induction in this language must be easy to compute. In specific, corresponding to first order logic the lower bound of generalisations and unifications of terms must be unique as both are unique for the language of first order terms.

Following Robinson, unification has become the basis of resolution-based logic deduction. Huet [6] found a semi-deterministic unification algorithm for higher-order terms. Inspired by the success of the logic programming language Prolog, λProlog [14] has been developed as a full higher-order extension of Prolog. It has been successfully applied in writing program transformers, theorem provers and a number of other areas [15, 2].

Similar to MGU, the least general generalisation (LGG) of terms [17, 13] plays an important role in the emerging field of *inductive logic programming*, which is evolved from logic programming and conventional machine learning [12]. However, as the field matures the need for various higher-order notions has arisen in order to guide the induction of Horn clauses from facts (ground atoms). These include rule templates [20, 8, 4], predicate determinations and modes [13], variable types [9, 19] and predicate commutativity [9].

At present, higher order terms already have many applications in machine learning: these include inductive learning [13], analogical reasoning [5], constructive induction [20] and model-driven induction [8]. It is argued that higher-order terms provide elegant expressions for many problems. However, the higher order formalisms used in research to date are *ad hoc*. In particular they deliberately avoid using λ-calculus. They therefore lose the expressive power of higher-order logic.

In a higher order logic, a term E is more *general* than another term F, or F is more specific than E, if and only if there is a substitution θ such that $E\theta$ is λ-convertible to F. This is denoted by $E \geq_\theta F$ or $F \leq_\theta E$. The relation $\geq_\theta$ is transitive, nonsymmetric and reflexive and thus defines a partial ordering over terms. $E =_\theta F$, or E is θ-equal to F if and only if $E \geq_\theta F$ and $F \geq_\theta E$. When $E \geq_\theta F$ but $E \neq_\theta F$, we say $E >_\theta F$ or $F <_\theta E$.

A term E is a *common generalisation* of a set of terms $\mathcal{T}$ if and only if E is more general than each of the terms in $\mathcal{T}$. A term E is a *least general generalisation* of a finite set of terms $\mathcal{T}$ if and only if: 1) E is a common generalisation of $\mathcal{T}$; and 2) for any $F <_\theta E$, F is not a common generalisation of $\mathcal{T}$.

It is shown in [3] that in general there are multiple or an infinite number of solutions for the LGG of two higher-order terms that are expressed in full $\alpha\beta\eta$ λ-calculus (see Appendix A). In order to compute LGGs efficiently, it is desirable to have a restricted λ calculus such that MGU and LGG are unique in the condition of not sacrificing effectiveness. It is, indeed, only the formula (3) in Section 1 that we are most interested in. To achieve this we will have to place some additional conditions on λ-calculus and hopefully we can find a manageable subclass of the general λ-calculus. One such restriction is L_λ [11]. L_λ [11, 16] is a restricted subset of λProlog which has unique MGU and LGG. But it is ill-equipped to express many problems. In particular it cannot represent recursions in its terms which we shall see in Section 6.

3 M_λ: a restricted higher order language

Readers unfamiliar with simple (typed) λ-calculus may go to the Appendix A for the relevant information on λ-calculus. Throughout this paper, the terminology is similar to that used in logic programming, especially Prolog. Variables are denoted by upper case letters U to Z; formulae and terms by other upper case letters; constants by lower case letters; types by Greek letters α, β and γ; and substitutions by Greek letters θ, δ and etc., when confusion does not arise. The constant "$\leftarrow$" is often used as an infix operator for convenience of understanding. In a few occasions we will use calligraphical upper-case letters to denote sets. Suitable super- or sub-scripts may also be used.

A L_λ [11] term contains only free variables that are applied to bound variables. For example, the formula

$$\lambda X.\lambda Y.\lambda Z.(P(X,Y) \leftarrow P(X,Z), P(Z,Y)). \tag{4}$$

is a L_λ term. The free variable P only has arguments that are bound variables in the formula. P is said to be applied to X and Y and λX is called an abstraction from $\lambda Y.\lambda Z.(P(X,Y) \leftarrow P(X,Z), P(Z,Y))$. A variable in an abstraction is called a bound variable, otherwise it is a free variable. We will also say a variable is free to a term (or subterm) if it is not bound by that term (or subterm). When there are many successive abstraction variables in a formula such as in (4) we may write "λXYZ".

The term in (4) is closely related to (3) as the quantifier "$\forall X.F$" is an abbreviation for "$\Pi(\lambda X.F)$" and Π is a constant expressing universal quantification (and similarly "$\exists X.F$" is for "$\Sigma(\lambda X.F)$").

Though MGU and LGG is unique in L_λ, it is too constrained to express simple terms that contain recursion. Practical examples of the restrictions of L_λ can be seen in Section 6. But now let us look at a simple modification of the above term:

$$\lambda XYZ.(P(s(X),Y) \leftarrow P(X,Z), P(Z,s(Y))). \tag{5}$$

Because it contains a constant s in the arguments of P, (5) is not a L_λ term which is important in many logic programming problems (see Section 6). We shall introduce a more powerful calculus called M_λ. In this calculus, a subterm such as $s(X)$, called an object term, is explicitly allowed.

An *object term* consists of externally bound variables and constants of distinct types from any bound variables without abstraction. A variable X in a term F is ***externally bound*** if F is a subterm of some term E and X is bound by an abstraction in E outside of F. The use of object terms gives rise to the necessity of the other two extensions in M_λ.

M_λ is a restricted typed λ-calculus. In M_λ: 1) It is allowed to perform α, β, δ_0 and η conversions on terms. The δ_0 conversion rule is described in Appendix B; 2) Any free variables in M_λ terms are only applied to arguments that are object terms; 3) It contains at

least constants χ and ψ such that for any object term of M_λ, E, $\chi(E) = E_2$ and $\psi(E) = E_1$ if $E = (E_1 E_2)$, and $\psi(E) = \chi(E) = E$ otherwise. (Both are undefined for non-object terms).

Clearly M_λ is an extension of the L_λ language [11], in which the δ_0 rule is not permitted and the arguments of free variable functions can only be externally bound variables. χ and ψ are analytical selectors in an analytical syntax as defined by McCarthy. Fortunately many programming languages, such as Lisp and Prolog, have these functions. In M_λ these are further restricted to the application to object terms, which do not contain abstractions as defined in this paper.

4 M_λ normal and nonredundant terms

A term is M_λ *normal* if and only if it is in $\alpha\beta\delta_0\eta$ normal form and contains no irreducible δ_0 expressions. We will be interested in LGGs and MGIs that are M_λ normal. The following definition is adopted from that in Section 2. E is the M_λ *normal LGG* of a set of M_λ normal terms $\mathcal{T}$, if and only if: 1) E is M_λ normal and is also a common generalisation of $\mathcal{T}$; and 2) F is not a common generalisation of $\mathcal{T}$ for any M_λ normal term $F <_\theta E$.

For M_λ normal term $S = p(U, U(a))$, the substitution containing δ_0 conversions such as $\theta = \{U/\lambda X.if\ X = a\ then\ E_1\ else\ E_2\}$ is not applicable. If applied it results in $S\theta = p((\lambda X.if\ X = a\ then\ E_1\ else\ E_2), E_1)$, which cannot be reduced further because X in $X = a$ is a free variable to X. This term is not a M_λ normal term.

Let E be a M_λ term and E' be a θ-equal term to E. A subterm F of E will have some trace F', so to speak, in E'. F' is called the *residual* of F. A subterm F of E is *redundant* if $E'' = E' - \{F'\}$ and $E'' =_\theta E$, i.e the residual of F can be removed and still maintain θ-equality.

Clearly, any redundancy can only happen in terms with free variable functors. If E is a subterm of F with free variable functor and no other subterm of F has the same functor, then any term containing only constants and repetitive bound variables in E is redundant: Because E is the only subterm that has the free variable as its functor, we are free to devise various substitutions to decompose E into its subterms and then remove all constants and repetitive bound variables from it. The operation to remove redundant subterms is called a reduction (not to be confused with α, β, δ_0 and η reductions of λ calculus). After simple reductions, constants will appear only in subterms that appear in multiple places of a terms. [3] describes an algorithm that can reduce a M_λ to its nonredundant form. In the rest of the paper we refer terms to M_λ normal, nonredundant terms and their LGGs to M_λ normal, nonredundant LGGs.

We need also to consider unification, which is the dual of LGG. E is the *common instance* of a set of M_λ terms $\mathcal{T}$ if E is more specific than each element of $\mathcal{T}$. E is the most general instance (MGI) of a set of M_λ terms $\mathcal{T}$ if and only if: 1) E is the common instance of $\mathcal{T}$; and 2) F is not the common instance of $\mathcal{T}$ for any $F >_\theta E$. The substitution θ for which $E_i\theta = E, \forall E_i \in \mathcal{T}$ is called the most general unifier (MGU). M_λ normal MGU is similarly defined except that E, F and $E_i \in \mathcal{T}$ are all M_λ normal terms. It is proved in [3], in normal and nonredundant form the MGU and LGG of M_λ terms are unique. The unification algorithm is also given in [3].

5 Implementation

LGG and unification in M_λ is implemented in Prolog, and is called HOLGG. The LGG algorithm has two parses. The first parse is ELGG which collects all the multiple appearances of subterms into a set of triples $\Gamma = \{(\tilde{X}, S_1, T_1)\}$. Then Γ is sorted to obtain $\Gamma = \bigcup_{i=1}^{m} \Gamma_i$ for which $\Gamma_i = \{(\tilde{X}, S_j, T_j)\}$ and each S_{j1} and S_{j2} in Γ_i have the same functor and so do T_{j1}

and T_{j2}. In the second parse, CLGG takes S, T and Γ_i $(i = 1, ..., m)$ as input and produces F. CLGG also calls the algorithm VLGG.

MLGG: anti-unification (LGG) algorithm for M_λ normal terms.
INPUT: two M_λ normal terms S and T of the same type;
OUTPUT: a M_λ normal term $F = MLGG(S, T)$.
1. $\Gamma = ELGG(S, T, \emptyset)$;
2. Sort Γ such that $\Gamma = \bigcup_{i=1}^{m} \Gamma_i$ for which $\Gamma_i = \{\langle \tilde{X}, S_i(S_{j1}, ..., S_{jn}), T_i(T_{j1}, ..., s_{jk})\rangle\}$ for $i = 1, ..., m$;
3. $F = CLGG(S, T, \Gamma, \emptyset)$.

ELGG
INPUT: S, T and the binding variables $\tilde{X}$;
OUTPUT: $\Gamma = \{\langle F_p, S_p, T_p\rangle\}$ where F_p, S_p have different functors.
1. If $S = \lambda\tilde{Y}.S_1$ and $T = \lambda\tilde{Y}.T_1$, then $\Gamma = ELGG(S_1, T_1, \tilde{X}\tilde{Y})$;
2. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 = T_0 = C$, where C is a constant or $C \in \tilde{X}$, then $\Gamma = \bigcup_{i=1}^{n} ELGG(S_i, T_i, \tilde{X})$;
3. If $S_0 \neq T_0$, then $\Gamma = \{\langle \tilde{X}, S, T\rangle\}$.

CLGG
INPUT: S, T, the sorted Γ and the binding variables $\tilde{X}$;
OUTPUT: F is the M_λ normal LGG of S and T.
1. If $S = \lambda\tilde{Y}.S_1$ and $T = \lambda\tilde{Y}.T_1$, $F = \lambda\tilde{Y}.CLGG(S_1, T_1, \Gamma, \tilde{X}\tilde{Y})$;
2. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 = T_0$, then $F = C(F_1, ..., F_n)$ where $F_i = CLGG(S_i, T_i, \Gamma, \tilde{X})$;
3. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 \neq T_0$, and $\langle \tilde{X}, S, T\rangle$ is in Γ_i and there exist F_i $(i = 1, ..., l)$,
   3.1. If S_0 and T_0 are constants or free variables, then $F = V_{S_0,T_0}(F_1, F_2, ..., F_l)$;
   3.2. If $S_0 \in \tilde{X}$ and T_0 is a constant or a free variable, then $F = V_{S_0,T_0}(S_0, F_1, F_2, ..., F_l)$;
   3.3. If S_0 is a constant or a free variable and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(T_0, F_1, F_2, ..., F_l)$;
   3.4. If $S_0 \in \tilde{X}$ and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(S_0, T_0, F_1, F_2, ..., F_l)$;
   where $F_S = VLGG(S_{i1}, VLGG(..., VLGG(S_{i,j-1}, S_{ij}, \tilde{X}), ...), \tilde{X})$
   and $F_T = VLGG(T_{i1}, VLGG(..., VLGG(T_{i,j-1}, T_{ij}, \tilde{X}), ...), \tilde{X})$
   such that $S = (...((\lambda\tilde{X}.F_S)F_1)...F_l)$ and $T = (...((\lambda\tilde{X}.F_T)F_1)...F_l)$;
4. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 \neq T_0$,
   4.1. If S_0 and T_0 are constants or free variables, then $F = V_{S_0,T_0}(X_1, X_2, ..., X_l)$;
   4.2. If $S_0 \in \tilde{X}$ and T_0 is a constant or a free variable, then $F = V_{S_0,T_0}(S_0, X_1, X_2, ..., X_l)$;
   4.3. If S_0 is a constant or a free variable and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(T_0, X_1, X_2, ..., X_l)$;
   4.4. If $S_0 \in \tilde{X}$ and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(S_0, T_0, X_1, X_2, ..., X_l)$;
   where $X_i \in \tilde{X}$ $(i = 1, ..., l)$ are bound variables in S and T.

VLGG: Variablisation of terms in M_λ.
INPUT: two terms S and T and binding variables $\tilde{X}$;
OUTPUT: $F = VLGG(S, T, \tilde{X})$.
1. If $S = \lambda\tilde{Y}.S_1$ and $T = \lambda\tilde{Y}.T_1$, then $F = \lambda\tilde{Y}.VLGG(S_1, T_1, \tilde{X}\tilde{Y})$;
2. If $S = T = C$ and C is a constant or $C \in \tilde{X}$, then $F = C$;

3. If $S = C(S_1, ..., S_n)$ and $T = C(T_1, ..., T_n)$ and C is a constant or $C \in \tilde{X}$, then $F = C(VLGG(S_1, T_1, \tilde{X}), ..., VLGG(S_n, T_n, \tilde{X}))$;
4. If S, T contain variables in $\tilde{X}$, then fail otherwise $F = V_{S,T}$ for a variable named by S and T.

6 Applications

LGG has played an important role in inductive logic programming in first-order logic. The following examples show the applications of LGG to acquire higher-order clause templates from given first-order clauses.

Example 1. Given the first-order facts:

less_than(0,2), less_than(1,3), less_than(2,4), less_than(3,5), ...
less_than(0,3), less_than(1,4), less_than(2,5), less_than(3,6), ...
less_than(0,1), less_than(1,2), less_than(2,3), less_than(3,4), less_than(1,5), ...

we are able to obtain the following, using an algorithm such as Golem [13]:

$$\forall XYZ.less\_than(X,Y) \leftarrow less\_than(X,Z), less\_than(Z,Y),$$

and similarly

$$\forall XYZ.ancestor(X,Y) \leftarrow ancestor(X,Z), ancestor(Z,Y),$$

In both clauses X, Y and Z are universally quantified. Note $\forall X.F$ is the abbreviation of $\Pi(\lambda X.F)$ and Π expresses universal quantification. Their higher-order LGG is

$$\forall XYZ.P(X,Y) \leftarrow P(X,Z), P(Y,Y),$$

where P is a free variable and "$\leftarrow$" is an infix constant. P may then become universally quantified as this generalisation is accepted.

Though L_λ can still be used to express the higher-order term in Example 1, the clause in the following example, which contains recursion and represents a majority of problems that we are interested in, cannot be expressed within L_λ. This is because L_λ forbids the existence of constants in the arguments of free variables.

Example 2. Given the first-order clauses with X, Y, Z and W universally quantified,

$$\forall XYZW.reverse(cons(X,Y),Z) \leftarrow reverse(Y,W), append(W, cons(X, nil), Z),$$
$$\forall XYZW.insert\_sort(cons(X,Y),Z) \leftarrow insert\_sort(Y,W), insert(X,W,Z),$$

we can obtain

$$\forall XYZW.P(cons(X,Y),Z) \leftarrow P(Y,W), Q(W,X,Z),$$

where "*reverse*" is a version often referred to as "naive reverse", P and Q are free variables, and "*cons*" is a list processing function.

One may observe that the first-order facts seem to be the objects in the induction of first-order clauses. These clauses then become objects that characterise the properties of *higher-order objects* — in this particular case it is second order predicate constants such as "*reverse*" and "*insert*". At this "order", we are mainly concerned with the properties of the predicates, the first order objects will be universally quantified, and they are "taken for granted" when studying objects that may apply on them. After this, the higher-order objects may become universally quantified.

If we extend this scenario further, we can imagine that through progressive quantification clauses of successive orders can be induced that characterise objects of higher order objects.

Another application area is the discovery of program transformation rules. Huet and Lang [7] discussed methods for program transformation of recursive computations into iterative ones. A set of second order clause templates for transformation were suggested and they are applied through second order unification to produce more efficient programs based on the Darlington and Burstall [1] method. As they remarked, the opposite problem with regarding to the discovery of such templates is a difficult task. Few templates are known and no automatic methods exist for performing such discovery.

Example 3. The higher-order logic clause in Example 1, though interesting, is computationally inefficient. To satisfy the first predicate $P(X,Y)$, it needs to nondeterministically satisfy $P(X,Z)$ and then $P(Z,Y)$. More efficient programs for *ancestor* and *less_than* are respectively:

$$\forall XYZ.less\_than(X,Y) \leftarrow successor(X,Z), less\_than(Z,X),$$
$$\forall XYZ.ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,X),$$

where $successor(X,Z)$ expresses that Z is the successor of X in Peano's formalism, and $parent(X,Z)$ states that Z is the parent of X. Both are computational more efficient. Thus possible program transformations are:

$$(\forall XYZ.less\_than(X,Y) \leftarrow successor(X,Z), less\_than(Z,X)))$$
$$\Longleftarrow$$
$$(\forall XYZ.less\_than(X,Y) \leftarrow less\_than(X,Z), less\_than(X,Y)),$$

$$(\forall XYZ.ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,X)))$$
$$\Longleftarrow$$
$$(\forall XYZ.ancestor(X,Y) \leftarrow ancestor(X,Z), ancestor(X,Y)).$$

The LGG produces a program transformation template, though the conditions for the transformation are omitted.

$$(\forall XYZ.P(X,Y) \leftarrow Q(X,Z), P(Z,X)))$$
$$\Longleftarrow$$
$$(\forall XYZ.P(X,Y) \leftarrow P(X,Z), P(X,Y)).$$

In fact, such a template is applicable when Q is a special case of P (i.e. P by one).

Example 4. The recursive list reverse program is described in 3. A more efficient iterative (tail recursive) version of it is:

$$\forall XYZWU.reverse1(cons(X,Y),Z,W) \leftarrow append([X],Z,U), reverse1(Y,U,W))$$

where "*reverse*1" contains an accumulator Z. When "*reverse*1" starts with "$reverse1(List, nil, ReversedList)$" and is terminated by "$reverse1(nil, ReversedList, ReversedList)$", it yields the reversed list. Thus we have a transformation

$$(\forall Z.reverse1(nill,Z,Z)\ \&$$
$$\forall XYZWU.reverse1(cons(X,Y),Z,W) \leftarrow append([X],Z,U), reverse1(Y,U,W))$$
$$\Longleftarrow$$
$$(reverse(nil,nil)\ \&$$
$$\forall XYZW.reverse(cons(X,Y),Z) \leftarrow reverse(Y,W), append(W,cons(X,nil),Z)).$$

We also know another transformation which concerns with the addition of the elements in a list:

$$(\forall Z.sumlist1(nil,Z,Z)\ \&$$
$$\forall XYZWU.sumlist1(cons(X,Y),Z,W) \leftarrow add(X,Z,U), sumlist1(Y,U,W))$$
$$\Longleftarrow$$
$$(sumlist(nil,0)\ \&$$
$$\forall XYZW.sumlist(cons(X,Y),Z) \leftarrow sumlist(Y,W), add(W,X,Z)),$$

where "*plus*" is a function that returns the addition of two numbers. When started with "$sumlist1(List, 0, SumOfList)$" and terminated by "$sumlist1(nil, SumOfList, SumOfList)$", the iterative computation also returns the sum of the elements in the list. The LGG of the two is:

$$
\begin{array}{l}
(\forall Z.P1(nil, Z, Z)\ \&\\
\forall XYZWU.P1(cons(X,Y),Z,W) \leftarrow Q(Z,X,U), P1(Y,U,W))\\
\quad \Longleftarrow\\
P(nil, V)\ \&\\
\forall XYZW.P(cons(X,Y),Z) \leftarrow P(Y,W), Q(W,X,Z))
\end{array}
$$

with free variables P, Q, $P1$ and V.

This is an alternative expression of McCarthy's transformation [10]. For the sake of convenience we have omitted the conditions for this transformation to apply. It is in fact that, among others, V must be the lower bound element of the appropriate type and Q be a transitive and communicative function. This problem can be addressed by relative least general generalisation (RLGG) that will be discussed briefly in Section 7. However, its detail is beyond the scope of this paper.

The other potential application areas are analogical reasoning and the automatic acquisition of grammar rules from example sentences and the generalisation of proofs. However we will not discuss them in this paper.

7 Discussion and future research directions

Recently, inductive logic programming [12] has witnessed a growing trend in utilising higher-order (or meta-level) logical notions in existing ILP framework. This is motivated, in part, by the need to develop more effective and efficient ILP methods. These notions are often adopted as declarative biases in many forms including functional constraints on the predicates in clauses and templates for the clauses being induced. However current methods lack a coherent framework for accommodating these notions.

Our future research work is concerned with expanding the current (first order)ILP framework, in which higher-order inductive inference may be described as the discovery of a hypothesis H from examples and background knowledge such that:

$$
\begin{array}{l}
M \wedge B \wedge H \vdash E^+\\
M \wedge B \wedge H \nvdash E^-
\end{array}
\qquad (6)
$$

if $M \wedge B \nvdash E^+$. M in relation (6) represents a set of higher-order (λ Prolog) clauses. The hypotheses in H now can be either first-order or higher-order clauses. B is the background knowledge, E^+ and E^- are respectively the set of positive and negative examples. Corresponding to existing ILP theory, it is necessary to develop methods for generalisation in higher-order logic. In doing so we hope to achieve two aims: a) to develop more efficient methods for inducing first-order clauses, and b) to induce higher-order clauses such that they can aid induction of both classes of logic programs.

From the experience of ILP in first-order logic, in the next step we need to study the relative least general generalisation (RLGG) for higher-order logic programs in the presence of background knowledge. Similar to the first-order case [13], we may have to deal with a restricted logical model of the background knowledge. We also need to investigate the computability of such a model of the background knowledge. If this is successful the results will have implications in ILP and to the discovery of automatic program transformation techniques.

Acknowledgements. The authors are grateful to the ILP group at the Turing Institute.

We are also thankful to Dale Miller and Frank Pfenning for providing general information in higher-order logic programming.

A Syntax of λ-calculus terms

A term in (simply typed) λ-calculus can be one of the following:

1. **Atom.** A variable or constant (of type α) is a term (of type α);
2. **Application.** An application of F (of type $\beta \to \alpha$) to E (of type β) is a term $(F\ E)$ (of type α);
3. **Abstraction.** The abstraction of a term F (of type α) on a variable X (of type β) is the term $(\lambda X.F)$ (of type $\beta \to \alpha$) that binds X in the scope F.

The type of a term F is denoted by $\tau(F)$. One may verify that (1), (2) and (3) in Section 1 are terms, and $\tau(P) = (\alpha, \alpha \to \beta)$ and $\tau(\leftarrow) = (\beta, \beta, \beta \to \gamma)$, assuming that $\tau(X) = \tau(Y) = \tau(Z) = \alpha$. Types are not essential in some results. They are polymorphic when used.

We denote $(...((F\ E_1)\ E_2)\ ...E_n)$ by the expression $F(E_1, E_2, ..., E_n)$ if F is an atom, and its type $(\alpha_1 \to (\alpha_2 \to ...(\alpha_n \to \beta)...))$ by $(\alpha_1, \alpha_2, ..., \alpha_n \to \beta)$, where $\tau(E_i) = \alpha_i$. F is called the functor, E_i $(i = 1, 2, ..., n)$ the arguments and n the arity of F. We also abbreviate $(\lambda X_1.(\lambda X_2...(\lambda X_n.F)...))$ to $(\lambda X_1 X_2...X_n.F)$ and $X_1 X_2...X_n$ to $\tilde{X}$.

The *order* of an atom is the depth of the nesting of parentheses in its type + 1. The order of a term is the highest order of its atoms. The order of X, Y and Z in (3) is 1, and the order of P is 2. The order of (3) is therefore 2.

A.1 λ-conversions

Let $F\{X/E\}$ be the operation of substitution that replaces each occurrence of X in F by E. X is free in F if it does not occur in the scope of an abstraction in F that binds X. E is free for X in F if E does not appear in the scope of an abstraction in F that binds X. These two conditions are used to avoid possible name clashes in the following definition of λ conversions.

A *substitution* is a set of ordered pairs $\theta = \{X_i/E_i \mid i = 1, ..., n\}$, where X_i are distinct variables and each E_i is a term of the same type as X_i. The application of θ to a term F is denoted by $F\theta$ and it is the term $(...((\lambda\tilde{X}.F)E_1)E_2)...)E_n)$. Intuitively for each $X_i (i = 1, ..., n)$ $F\theta$ is the results of replacing X at each place in F by the subterm E_i. The composition of substitutions, denoted by $\theta \cdot \delta$, is the same as defined in first-order logic (sometimes we may omit the dot).

The *conversion rules* of the general λ-calculus is

1. α-**rule.** $\lambda X.F$ is convertible to $\lambda Y.(F\{X/Y\})$ if Y is free for X in F, and vice versa;
2. β-**rule.** $(\lambda X.F)E$ is convertible to $F\{X/E\}$ if E is free for X in F, and vice versa;
3. η-**rule.** $\lambda X.FX$ is convertible to F if X is not free in F, and vice versa.

The *convertibility*[1] of two terms is an equivalence relation. The rules can be carried out in both left-to-right and right-to-left directions. λ-Convertible terms are considered to be equal to each other. An application of a conversion rule is called a α (or β and etc.) *reduction* when applied in the left-to-right direction and an *expansion* in the opposite direction. A λ term that cannot be reduced by rules of any kind is said to be in *normal form*.

Church and Rosser proved that the normal form of a term is unique. For any convertible λ terms E and F, there is a term S in normal form such that E and F can be reduced to S. They also proved the normal form of a term can be obtained by attacking the leftmost

[1] Note often an α conversion is used prior to a β conversion to change the names of bound variables.

reduction on one-by-one basis until no reduction is possible. We should bear in mind at this point, apart from α, β and η rules, there are other conversion rules that also maintain Church and Rosser properties such as the δ and δ_0 rules that will be introduced in Appendix B.

B Extension to δ_0 conversion

We will first consider a conversion rule of δ_0, which is a variant of the δ conversion rule studied by Church in an extension to $\alpha\beta\eta$ λ-calculus. The δ_0 conversion rule is defined as follows. If E and F are in $\beta\delta_0\eta$ normal form without variables free in E and F:

1. $\delta_0 EF = \lambda XY.X$ if E is α-convertible to F;
2. $\delta_0 EF = \lambda XY.Y$ otherwise.

$\lambda XY.X$ is known as "true" and $\lambda XY.Y$ as "false" in the standard λ-calculus (In Church's system, truth is expressed by $\lambda XY.XY$ — the combinatory number 1 – and falsity by $\lambda XY.X(XY)$ — the number 2. This is an arbitrary choice).

δ_0 rule is similar to a weaker form of the equality theory introduced in modern deductive logic. Similarly to $\alpha\beta\eta$ λ-calculus, the application of the δ_0 rule from left-to-right is called a δ_0 reduction. It is shown that δ_0 reductions maintain the Church-Rosser properties of λ-calculus when the reduction expressions are carried over from the original (perhaps not in $\beta\delta_0\eta$ normal form) terms. We shall restrict ourselves to such a λ-calculus.

The term $\lambda XY.X$ selects the first argument of X and Y, whereas $\lambda XY.Y$ selects the second. Using the δ_0 rule we can construct a term: $F = (\lambda X.((\delta_0 XN)E_1)E_2)$. When F is applied to a term M, we obtain $FM = (((\delta_0 MN)E_1)E_2)$ which corresponds to "if M is α convertible to N, then E_1 else E_2", where $\delta_0 MN$ is the δ_0 reduction expression and E_1 and E_2 are the terms applied upon by this reduction. M, N, E_1 and E_2 are in $\beta\delta_0\eta$ normal form. M and N contain no variables free in them. We shall denote F by "$\lambda X.if\ X = N\ then\ E_1\ else\ E_2$". Clearly, $(\lambda X.E_1)$ is convertible to $(\lambda X.if\ X = N\ then\ E_1\ else\ E_2)$ if E_1 is α-convertible to E_2.

References

[1] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), 1977.

[2] S. Dietzen and F. Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. In B. Spatz, editor, *Proceedings of the sixth international workshop on machine learning. Cornell University, Ithaca, New York*, pages 447 – 449. San Mateo, CA: Morgan Kaufmann, June 1989.

[3] Cao Feng and Stephen Muggleton. Least general generalisation in higher order logic. TIRM, The Turing Institute, George House, 36 NOrth Hanover St, Glasgow, UK, 1991. Also submitted to the Journal of Symbolic Logic.

[4] M. Harao. Analogical reasoning based on higher order unification. In *First International Conference on Algorithmic Learning Theory*, pages 151–163, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[5] Masateru Harao. Analogical reasoning based on higher order unification. In *First International Conference on Algorithmic Learning Theory*, pages 151–163, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[6] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[7] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

[8] J. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntact ic and task orieted models. In S. Muggleton, editor, *Proceedings of the First International Workshop on Inductive Logic Programming*, Viana de Castelo, Portugal, to appear. Academic Press.

[9] N. Lavrac and S. Dzeroski. Learning nonrecursive definitions of relations with linus. In Y. Kodratoff, editor, *EWSL '91: machine learning: proceedings of the European working session on learning*, pages 265–281, Porto, Portugal., 1991. Berlin: Springer-Verlag.

[10] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (part 1). *Communications of the Association of Computational Machinery*, 3:184–204, 1960.

[11] D. Miller. A logic programming language with λ-abstraction, function variables and simple unification. In P. Schroeder-Heister, editor, *Extensions of logic programming*, pages 237–258. Sprinter-Verlag LNCS, 1990.

[12] S. Muggleton. Inductive logic programming. In *First International Conference on Algorithmic Learning Theory*, volume Also in New Generation Computing, 1991, Vol 8, pages 42–61, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[13] S. Muggleton and C. Feng. Efficient induction of logic programs. In *First International Conference on Algorithmic Learning Theory*, pages 369–381, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[14] G. Nadathur and D. Miller. Higher-order horn clauses. *Journal of the Association for Computing Machinery*, 37(4):777–814, 1990.

[15] L. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258, 1986.

[16] F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74 – 85. IEEE, July 1991.

[17] G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Elsevier North-Holland, New York, 1970.

[18] G.D. Plotkin. A further note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 101–124. Elsevier North-Holland, New York, 1971.

[19] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[20] L. De Raedt and M. Bruyonooghe. Constructive induction by analogy: a new method to learn how to learn? In K. Morik, editor, *EWSL'89: proceedings of the fourth european working session on learning*, Montpellier, France, 1990. London: Pitman.

[21] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–151. Elsevier North-Holland, New York, 1970.

ABSTRACTION BASED ANALOGICAL REASONING FOR NATURAL DEDUCTION PROOF

Masateru HARAO

Department of Artificial Intelligence
Kyushu Institute of Technology, Iizuka ,820,JAPAN
harao@dumbo.ai.kyutech.ac.jp

Abstraction

This paper presents a framework of obtaining a natural deduction proof of a given logic formulae based on similarity among formulas under the assumption that similar formulas have similar solutions. A formulae to which a proof has already been given is called a guiding problem. From a guiding problem, a schema which is applicable to a class of similar formulas is constructed by abstraction. A schema acts as a specification of proofs and any object formulae having the same type to a schema can be obtained according to the typed proof structure. The analogical reasoning based on this idea is formalized using typed language in the framework of higher order logic. Finally, we show that this analogical reasoning procedure can be realized based on higher order unification within the computable scope.

1.Introduction

In order to realize an intelligent system on machine, one of the most important problem is to introduce a reasoning mechanism which break through the wall of present deductive theorem proving paradigm. For such purpose many reasoning systems such as abductive reasoning, inductive reasoning, non-monotonic reasoning and so on have been studied. The analogical reasoning is a mechanism to reason by finding certain similarity with some already known problem, and is considered as a most essential mechanism which supports the creative thinking of human beings. It has been proposed several kinds of models for analogical reasoning systems[6,10,13]. Among them , the reasoning system based on the generalized knowledge produced from already known formulae by abstraction is called the abstraction based analogy[5,14].There are some papers applied the analogical reasoning to the scientific problems[2,3,4,8,16]. In this paper,an abstraction based analogical reasoning system for LK proving will be formalized focussing the following aspects.

(1) What is the suitable formal system for formalizing the abstraction based analogical reasoning ?
(2)How to realize the similarity among objects?
(3) How to realize the abstraction process?
(4)How to mechanize the analogical reasoning processes?

We will formalize this framework as illustrated in Fig.1. Where,the proof of a guiding problem is obtained probably by trial and error. This proof structure is abstracted as proof schema. Then a new formulae ? is proved using the similarity between ? and some guiding problem. This similarity check is done by the unifiability with schema constructed from the guiding problem. The proof of ? is derived by the substitution obtained from the unification. By this analogical reasoning process,we can

expect to reduce the nondeterministic aspects from the processing and to realize certain non-deductive reasoning

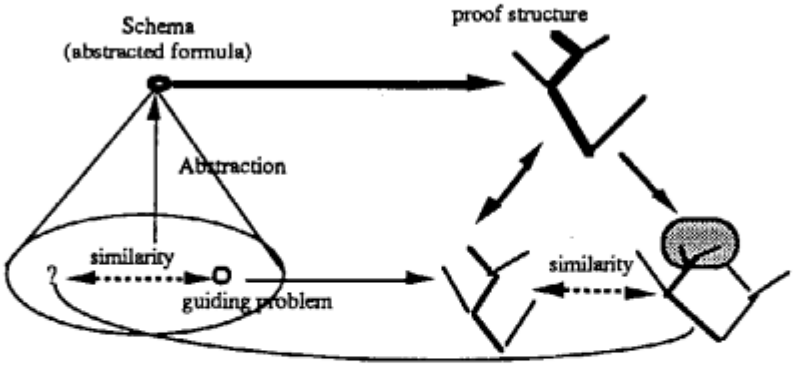


Fig.1 Proof construction by analogy

In Section 2, the basic properties and proof theory of natural deduction system LK is surveyed.The analogical reasoning system proposed uses the abstracted knowledge called schema: In Section 3, the schema construction from guiding problems is discussed. In Section 4, a unification based analogical reasoning procedure is proposed. Some examples are also given and some remarks on these approach are discussed as the conclusion in Section 5.

2 LK system and natural deduction proof

2.1 LK system

The logical formulae is defined inductively as follows.

$$\phi ::= A \mid \sim\phi \mid \phi\vee\phi \mid \phi\wedge\phi \mid \phi\supset\phi \mid \forall x.\phi \mid \exists x.\phi$$

,where A is an atom. By $\Gamma\rightarrow\Theta$, we denote a sequent. The LK system is a logic system which consists of the following inference rules:

$$\text{(or-L)}\ \frac{A,\Gamma\rightarrow\Theta \quad B,\Gamma\rightarrow\Theta}{A\vee B,\Gamma\rightarrow\Theta} \qquad \text{(or-R)}\ \frac{\Gamma\rightarrow\Theta,A,B}{\Gamma\rightarrow\Theta,A\vee B} \qquad \text{(and-L)}\ \frac{A,B,\Gamma\rightarrow\Theta}{A\wedge B,\Gamma\rightarrow\Theta} \qquad \text{(and-R)}\ \frac{\Gamma\rightarrow A \quad \Gamma\rightarrow B}{\Gamma\rightarrow A\wedge B}$$

$$\text{(imp-L)}\ \frac{\Gamma\rightarrow\Theta,A \quad B,\Delta\rightarrow\Lambda}{\Gamma,A\supset B,\Delta\rightarrow\Theta,\Lambda} \qquad \text{(imp-R)}\ \frac{\Gamma,A\rightarrow\Theta,B}{\Gamma\rightarrow\Theta,A\supset B} \qquad \text{(all-L)}\ \frac{A[x:=t],\Gamma\rightarrow\Theta}{\forall xA(x),\Gamma\rightarrow\Theta} \qquad \text{(all-R)}\ \frac{\Gamma\rightarrow\Theta,A[x:=y]}{\Gamma\rightarrow\Theta,\forall xA(x)}$$

$$\text{(some-L)}\ \frac{A[x:=y],\Gamma\rightarrow\Theta}{\exists xA(x),\Gamma\rightarrow\Theta} \qquad \text{(some-R)}\ \frac{\Gamma\rightarrow\Theta,A[x:=t]}{\Gamma\rightarrow\Theta,\exists xA(x)} \qquad \text{(thin-L)}\ \frac{\Gamma\rightarrow\Theta}{A,\Gamma\rightarrow\Theta} \qquad \text{(thin-R)}\ \frac{\Gamma\rightarrow\Theta}{\Gamma\rightarrow\Theta,A}$$

$$\text{(not-L)}\ \frac{\Gamma\rightarrow\Theta,A}{\sim A,\Gamma\rightarrow\Theta} \qquad \text{(or-R)}\ \frac{A,\Gamma\rightarrow\Theta}{\Gamma\rightarrow\sim A,\Theta} \qquad \text{(Cut)}\ \frac{\Gamma\rightarrow\Theta,A \quad A,\Delta\rightarrow\Lambda}{\Gamma,\Delta\rightarrow\Theta,\Lambda}$$

A sequent A→A is trivially true and is called an axiom. A LK natural deduction proof is produced by applying the inference rules in nondeterministic, and can be represented by a derivation tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{q(a)\rightarrow q(a)}{p(a)\rightarrow p(a) \quad q(a)\rightarrow\exists xq(x)}}{p(a),p(a)\supset q(a)\rightarrow\exists xq(x)}}{p(a),\forall x\,(p(x)\supset q(x))\rightarrow\exists xq(x)} \qquad \dfrac{\dfrac{q(b)\rightarrow q(b)}{q(b)\rightarrow\exists xq(x)}}{q(b),\forall x\,(p(x)\supset q(x))\rightarrow\exists xq(x)}}{\begin{array}{l}p(a)\vee q(b),\ \forall x\,(p(x)\supset q(x))\rightarrow\exists xq(x)\\ (p(a)\vee q(b))\wedge\forall x\,(p(x)\supset q(x))\rightarrow\exists xq(x)\\ \rightarrow p(a)\vee q(b)\wedge\forall x\,(p(x)\supset q(x))\supset\exists xq(x)\end{array}}$$

Fig.2 A natural deduction proof.

A formulae is provable if there exists a proof tree whose root and leaves are labeled with the formulae and certain axioms respectively. In Fig.2, an example of LK proof of the following formulae is shown: $\rightarrow p(a) \vee q(b) \wedge \forall x\,(p(x) \supset q(x)) \supset \exists x q(x)$

2.2 Term representation of LK proof

Each inference rule is looked upon a function which maps from the assumptions given in the upper side of the rule to the conclusion given in the lower side of the rule. For example, the or-L rule corresponds to a function with the type $[A, \Gamma\rightarrow\Theta]\rightarrow[B,\Gamma\rightarrow\Theta]\rightarrow[A\vee B,\Gamma\rightarrow\Theta]$.

$$\frac{A, \Gamma\rightarrow\Theta \quad B,\Gamma\rightarrow\Theta}{A\vee B,\Gamma\rightarrow\Theta}\ (\text{or\_L})$$

This can be represented as the following term:

$$[A\vee B,\Gamma\rightarrow\Theta] = \text{or-L}([A, \Gamma\rightarrow\Theta],[B,\Gamma\rightarrow\Theta])$$

In the similar way, any LK proof is able to be represented by a term. In the followings, we denote the term representation of a proof for sequent $\Gamma\rightarrow\Theta$ as *proof*$(\Gamma\rightarrow\Theta)$, and call it as a *proof term*..

*Example 2.1*The proof term of the next formulae ϕ (the same one given in Fig.2)

$$\phi = \rightarrow(p(a)\vee q(b)) \wedge \forall x\,(p(x) \supset q(x)) \supset \exists x\, q(x)$$

is expressed as

proof(ϕ)= imp-R(and-L(or-L(all-L(imp-L(p(a)→p(a),some-R(q(a)→q(a))),thin-L(some-R(q(b)→q(b))))))

, and this term can be considered as a function from $[p(a)\rightarrow p(a)]$, $[q(a)\rightarrow q(a)]$ to $[(p(a)\vee q(b))\wedge\forall x\,(p(x) \supset q(x)) \supset \exists x\, q(x)]$, that is, this proof term has the following type.

$$[p(a)\rightarrow p(a)]\rightarrow[q(a)\rightarrow q(a)]\rightarrow[q(b)\rightarrow q(b)]\rightarrow[(p(a)\vee q(b))\wedge\forall x\,(p(x) \supset q(x)) \supset \exists x\, q(x)].$$

The labels of any leaves of completed proof are the axioms, that is, the sequents in the form $[A, \Gamma\rightarrow\Theta,A]$ or equivalently in the form $[A\rightarrow A]$. Thus,the proof term of each completed proof is the form such that *term*(...,*term*(A→A),...,*term*(B→B)). A proof in which some parts are not completed is called a *partial proof*. It is noted that the sequents at the leaves of partial proofs are not always axioms. For example, let us consider the proof given in Fig.3,which is a proof of Fig.2 in which the subproofs for the formulas $[p(a), \forall x(p(x)\supset q(x))\rightarrow\exists x q(x)]$ and $[q(b)\rightarrow\exists x q(x)]$ are not completed.

$$\dfrac{\dfrac{\dfrac{}{p(a), \forall x\,(p(x)\supset q(x))\rightarrow \exists x q(x)} \quad \dfrac{\dfrac{}{q(b)\rightarrow \exists x q(x)}\ (\text{thin\_L})}{q(b), \forall x\,(p(x)\supset q(x))\rightarrow \exists x q(x)}\ (\vee\text{\_L})}{p(a)\vee q(b), \forall x\,(p(x)\supset q(x))\rightarrow \exists x q(x)}\ (\wedge\text{\_L})}{(p(a)\vee q(b))\wedge\forall x\,(p(x)\supset q(x))\rightarrow\exists x q(x)}\ (\supset\text{\_R})$$

$$\rightarrow(p(a)\vee q(b))\wedge\forall x\,(p(x)\supset q(x))\supset\exists x q(x)$$

Fig.3 A partial proof of Fig.2

A term representation of this partial proof is given as follows:

$$\lambda X\,\lambda Y.\ \text{imp-R(and-L(or-L(X,thin-L(Y))))}$$

,where X and Y represent the partial proof for $[p(a),\forall x\,(p(x)\supset q(x))\rightarrow\exists x q(x)]$ and $[q(b)\rightarrow\exists x q(x)]$ respectively. Therefore, the partial proof given in Fig.3 implies the proof having the type

$$[p(a),\forall x(p(x)\supset q(x))\rightarrow\exists x q(x)]\rightarrow[\,q(b)\rightarrow\exists x q(x)]\rightarrow[\rightarrow(p(a)\vee q(b))\wedge\forall x\,(p(x)\supset q(x))\supset\exists x q(x)].$$

Example 2.2 We can consider each LK inference rule as a partial proof in which its assumptions are not completed. In the case of or-L inference rule, or-L is a function of the type $(A, \Gamma\rightarrow\Theta)\rightarrow (B,\Gamma\rightarrow\Theta)\rightarrow (A\vee B,\Gamma\rightarrow\Theta)$ such that $\lambda X\, \lambda Y.$or-L(X,Y) ,where X is the variable for the subproof $(A,\Gamma\rightarrow\Theta)$, and Y is the variable for the subproof $(B,\Gamma\rightarrow\Theta)$.

We can consider that the proofs whose proof tree are different only at the leaves are similar together. Basing on this idea, an analogical reasoning system will be designed.

3 Schemata for Proof Analogy

3.2 Simple Schema as proof types

We call a formulae whose proofs have already been known to be a guiding formulae or guiding problem.We assume that some guiding problems are collected as a database. A schema constructed from guiding problem g is defined as a formulae in which some predicates of g are abstracted as predicate variables. A *simple schema* is a schema which is constructed from g by simply replacing several predicates appearing in g with predicate variables. For example, let g be a formulae such that

$$g = [p(a)\vee q(b)]\wedge\forall x(p(x)\supset q(x))\supset \exists x.q(x).$$

Then the following formulae is a simple schema.

$$schema_g = (P(a)\vee Q(b))\wedge\forall x(P(x)\supset Q(x))\supset \exists x.Q(x)$$

,where P,Q are 2nd order predicate variables. The proof tree for $schema_g$ is given by replacing the predicates p, q with P, Q as illustrated in Fig.4.

$$\dfrac{\dfrac{\dfrac{\dfrac{Q(a)\rightarrow Q(a)}{P(a)\rightarrow P(a)\quad Q(a)\rightarrow \exists xQ(x)}}{P(a),P(a)\supset Q(a)\rightarrow \exists xQ(x)}}{P(a),\forall x\,(P(x)\supset Q(x))\rightarrow \exists xQ(x)}\qquad \dfrac{\dfrac{Q(b)\rightarrow Q(b)}{Q(b)\rightarrow \exists xQ(x)}}{Q(b),\forall x\,(P(x)\supset Q(x))\rightarrow \exists xQ(x)}$$

$$\dfrac{P(a)\vee Q(b),\ \forall x\,(P(x)\supset Q(x))\rightarrow \exists xQ(x)}{(P(a)\vee Q(b))\wedge\forall x\,(P(x)\supset Q(x))\rightarrow \exists xQ(x)}$$

$$\rightarrow (P(a)\vee Q(b))\wedge\forall x\,(P(x)\supset Q(x))\supset \exists xQ(x)$$

Fig.4 A simple shema construction

This means that the type of $schema_g$ is the following.

$$[P(a)\rightarrow P(a)]\rightarrow[Q(a)\rightarrow Q(a)]\rightarrow[Q(b)\rightarrow Q(b)]\rightarrow[\rightarrow(P(a)\vee Q(b))\wedge\forall x\,(P(x)\supset Q(x))\supset\exists x\,Q(x)]$$

and the proof term of $schema_g$ is given as follows:

proof($schema_g$)=imp-R(and-L(or-L(all-L(imp-L(P(a)→P(a),some-R(Q(a)→Q(a))),
thin-L(some-R(Q(b)→Q(b))))))).

From this $schema_g$, the proof of any formulae obtained by replacing the symbols P,Q of $schema_g$ with any formulas can be derived. This depends on the following well-known property.

[Proposition 3.1:Formulae substitution rule]

If a sequent $\Gamma \rightarrow A$ is provable, then the substituted formulae $\Gamma[P:=p(x_1,x_2,\ldots,x_n)] \rightarrow A[P:=p(x_1,x_2,\ldots,x_n)]$ is also provable, where $[P:=p(x_1,x_2,\ldots,x_n)]$ is a substitution of the formulae in the form of $P(t_1,t_2,\ldots,t_n)$ in Γ and A with $p(x_1:=t_1,x_2:=t_2,\ldots,x_n:=t_n)$.

Example 3.1 The proof of the following formulae h which is obtained from $schema_g$ by substituting P and Q by $p \supset r$ and $s \wedge t$ respectively has similar proof structure to $schema_g$ as shown in Fig.5.

$h=[\rightarrow ((p(a) \supset r(b)] \vee (s(b) \wedge t(b))) \wedge \forall x((p(x) \supset r(x)) \supset ((s(x) \wedge t(x)) \supset \exists x.(s(x) \wedge t(x))]$

Its proof term is obtained as $proof(h)=[proof(schema_g)](p \supset r)(s \wedge t)$.

$$\dfrac{\dfrac{\dfrac{}{p(a) \supset r(a) \rightarrow p(a) \supset r(a)} \quad \dfrac{}{s(a) \wedge t(a) \rightarrow s(a) \wedge t(a)}}{(a) \supset r(a), p(a) \supset r(a) \supset s(a) \wedge t(a) \rightarrow \exists x(s(x) \wedge t(x))} \quad \dfrac{}{s(a) \wedge t(a) \rightarrow \exists x(s(x) \wedge t(x))}}{p(a) \supset r(a), \forall x(p(x) \supset r(x) \supset s(x) \wedge t(x)) \rightarrow \exists x(s(x) \wedge t(x))} \quad \dfrac{\dfrac{}{s(b) \wedge t(b) \rightarrow s(b) \wedge t(b)}}{s(b) \wedge t(b) \rightarrow \exists x(s(x) \wedge t(x))}$$

$$\dfrac{p(a) \supset r(a), \forall x(p(x) \supset r(x) \supset s(x) \wedge t(x)) \rightarrow \exists x(s(x) \wedge t(x)) \quad s(b) \wedge t(b), \forall x(p(x) \supset r(x) \supset s(x) \wedge t(x)) \rightarrow \exists x(s(x) \wedge t(x))}{(p(a) \supset r(a)) \vee (s(b) \wedge t(b)), \forall x((p(x) \supset r(x)) \supset s(x) \wedge t(x)) \rightarrow \exists x s(x) \wedge t(x)}$$

$$\dfrac{(p(a) \supset r(a)) \vee (s(b) \wedge t(b))), \forall x((p(x) \supset r(x)) \supset s(x) \wedge t(x)) \rightarrow \exists x s(x) \wedge t(x)}{\rightarrow [(p(a) \supset r(a)) \vee (s(b) \wedge t(b))] \wedge \forall x(p(x) \supset r(x)) \supset (s(x) \wedge t(x))) \supset \exists x.(s(x) \wedge t(x))}$$

Fig.5 The proof of h by analogy .

3.2 Fundamental Schema with Constraints

We call the sequents appearing at the conclusions of LK inference rules as the fundamental formulae. Let *form*(A,B) be a fundamental formulae whose proof is obtained from assumption *Asp*(A,B). We denote the formulas in which A, B are replaced with some formulas X,Y or Φ by F_schema(X,Y) or F_schema(Φ) respectively. and call them fundamental schema. Now, we discuss the provability of F_schemata. Since X,Y and Φ take any formulas as their values, we consider them second order variables. We show, at first , that if a formula *form*(A,B) is provable from assumption *Asp*(A,B), then corresponding any F_schema is also derivable from the same assumptions *Asp*(A,B) and the imposed constraints introduced according to the used rule.

For example, let *form*(A,B) be a sequent $(P \wedge Q) \vee R, \Gamma \rightarrow \Theta$. and this is proved using the or_L inference rule with $Asp(A,B)=\{(P \wedge Q), \Gamma \rightarrow \Theta, R, \Gamma \rightarrow \Theta\}$. Then any formula of the form $X \vee Y, \Gamma \rightarrow \Theta$,where X and Y are arbitrary formulas, can be proved using Asp(A,B) and constraints $X \rightarrow (P \wedge Q)$, $Y \rightarrow R$ in the following way.

$$\dfrac{\dfrac{X \rightarrow (P \wedge Q) \quad (P \wedge Q), \Gamma \rightarrow \Theta}{X, \Gamma \rightarrow \Theta} \quad \dfrac{Y \rightarrow R \quad R, \Gamma \rightarrow \Theta}{Y, \Gamma \rightarrow \Theta}}{X \vee Y, \Gamma \rightarrow \Theta}$$

Fig.6 A proof construction based on F_schema.

This property holds each F_schemata and is stated as follows.

[Theorem 3.1] If a fundamental formulae is provable and constraints are satisfied, then the corresponding F_schema is provable using the same assumptions of the fundamental formulae and the constraints

Proof: The proof structure of each fundamental schema can be constructed by substituting the

symbols A, B with X, Y or Φ respectively, and combining the partial proofs for constraints using Cut rule. Since the proof for each inference rule and constraints are provable, the combined proof for each F_schema is provable. Q.E.D

F formulae	F_schema	Constraint	Proof construction
$\frac{A,\Gamma\to\Theta \quad B,\Gamma\to\Theta}{A\vee B,\Gamma\to\Theta}$	$\frac{X,\Gamma\to\Theta \quad Y,\Gamma\to\Theta}{X\vee Y,\Gamma\to\Theta}$	$X\to A,\ Y\to B$	$\frac{X\to A \quad A,\Gamma\to\Theta \qquad Y\to B \quad B,\Gamma\to\Theta}{\frac{X,\Gamma\to\Theta \qquad Y,\Gamma\to\Theta}{X\vee Y,\Gamma\to\Theta}}$
$\frac{\Gamma\to\Theta,A,B}{\Gamma\to\Theta,A\vee B}$	$\frac{\Gamma\to\Theta,X,Y}{\Gamma\to\Theta,X\vee Y}$	$A\to X,\ B\to Y$	$\frac{\Gamma\to\Theta,A,B \quad A\to X \quad B\to Y}{\frac{\Gamma\to\Theta,X,Y}{\Gamma\to\Theta,X\vee Y}}$
$\frac{A,B,\Gamma\to\Theta}{A\wedge B,\Gamma\to\Theta}$	$X\wedge Y,\Gamma\to\Theta$	$\frac{X,\ Y,\Gamma\to\Theta}{X\to A,\ Y\to B}$	$\frac{X\to A \quad Y\to B \quad A,B,\Gamma\to\Theta}{\frac{X,\ Y,\ \Gamma\to\Theta}{X\wedge Y,\ \Gamma\to\Theta}}$
$\frac{\Gamma\to A \quad \Gamma\to B}{\Gamma\to A\wedge B}$	$\frac{\Gamma\to X \quad \Gamma\to Y}{\Gamma\to X\wedge Y}$	$A\to X,\ B\to Y$	$\frac{\Gamma\to A \quad A\to X \qquad \Gamma\to B \quad B\to Y}{\frac{\Gamma\to X \qquad\qquad \Gamma\to Y}{\Gamma\to X\wedge Y}}$
$\frac{\Gamma\to\Theta,A \quad B,\Delta\to\Lambda}{\Gamma,A\supset B,\Delta\to\Theta,\Lambda}$	$\frac{\Gamma\to\Theta,X \quad Y,\Delta\to\Lambda}{\Gamma,\ X\supset Y,\Delta\to\Theta,\Lambda}$	$A\to X,Y\to B$	$\frac{\Gamma\to\Theta,A \quad A\to X \quad B,\Delta\to\Lambda \quad Y\to B}{\frac{\Gamma\to\Theta,X \quad Y,\Delta\to\Lambda}{\Gamma,\ X\supset Y,\Delta\to\Theta,\Lambda}}$
$\frac{\Gamma,A\to\Theta,B}{\Gamma\to\Theta,A\supset B}$	$\frac{X,\Gamma\to\Theta,Y}{\Gamma\to\Theta,\ X\supset Y}$	$X\to A,B\to Y$	$\frac{\Gamma,A\to\Theta,B \quad X\to A \quad B\to Y}{\frac{X,\Gamma\to\Theta,Y}{\Gamma\to\Theta,\ X\supset Y}}$
$\frac{A[x:=t],\Gamma\to\Theta}{\forall xA(x),\Gamma\to\Theta}$	$\frac{\Phi(x:=t),\Gamma\to\Theta}{\forall x\Phi(x),\Gamma\to\Theta}$	$\forall x.\Phi(x)\to\forall x.A(x)$	$\frac{\Phi(x:=t)\to A(x:=t) \quad A[x:=t],\Gamma\to\Theta}{\frac{\Phi(x:=t),\Gamma\to\Theta}{\forall x\Phi(x),\Gamma\to\Theta}}$
$\frac{\Gamma\to\Theta,A[x:=y]}{\Gamma\to\Theta,\forall xA(x)}$	$\frac{\Gamma\to\Theta,\Phi[x:=y]}{\Gamma,\to\Theta,\forall x\Phi(x)}$	$\forall x.A(x)\to\forall x.\Phi(x)$	$\frac{\Gamma\to\Theta,A(x:=y) \quad A(x:=y)\to\Phi(x:=y)}{\frac{\Gamma\to\Theta,\Phi[x:=y]}{\Gamma\to\Theta,\forall x\Phi(x)}}$
$\frac{A[x:=y],\Gamma\to\Theta}{\exists xA(x),\Gamma\to\Theta}$	$\frac{\Phi[x:=y],\Gamma\to\Theta}{\exists x\Phi(x),\Gamma\to\Theta}$	$\exists x\Phi(x)\to\exists xA(x)$	$\frac{\Phi(x:=y)\to A(x:=y) \quad A(x:=y),\Gamma\to\Theta}{\frac{\Phi[x:=y]\ \Gamma\to\Theta}{\exists x\Phi(x),\ \Gamma\to\Theta}}$
$\frac{\Gamma\to\Theta,A[x:=t]}{\Gamma\to\Theta,\exists xA(x)}$	$\frac{\Gamma\to\Theta,\Phi[x:=t]}{\Gamma\to\Theta,\exists x\Phi(x)}$	$\exists x.A(x)\to\exists x\Phi(x)$	$\frac{\Gamma\to\Theta,A[x:=t] \quad A(x:=t)\to\Phi(x:=t)}{\frac{\Gamma\to\Theta,\Phi[x:=t]}{\Gamma\to\Theta,\ \exists x\Phi(x)}}$
$\frac{\Gamma\to\Theta,A}{\sim A,\Gamma\to\Theta}$	$\frac{\Gamma\to\Theta,X}{\sim X,\Gamma\to\Theta}$	$A\to X$	$\frac{\Gamma\to\Theta,A \quad A\to X}{\frac{\Gamma\to\Theta,X}{\sim X,\Gamma\to\Theta}}$
$\frac{A,\Gamma\to\Theta}{\Gamma\to\Theta,\sim A}$	$\frac{X,\Gamma\to\Theta}{\Gamma\to\Theta,\sim X}$	$X\to A$	$\frac{X\to A \quad A,\Gamma\to\Theta}{\frac{X,\Gamma\to\Theta}{\Gamma\to\Theta,\sim X}}$

Fig.7 F_schemata and proof structures

The proof for each F_schema is summarized in Fig.7, where the double underlines mean some composition of inference rules. By *proof*(F_schema) and *proof**(F_schema), we mean the partial proof whose assumptions are not specified and its completed proof, respectively.Then it holds the following term representation for proofs of F_schemata

*proof**(F_schema(X,Y)) =*proof*(F_schema(X,Y))+ proof(X,Y:A,B),

proof(F_schema(X,Y))=[λAλB.(*proof*(guiding-form(A,B))]X Y

where *proof*(X,Y:A,B) is a proof of assumptions in partial proof obtained from constraints, and + means its addition to partial proof.

Example 3.2 For a F_schema having $X \vee Y, \Gamma \to \Theta$ as its conclusion, its partial proof is given as follows.

$$\frac{A, \Gamma \to \Theta \quad B, \Gamma \to \Theta}{A \vee B, \Gamma \to \Theta} \quad \Rightarrow \quad \frac{X, \Gamma \to \Theta \quad Y, \Gamma \to \Theta}{X \vee Y, \Gamma \to \Theta}$$

The term representation of this F_schema is established by substituting the symbols A and B with X and Y of the proof term of fundamental formulae as follows.

$$proof^*(X \vee Y, \Gamma \to \Theta) = proof(X \vee Y, \Gamma \to \Theta) + proof(X,Y:A,B)$$
$$proof(X \vee Y, \Gamma \to \Theta) = [\lambda A \lambda B.(or\_L(proof(A,\Gamma \to \Theta), proof(B,\Gamma \to \Theta)))]XY$$
$$proof(X,Y:A,B) = proof(X,\Gamma \to \Theta) + proof(Y,\Gamma \to \Theta)$$
$$proof(X,\Gamma \to \Theta) = cut(proof(A,\Gamma \to \Theta), subproof(X \to A))$$
$$proof(Y,\Gamma \to \Theta) = cut(proof(B,\Gamma \to \Theta), subproof(Y \to B))$$

3.3 Schema with constraints

In this section,we shall discuss the relation between the proof terms of schemata with constraints and their instances. Let $form(g(\dot{A}))$ be a formula whose proof has already been derived as $proof(g(\dot{A}))$, and let $form(g(\dot{X}))$ and $proof(g(\dot{X}))$ be the formula and proof term obtained from $form(g(\dot{A}))$ and $proof(g(\dot{A}))$ by replacing some of the symbols in $\dot{A}$ with the symbols in $\dot{X}$ respectively, where $\dot{A}$ and $\dot{X}$ are the list of predicates in $form(g(\dot{A}))$ and $form(g(\dot{X}))$. It is noted that $proof(g(\dot{X}))$ is not always completed. We denote a completed proof of $proof(g(\dot{X}))$ by $proof^*(g(\dot{X}))$. In the followings, we take $form(g(\dot{X}))$ as a schema constructed from g, and sometimes denote $form(g(\dot{X}))$ as $schema_g$.For example, let $form(g(p,q))=[p(a) \vee q(b)] \wedge \forall x(p(x) \supset q(x)) \supset \exists x.q(x)$. Then we have $schema_g = form(g(\Phi, \Xi, \Psi, \Theta)) = (\Phi(a) \vee \Psi(b)) \wedge \forall x(\Xi(x) \supset Q(x)) \supset \exists x.\Theta(x)$ as one of the schemata. The proof term of this schema $proof(g(\Phi, \Xi, \Psi, \Theta))$ is obtained as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\Phi(a) \to \Xi(a)} \quad \dfrac{\boxed{Q(a) \to \Theta(a)}}{Q(a) \to \exists x \Theta(x)}}{\Phi(a), \Xi(a) \supset Q(a) \to \exists x \Theta(x)}}{\Phi(a), \forall x (\Xi(x) \supset Q(x)) \to \exists x \Theta(x)} \quad \dfrac{\boxed{\Psi(b) \to \Theta(b)}}{\Psi(b) \to \exists x \Theta(x)}}{\Phi(a) \vee Q(b), \forall x (\Xi(x) \supset Q(x)) \to \exists x \Theta(x)} \quad \dfrac{}{\Psi(b), \forall x (\Xi(x) \supset Q(x)) \to \exists x \Theta(x)}}{(\Phi(a) \vee \Psi(b)) \wedge \forall x (\Xi(x) \supset Q(x)) \to \exists x \Theta(x)}}{\to (\Phi(a) \vee \Psi(b)) \wedge \forall x (\Xi(x) \supset Q(x)) \supset \exists x \Theta(x)}$$

Constraints:
$\Phi(a) \to \Xi(a)$
$\Psi(b) \to \Theta(b)$
$Q(a) \to \Theta(a)$

Fig.8 proof schema.

It is noted that each leaf of $proof(g(\dot{A}))$ is of the form $A, \Gamma \to A, \Delta$, and the corresponding sequents of $proof(g(\dot{X}))$ may be partial proof. That is,the sequents at the leaves of the proof tree are the subproofs which should be proved further. They are called the constraints. In the example of Fig.8, we have to prove the constraints and combine with $proof(g(\Phi,\Xi,\Psi,\Theta))$ to obtain the complete proof $proof^*(g(\Phi,\Xi,\Psi,\Theta))$ from $proof(g(\Phi,\Xi,\Psi,\Theta))$.

Let $constr(g(\dot{X}:\dot{A}))$ be the set of constraints between $form(g(\dot{A}))$ and $form(g(\dot{X}))$. For example, the formulae $form(g(p,q,r,s))=(p(a) \vee q(a)) \wedge \forall x(r(x) \supset s(x)) \supset \exists x.t(x)$ is provable if the constraints $p(a) \to s(a)$, $r(b) \to t(b)$, $r(a) \to t(b)$ are all provable.This intuitive meaning is given in the following inference rule.

$$\frac{proof(g(\dot{X}))\quad proof(constr(g(\dot{X}{:}\dot{A}))\ \ldots}{proof^{*}(g(\dot{X}))}$$

[Theorem 3.2] If $form(g(\dot{A}))$ is provable, then the proof of schema $form(g(\dot{X}))$ is provable. The proof $proof^{*}(g(\dot{X}))$ is given by patching the proof$(g(\dot{X}))$ with the proof of constraints $proof(constr(g(\dot{X}{:}\dot{A})))$ which are introduced according to the used inference rules.

Proof: Let (r) be any LK inference rule used at the root of $proof(g(\dot{A}))$. Then it holds from Theorem 3.1 that the corresponding part of the $proof(g(\dot{X}))$ is provable under the constraints decided by (r). For each step of the proof $proof(g(\dot{X}))$, the corresponding part of proof of $proof(g(\dot{X}))$ is correct in the similar discussion, where its constraints are the union of the constraints used until the step. The final step of the proof, i.e., the leaves of $proof(g(\dot{X}))$, are not completed, but they are also provable from Theorem 3.1 using the same assumption to $proof(g(\dot{A}))$ and constraints. That is, the $proof^{*}(g(\dot{X}))$ can be established by combining the $proof(g(\dot{X}))$ and $proof(constr(g(\dot{X}{:}\dot{A})))$. Q.E.D

4. Schema Construction

4.1 Unification based Abstraction

A schema is a meta representation for formulas which are syntactically similar, and its proof term represents the proof type. We can consider the proof term of each schema as the specification of proofs. The proofs of the instances of the schema have the similar structure. This means that each instance formulae of a schema is a realization of the specification corresponding to the schema and its proof is an instance of the proof schema. This relation is illustrated in Fig. 9.

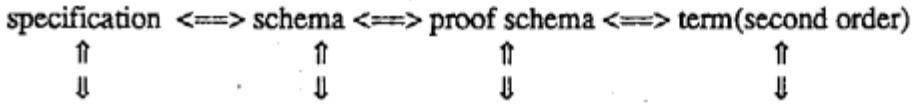
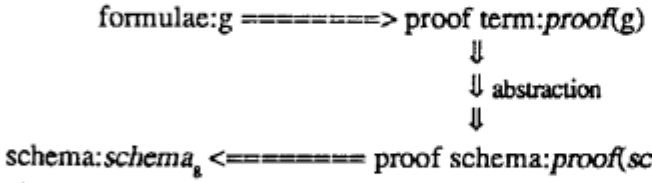


Fig.9 Schema as proof types

Unification theory is concerned with problems of finding the existence of some substitution σ for given terms t_1, t_2 such that $s(t_1)=s(t_2)$. The dual of the unification is called the *generalization* or *anti-unification*. Intuitively, the generalization is to find for given terms s, t, a term z of which both s and t are instances, where z is called a *generalizer* of s, t[12]. As we have observed in the previous section, the generalization for g,h is performed by transforming to *proof*(g), *proof*(h) using this matching algorithm.



Concerning to the higher order unification and anti-unification algorithm, the other articles should be referred [7,9,12,15]. A term built on variables of order at most two and constants of order at most three is called a second order term. The unification between a second order term and the first order

rigid term is called the second order matching problem. It holds the following property.

Proposition 4.1[15] The 2nd order matching problem is decidable and produces a complete set of minimal match of t and r.

4.2 Schema Construction By Abstraction

In this section, we demonstrate how schemata are constructed from guiding formulas using examples. Let g and h be a guiding formulae and a given target formulae respectively. Assume that the proof of g is given as in Fig.10 and the type of proof structure of g is represented as *proof*(g). The schema for proving h is constructed by the generalization of g and h assuming that h is provable according to this proof type.The abstraction is done by replacing the predicate p,q by second order variables P,Q and disagreement parts with new second order variables Φ . In this example, the schema $schema_g$ can be derived, and constraints should be constructed according to the F_schema. In this case, we have the constraint such that $\Phi \rightarrow P$ which appears at some leaf.Then the proof of h by analogy to g is given by patching the *proof*(g) with the proof of constraints as shown in Fig.11

$g = p(a) \wedge \forall x(p(x) \supset q(x)) \supset \exists x.q(x)$

proof(g)=imp_R(and_L(all_L(imp_L([p(a)→p(a)]
,some_R([q(a)→q(a)]))).

$h = [(r(a) \vee p(a)) \wedge (r(a) \supset p(a))] \wedge \forall x(p(x) \supset q(x)) \supset \exists x.q(x)$

proof(h)=imp_R(and_R(all_L(imp_L([(r(a) ∨p(a)) ∧(r(a) ⊃p(a)→p(a))]
,some_R([q(a)→q(a)])))

$$\frac{\dfrac{\dfrac{\dfrac{q(a) \rightarrow q(a)}{p(a) \rightarrow p(a) \quad q(a) \rightarrow \exists x q(x)}}{p(a), p(a) \supset q(a) \rightarrow \exists x q(x)}}{p(a), \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}}{\dfrac{p(a) \wedge \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}{\rightarrow p(a) \wedge \forall x (p(x) \supset q(x)) \supset \exists x q(x)}}$$

Fig.10 A natural deduction proof for g.

$schema_g = \Phi(a) \wedge \forall x(P(x) \supset Q(x)) \supset \exists x.Q(x)$,
Constraint: $\Phi \rightarrow P$

$$\frac{\dfrac{\dfrac{\dfrac{q(a) \rightarrow q(a)}{\Phi(a) \rightarrow p(a) \quad q(a) \rightarrow \exists x q(x)}}{\Phi(a), p(a) \supset q(a) \rightarrow \exists x q(x)}}{\Phi(a), \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}}{\dfrac{\Phi(a) \wedge \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}{\rightarrow \Phi(a) \wedge \forall x (p(x) \supset q(x)) \supset \exists x q(x)}}$$

(a) Proof schema $schema_g$

$$\frac{\dfrac{r(a) \rightarrow r(a) \quad p(a) \rightarrow p(a)}{r(a), r(a) \supset p(a) \rightarrow p(a)} \quad \dfrac{p(a) \rightarrow p(a)}{p(a), r(a) \supset p(a) \rightarrow p(a)}}{\dfrac{r(a) \vee p(a), r(a) \supset p(a) \rightarrow p(a)}{(r(a) \vee p(a)) \wedge (r(a) \supset p(a)) \rightarrow p(a)}}$$

(b)Proof of constraint

Fig.11 A proof construction for schema with constraints.

Next, let us consider to prove the following formula v which is syntactically similar to g using the $schema_g$: $v = q(a) \wedge \forall x(p(x) \supset q(x)) \supset \exists x.q(x)$.

But, its proof is slightly different with the previous example. By examining similarly, it is easy to see that its proof cannot be obtained unless q(a)→p(a) is provable. But this can be proved if we use the other proof schema as shown in Fig.13.

$$\frac{\dfrac{\dfrac{\dfrac{q(a) \rightarrow q(a)}{q(a) \rightarrow \exists x q(x)}}{q(a), (p(a) \supset q(a)) \rightarrow \exists x q(x)}}{q(a), \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}}{\dfrac{q(a) \wedge \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}{\rightarrow q(a) \wedge \forall x (p(x) \supset q(x)) \supset \exists x q(x)}}$$

Fig.12 A proof structure for v

$$\frac{\dfrac{\dfrac{\dfrac{\boxed{\Psi(a) \rightarrow q(a)}}{\Psi(a) \rightarrow \exists x q(x)}}{\Psi(a), (p(a) \supset q(a)) \rightarrow \exists x q(x)}}{\Psi(a), \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}}{\dfrac{\Psi(a) \wedge \forall x (p(x) \supset q(x)) \rightarrow \exists x q(x)}{\rightarrow \Psi(a) \wedge \forall x (p(x) \supset q(x)) \supset \exists x q(x)}}$$

Fig.13 Proof for $schema_v$

schema$_v$ is given in the similar form to *schema*$_g$ such that $\Psi(a)\wedge\forall x(P(x)\supset Q(x))\supset \exists x.Q(x)$,but its constraint is $\Psi\rightarrow Q$. Using this *schema*$_v$, the proof of the following formula w_

$$w= [r(a)\wedge(r(a)\supset q(a))]\wedge\forall x(p(x)\supset q(x)) \supset \exists x.q(x).$$

can be obtained by replacing Ψ with $[r(a)\wedge(r(a)\supset q(a))]$ and completing the proof of constraints $[r(a)\wedge(r(a)\supset q(a))]\rightarrow q(a)$. By such observations,we get a general proof schema such that

$$schemata = (\Phi(a)\vee\Psi(b))\wedge\forall x(P(x) \supset Q(x)) \supset \exists x.Q(x)$$

Constraints: $\Phi\rightarrow P$, $\Psi\rightarrow Q$.

Example 3.3 The analogical proving of w using *schemata* is not available though its syntax is similar to g or h, because its constraint $r(a)\rightarrow p(a)\vee q(a)$ is not provable. $w= r(a)\wedge\forall x(p(x)\supset q(x)) \supset \exists x.q(x)$.

4.Proving by Analogy

4.1 Proof Procedure

The rough sketch of this procedure is illustrated in Fig.14. We assume that standard schema have already been obtained as schema database, and let its elements be $S_1,S_2,\ldots,S_n$.Firstly, a given target problem w is checked if some similar guiding problem exists or not. There are two cases for this step. One is to construct a schema from g and w by generalization. The other is to search a schema on the schema database which is unifiable both with g and w. We are intending to develop a system which combine the both cases. This similarity check is examined using 2nd order matching algorithm. If there exists a schema S which match to w, then a unifier is produced. The proof of w is derived by σ(proof(S))+proof(constraints).

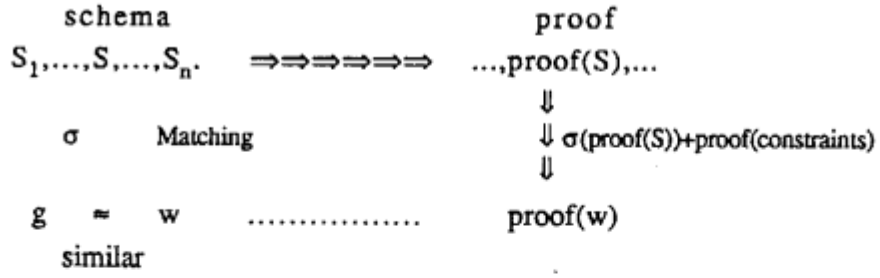


Fig.14 Analogical Proving procedure

Procedure
input: w (formulae) ;*output*: proof(w);
begin
Find a schema S which match with w
(1) *if* there is no such schema *then* stop and output *"prove by yourself"*
(2) *else* choose (in nondetermistic) a schema S;
(2-1)*compute* unifier such that σ(S)= w
(2-2) *check* if it satisfies the constraints
if it satisfies *then output* σ[proof(S)]+proof(constraint)
else "prove by yourself"
end

It is noted that the procedure uses only 2nd order matching, it is realized in the computable scope.

[Theorem 4.1] The analogical proof reasoning procedure proposed here for LK is computable.

4.2 Working Examples

In this section, We demonstrate an example. Let h be a formula such that

$$h= ((p(a)\wedge r(a))\vee(q(a)\wedge r(a))\wedge\forall x(p(x)\supset q(x))\supset \exists x.q(x).$$

Here, we assume that we want to solve this by the analogy with g.

$$g =[p(a)\vee q(b)]\wedge\forall x(p(x)\supset q(x))\supset \exists x.q(x).$$

Their proof terms are given in the following forms, where axiom parts of *proof*(h) are arranged according to the proof structure.

g====> *proof*(g)=imp-R(and-L(or-L(all-L(imp-L(p(a)→p(a),some-R(q(a)→q(a)))),

thin-L(some-R(q(b)→q(b))))))).

⇑⇓ *generalization*

h====>*proof*(h)= imp-R(and-L(or-L(all-L((imp-L((p(a)∧r(a))→p(a),some-R(q(a)→q(a)))),

thin-L(some-R((q(a)∧r(a))→q(a))))))).

By the generalization, we get the following proof schema.

imp-R(and-L(or-L(all-L((imp-L(Φ(a)→P(a),some-R(Q(a)→Q(a)))),

thin-L(some-R(Ψ(a)→Q(a))))))).

Then we get the following schema $schema_g$

$$schema_g = (\Phi(a)\vee\Psi(b))\wedge\forall x(P(x)\supset Q(x))\supset\exists x.Q(x)$$

constraints: $\Phi(a)\rightarrow P(a)$, $\Psi(b)\rightarrow Q(b)$

, where Φ and Ψ are predicate variables. The *proof*($schema_g$) and proof(constr($\Phi(a)\rightarrow P(a), \Psi(b)\rightarrow Q(b)$) are obtained as in Fig.15(a),(b).

$$\dfrac{\dfrac{\dfrac{\dfrac{Q(a)\rightarrow Q(a)}{\Phi(a)\rightarrow P(a)\quad Q(a)\rightarrow\exists xQ(x)}\quad \dfrac{\Psi(b)\rightarrow Q(b)}{\Psi(b)\rightarrow\exists xQ(x)}}{\Phi(a),P(a)\supset Q(a)\rightarrow\exists xQ(x)}}{\Phi(a),\forall x(P(x)\supset Q(x))\rightarrow\exists xQ(x)\qquad \Psi(b),\forall x(P(x)\supset Q(x))\rightarrow\exists xQ(x)}}{\Phi(a)\vee Q(b),\forall x(P(x)\supset Q(x))\rightarrow\exists xQ(x)}$$

$$(\Phi(a)\vee\Psi(b))\wedge\forall x(P(x)\supset Q(x))\rightarrow\exists xQ(x)$$

$$\rightarrow(\Phi(a)\vee\Psi(b))\wedge\forall x(P(x)\supset Q(x))\supset\exists xQ(x)$$

(a) Proofs for $schema_g$

Constraints:

$\Phi(a)\rightarrow P(a)$

$\Psi(b)\rightarrow Q(b)$

$$\dfrac{P(a)\rightarrow P(a)}{P(a),R(a)\rightarrow P(a)}$$

$$P(a)\wedge R(a)\rightarrow P(a)$$

$$\dfrac{Q(a)\rightarrow Q(a)}{Q(a),R(a)\rightarrow Q(a)}$$

$$Q(a)\wedge R(a)\rightarrow Q(a)$$

(b) Proofs for constraints.

Fig.15 Schema with constraints for proving h.

By the matching of h with $schema_g$ as typed terms, we get a substitution $\sigma=\{\Phi:=p\wedge r, \Psi:=q\wedge r, P:=p, Q:=q\}$. The proof(h) is derived as

proof(h)=[λΦλΨλPλQ. *proof*($schema_g$)](p∧r)(q∧r)(p)(r)+

[λΦλΨλPλQ. *proof*(constr(Φ→P,Ψ→Q))] (p∧r)(q∧r)(p)(r)

6.Discussions

We proposed an analogical reasoning for LK proof system based on higher-order abstraction. By this approach, a kind of proof system by analogy can be realized in natural way. Especially, it holds a similar interpretation of the analogy to the formulae as type concept such that the schemata

corresponds to specifications and object proofs to their realizations. The procedure proposed here can be realized using the higher order unification algorithm for typed terms in the computable scope.

The analogical reasoning system can be considered a reasoning system which uses the already known knowledge as heuristics. By these approach, we can expect to reduce the difficulties arising in the nondeterministic process by solving it according to the guiding solutions. If the knowledge of guiding problem is a assumption, then the proposed analogical reasoning process becomes a kind of abductive reasoning system. Such nondeductive reasoning systems may be formalized using proposed framework theoretically.

However, there exist several important problems to be solved . The most essential one is to design an efficient unification algorithm.In the unification procedure, there exists the nondeterminism in applying the imitation and projection rules. Therefore we should devise more tightly the strategy of unification procedure. One solution for this problem is to restrict the syntax of λ-term language and the usages of second order variables adequately.

The other problem is that the schema expressed by second order variables are too general for many cases. Hence undesirable unifiers will be output sometimes. In order to specify the schema more precisely, some additional axioms should be attached to such schema.

In present, the reasoning is realized as a simple substitution using schema. If a more powerful inference mechanism such as the higher order resolution system is introduced in this framework, the more flexible and intelligent analogical reasoning system can be realized.

When we prove a formulae using this system, then a proof is able to be obtained. But, it is not always good proof for us. To translate the result to a better proof form is one interesting problem.

Acknowledgement

This research was supported partly by the Telecommunications and Advancement Foundation.

5. References

[1]P.B.Andrew: An Introduction to Mathematical Logic and Type Theory, Academic Press.inc.,1986.

[2] B.Brock,S.Cooper and W.Pierce: Analogical Reasoning and Proof Discovery, LNCS, No.310, 9th Int.Conf on Automated Deduction, pp454 ~468,1988

[3]M.R.Donat, L.A.Wallen: Learning and Applying Generalized Solutions using Higher Order Resolution, LNCS,No.310,Int.Conf on Automated Deduction,pp41~60,1988.

[4]K.Fujita,M.Harao: Proving Based on Similarity, Proc. of 2nd Workshop on Algorithmic Learning Theory, Japan Society of Artificial Intelligence, pp.213-223,1991,10

[5] R.Greiner: Abstraction-Based Analogical Inference, Herman (Ed.), Analogical Reasoning,1988,pp.147-170.

[6]M.Haraguchi,S.Arikawa: A Formulation of Analogical Reasoning and Its Realization, Journal of JSAI, Vol.1 No.1,pp132~139,1986.

[7] M.Harao,K.Iwanuma: Complexity of Higher-Order Unification Algorithm, Computer Software, Vol.8,No.1, Jan.1991,pp.41-53

[8]M.Harao: Knowledge Processing Based on Higher-Order Unification, Technical Report JSAI, SIG-FAI-8903-3, pp21~30,1989.

[9]G.P.Huet, B.Lang: Proving and Applying Program Transformations Expressed with Second- Order Patterns, Acta Informatica, 11, pp 31 ~55, 1978.

[10]S.Kedar-Cabelli: Analogy from a unified perspective, Herman (Ed.), Analogical Reasoning,1988,pp.65-103.

[11] D.A.Miller, A.Felty: An Integration of Resolution and Natural Deduction Theorem Proving, Proc. of AAAI86,pp.198-202.1986

[12]S.Muggleton, C.Feng: Efficient Induction of Logic program, Proc. of Workshop on Algorithmic Learning Theory, 1990,10,pp.368-382.

[13]I.Niniluoto: Analogy and Similarity in Scientific Reasoning, Herman(Ed.) Analogical Reasoning, Kluwer Academic Pub. 1988, pp.271-298.

[14] D.A.Plaisted: Theorem Proving with Abstraction, Artificial Intelligence, Vol.16,No.1, 1981,pp.47-108.

[15] W.Snyder, J.Gallier: Higher Order Unification Reviced, Journal of Symbolic Computation,No.8,1989,pp.101-140.

[16] T.B. de la Tour, R.Caferra: Proof Analogy In Interactive Theorem Proving, IJCAI '87, pp95~99,1987.

Explanation-Based Generalization by Analogical Reasoning

Eizyu Hirowatari Setsuo Arikawa

Department of Information Systems
Kyushu University 39, Kasuga 816, Japan

Abstract

The EBG system builds an explanation and learns a concept description as its generalization, when a domain theory is complete. It does not work when a domain theory is incomplete. In order to solve this problem, we propose EBG by analogical reasoning as a method to carry out EBG for an incomplete domain theory. In this paper, we present the mathematical formalization of EBG, construct EBG by analogical reasoning and realize it for Prolog programs.

1 Introduction

EBG(Explanation-Based Generalization) takes as input a domain theory, a training example, a goal concept and an operationality criterion. Then, it returns as output an operational concept definition that is a generalization of the training example, and is a sufficient condition for the goal concept. EBG has extensively been studied in the field of machine learning[3, 8, 11, 12], and reported that it efficiently solved many problems. However, mathematical discussions for EBG have not yet been developed well. In the present paper we present the mathematical formalization of EBG.

EBG needs a complete domain theory so that it does not work when a domain theory is incomplete. It is important to solve this problem as pointed out by many papers[1, 3, 4, 9, 12, 13, 15, 16]. In this paper, we propose EBG by analogical reasoning which copes with an incomplete domain theory. We make up rules which are needed in a domain theory by means of analogical reasoning[2, 5, 6, 7].

Numao and Shimura presented a method of analogical inference using explanation-based learning[14]. In contract we present a method of EBG with an incomplete domain theory using analogical inference.

This paper is organized as follows. In Section 2 we present the mathematical formalization of EBG and characterize EBG. In Section 3 we give some concepts on analogical reasoning necessary for our discussion. In Section 4 we present the EBG by analogical reasoning. In Section 5 we describe the realized EBG by analogical reasoning for Prolog programs.

2 Formalization of EBG

In this section we define the EBG in a formal way. See[10] for detailed definitions of first order logic.

An atomic formula is simply called an *atom*. An atomic formula without variables is specially called a *ground atom*. A *definite clause* is a clause of the form

$$A \leftarrow B_1, \ldots, B_n \ (n \geq 0)$$

where A and B_j are positive literals, and it called a *rule*. A finite set D of rules is called a *domain theory*. A non-empty finite set T of ground atoms is called a *training example*. Both D and T are sets of definite clauses. $P = D \cup T$ is called a *program*. P has at least one constant symbol, because T in a non-empty set.

Let U_P be the Herbrand universe for P, and B_P be the Herbrand base for P. Let M_P be the least Herbrand model for P, that is, the set of all ground atoms which are logical consequences of P.

A generalization G of a ground atom is called a *goal concept*. Let P be a program, and Π_P be the set of predicate symbols in P. A subset O of Π_P is called an *operationality criterion*. Let $\Pi(O)$ be the set of all atoms which have predicate symbols in O.

The triple (P, G, O) is called an *input* of EBG.

Definition 1 For an input $I = (P, G, O)$, an *explanation tree* EX for I is a finite tree satisfying the following conditions:

(a) Each node of the tree is an element of M_P.

(b) The root node is an instantiation of G.

(c) If a node α in the tree has children $\beta_1, \ldots, \beta_n$ $(n \geq 1)$ in this order, then $\alpha \leftarrow \beta_1, \ldots, \beta_n$ is an instantiation of a rule in P.

(d) Nodes in $\Pi(O)$ have no children.

Standard Prolog systems employ the computation rule which always selects the left-most atom in a goal together with the depth-first search. We number rules which are used in the explanation tree in the following way:

(1) Each node which is not an element of $\Pi(O)$ is numbered with the depth-first search.

(2) Let α be a node which has children $\beta_1, \ldots, \beta_n$ $(n \geq 1)$ in this order and C be a rule in P which a generalization of $\alpha \leftarrow \beta_1, \ldots, \beta_n$. If α has a number j, then C is denoted by C_j.

By means of the numbering above we can use the rules in the generalized derivation.

Definition 2 For an input $I = (P, G, O)$, a *generalized derivation* GE for I is an SLD-derivation which consists of a finite sequence $(\leftarrow G_0) = (\leftarrow G), \leftarrow G_1, \ldots, \leftarrow G_n$ of goals, a sequence $C_1, \ldots, C_n$ of rules in P, which are given in EX, and a sequence $\theta_1, \ldots, \theta_n$ of most general unifiers such that each G_{k+1} is derived from G_k and C_{k+1} using θ_{k+1}.

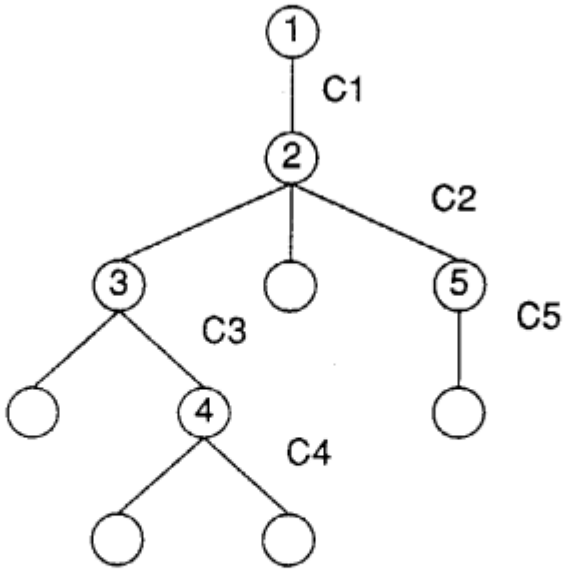


Figure 1: Numbering nodes and sequence of rules in a explanation tree.

A generalized derivation GE depends on an explanation tree EX. An *explanation* is to construct an explanation tree EX. A *generalization* is to generalize a sufficient conditions of a goal concept by a generalized derivation GE.

Definition 3 EBG is to derive a general definition R of a goal concept G from an input $I = (P, G, O)$ by an explanation and generalization.

It is denoted by

$$I \overset{EBG}{\rightarrow} R,$$

where

$$R = \begin{cases} G & (n = 0) \\ G \rightarrow G_n & (n \geq 1) \end{cases},$$

and $(\leftarrow G_n)$ is the goal of the generalized derivation of depth n.

Now we can easily prove the following facts:

Proposition 1 For any input $I = (P, G, O)$, $I \overset{EBG}{\rightarrow} R$ iff $G \in \Pi(O)$.

Theorem 1 *For an input $I = (P, G, O)$, let R be $G \leftarrow A_1, \ldots, A_m$ $(m \geq 1)$, where A_j are positive literals. Then, $I \overset{EBG}{\rightarrow} R$ iff the following conditions hold:*

(a) There exists a $P_r = \{(B \leftarrow B_1, \ldots, B_n) \in P | B \notin \Pi(O), n \geq 1\}$ such that $P_r \vdash R$.

(b) There exists a substitution θ such that $\{R\} \cup \Pi(O)|_{M_p} \vdash G\theta$.

(c) $G \notin \Pi(O)$.

These facts assert that EBG is a deductive reasoning from a program which satisfies the input conditions.

3 Analogical Reasoning

Haraguchi and Arikawa[2, 5, 6, 7] defined a formal analogy as a relation of terms with an identification of facts. This section prepares, according to their works, some concepts on analogical reasoning necessary for later discussion.

For programs P_1 and P_2, let U_i be the Herbrand universe for P_i, B_i be the Herbrand base for P_i and M_i be the least Herbrand model for P_i.

Definition 4 Let P_1 and P_2 be programs. We call a finite subset φ of $U_1 \times U_2$ a *pairing* of terms. We define the set φ^+ to be the smallest set that satisfies the following conditions:

(a) $\varphi \subseteq \varphi^+$,

(b) $\langle t_i, t_i' \rangle \in \varphi^+ \Rightarrow \langle f(t_1, \ldots, t_n), f(t_1', \ldots, t_n') \rangle \in \varphi^+$,

where f is a function symbol appearing in both P_1 and P_2.

Definition 5 A pairing φ is called a *partial identity* if φ^+ is a one-to-one relation between terms.

Definition 6 For a pairing φ, two ground atoms $\alpha = p(t_1, \ldots, t_n) \in P_1$ and $\alpha' = p(t_1', \ldots, t_n') \in P_2$ are said to be *identified by* φ, denoted by $\alpha \varphi \alpha'$, if $\langle t_i, t' \rangle \in \varphi^+$.

Definition 7 Let φ be a pairing, and A_i be a set of ground atoms. Then, two rules without variables

$$R = (\alpha \leftarrow \beta_1, \ldots, \beta_n), R' = (\alpha' \leftarrow \beta_1', \ldots, \beta_n')$$

are called a (φ, A_1, A_2)-*analogue* if

$$\beta_j \in A_1, \beta_j' \in A_2, \alpha \varphi \alpha', \beta_j \varphi \beta_j' \qquad (1 \leq j \leq n).$$

Definition 8 For a pairing φ, we define a set M_i^* for $i = 1, 2$.

$M_i^* = \bigcup_n M_i^n$

$M_i^0 = M_i$

$M_i^{n+1} = \{\alpha \mid R_i^n \cup M_i \cup P_i \vdash \alpha\}$

$$R_i^n = \left\{ R' = \alpha' \leftarrow \beta_1', \ldots, \beta_n' \;\middle|\; \begin{array}{l} \text{There exists the instantiation } R \text{ of} \\ \text{a rule in } D_j\ (i \neq j) \text{ such that} \\ R \text{ and } R' \text{ are a } (\varphi, A_1, A_2)\text{-analogue.} \end{array} \right\}.$$

Definition 9 For a pairing φ, the set $PAIR(\varphi)$ of rules is defined as follows:

(a) $t \sim t'$, for each $\langle t, t' \rangle \in \varphi$,

(b) $f(X_1, \ldots, X_n) \sim f(Y_1, \ldots, Y_n) \leftarrow X_1 \sim Y_1, \ldots, X_n \sim Y_n$,

where $\sim$ is a predicate symbol which does not appear in P_1 and P_2 either, and f is a function symbol appearing in both P_1 and P_2.

Proposition 2 The following conditions are equivalent:

(a) $\langle t, t' \rangle \in \varphi^+$.

(b) $PAIR(\varphi) \vdash t \sim t'$.

Definition 10 A *copy* of a program P_i, denoted by $copy(P_i)$, is defined as follows:

$$copy(P_i) = \{(A)_i \leftarrow (B_1)_i, \ldots, (B_n)_i \mid A \leftarrow B_1, \ldots, B_n \in P_i\},$$

where $(A)_i = p_i(t_1, \ldots, t_n)$ if $A = p(t_1, \ldots, t_n)$.

Definition 11 Let P_1, P_2 be programs, and φ be a pairing. The *analogical union* of P_1 and P_2, denoted by $P_1 \varphi P_2$, is defined as follows:

$$P_1 \varphi P_2 = copy(P_1) \cup copy(P_2) \cup trans(P_1) \cup trans(P_2) \cup PAIR(\varphi),$$

where

$$trans(P_2) = \left\{ \begin{array}{l|l} p_2(W_1, \ldots, W_n) \leftarrow \ldots, & p(t_1, \ldots, t_n) \leftarrow \ldots, \\ \quad t_1 \sim W_1, \ldots, t_n \sim W_n, & \quad q(s_1, \ldots, s_k), \\ \quad q_2(V_1, \ldots, V_k), q_1(s_1, \ldots, s_k), & \quad \ldots \\ \quad s_1 \sim V_1, \ldots, s_k \sim V_k, \ldots & \quad \in P_1 \end{array} \right\},$$

$$trans(P_1) = \left\{ \begin{array}{l|l} p_1(W_1, \ldots, W_n) \leftarrow \ldots, & p(t_1, \ldots, t_n) \leftarrow \ldots, \\ \quad W_1 \sim t_1, \ldots, W_n \sim t_n, & \quad q(s_1, \ldots, s_k), \\ \quad q_1(V_1, \ldots, V_k), q_2(s_1, \ldots, s_k), & \quad \ldots \\ \quad V_1 \sim s_1, \ldots, V_k \sim s_k, \ldots & \quad \in P_2 \end{array} \right\}.$$

Proposition 3 Let $p(t_1, \ldots, t_n)$ an atom. The following conditions are equivalent:

(a) $p(t_1, \ldots, t_n) \in M_i^*$.

(b) $P_1 \varphi P_2 \vdash p_i(t_1, \ldots, t_n)$.

4 EBG by Analogical Reasoning

EBG assumes that domain theory is sufficient to prove that the inferred generalizations follow deductively from what the learner already knows[12]. In this section, we take off this assumption, and define a notion of EBG with an incomplete domain theory.

We consider two objects S_1 and S_2 with analogous training examples and incomplete domain theories. First, we present the analogy φ. Then, using the analogical union on

φ, we make up rules which are needed in domain theories. In this way, we construct EBG by analogical reasoning.

Let D_i be a domain theory for S_i, T_i be a training example on S_i, G_i be a goal concept for S_i, O_i be an operationality criterion for S_i, $P_i = D_i \cup T_i$ be a program for S_i and $I_i = (P_i, G_i, O_i)$ be an input of EBG by analogical reasoning.

In this paper, we regard an analogy φ as a pairing of P_1 and P_2.

Definition 12 A *reformation* of a program P_i, denoted by $P_i(\varphi)$, is defined as follows:

$$P_i(\varphi) = P_i \cup ref(P_i),$$

where

$$ref(P_1) = \left\{ \begin{array}{l|l} p(W_1, \ldots, W_n) \leftarrow \ldots, & p_1(W_1, \ldots, W_n) \leftarrow \ldots, \\ \quad q(V_1, \ldots, V_k), & \quad W_1 \sim t_1, \ldots, W_n \sim t_n, \\ \quad \ldots & \quad q_1(V_1, \ldots, V_k), q_2(s_1, \ldots, s_k), \\ & \quad V_1 \sim s_1, \ldots, V_k \sim s_k, \ldots \\ & \quad \in trans(P_1) \end{array} \right\},$$

$$ref(P_2) = \left\{ \begin{array}{l|l} p(W_1, \ldots, W_n) \leftarrow \ldots, & p_2(W_1, \ldots, W_n) \leftarrow \ldots, \\ \quad q(V_1, \ldots, V_k), & \quad t_1 \sim W_1, \ldots, t_n \sim W_n, \\ \quad \ldots & \quad q_2(V_1, \ldots, V_k), q_1(s_1, \ldots, s_k), \\ & \quad s_1 \sim V_1, \ldots, s_k \sim V_k, \ldots \\ & \quad \in trans(P_2) \end{array} \right\}.$$

Definition 13 For inputs $I_1 = (P_1, G_1, O_1)$ and $I_2 = (P_2, G_2, O_2)$, let φ be a pairing of P_1 and P_2. An *explanation tree* EX_i for I_i is a finite tree satisfying the following conditions:

(a) Each node of the tree is an element of M_i^*.

(b) The root node is an instantiation of G_i.

(c) If a node α in the tree has children $\beta_1, \ldots, \beta_n$ $(n \geq 1)$ in this order, then $\alpha \leftarrow \beta_1, \ldots, \beta_n$ is an instantiation of a rule in $P_i(\varphi)$.

(d) Nodes in $\Pi(O_i)$ have no children.

In order to use rules in a generalized derivation GE_i, we number the rules which are used in an explanation tree EX_i, in the same way as in EBG.

(a) Each node which is not an element of $\Pi(O)$ is numbered with the depth-first search.

(b) Let α be a node which has children $\beta_1, \ldots, \beta_n$ $(n \geq 1)$ in this order and C be a rule in $P_i(\varphi)$ which is a generalization of $\alpha \leftarrow \beta_1, \ldots, \beta_n$. If α has a number j, then C is denoted by C_j.

Definition 14 For inputs $I_1 = (P_1, G_1, O_1)$ and $I_2 = (P_2, G_2, O_2)$, let φ be a pairing of P_1 and P_2. A *generalized derivation* GE_i for I_i is an SLD-derivation which consists of a finite sequence $(\leftarrow (G_i)_0) = (\leftarrow G_i), \leftarrow (G_i)_1, \ldots, \leftarrow (G_i)_n$ of goals, a sequence $C_1, \ldots, C_n$ of rules in $P_i(\varphi)$, which are given in EX_i, and a sequence $\theta_1, \ldots, \theta_n$ of most general unifiers such that each $(G_i)_{k+1}$ is derived from $(G_i)_k$ and C_{k+1} using θ_{k+1}.

A generalized derivation GE_i depends on an explanation tree EX_i. In EBG by analogical reasoning, an *explanation* is to construct an explanation tree EX_i, and *generalization* is to generalize a sufficient conditions of a goal concept using a generalized derivation GE_i.

Definition 15 For inputs $I_1 = (P_1, G_1, O_1)$ and $I_2 = (P_2, G_2, O_2)$, let φ be a pairing of P_1 and P_2. EBG *by analogical reasoning* (EBG by AR, for short) is to derive a general definition R_i of a goal concept G_i from an input I_i by an explanation and a generalization.

It is denoted by

$$I_i \overset{EBG}{\rightarrow}_{\varphi} R_i,$$

where

$$R_i = \begin{cases} G & (n = 0) \\ G \leftarrow (G_i)_n & (n \geq 1), \end{cases}$$

and $(\leftarrow (G_i)_n)$ is the goal of the generalized derivation of depth n.

Proposition 4 For inputs $I_1 = (P_1, G_1, O_1)$ and $I_2 = (P_2, G_2, O_2)$, let φ be a pairing of P_1 and P_2. Then, $I_i \overset{EBG}{\rightarrow}_{\varphi} G_i$ iff $G_i \in \Pi(O_i)$.

proof. $I_i \overset{EBG}{\rightarrow}_{\varphi} G_i$
iff there exists an explanation tree of depth 1 with the root G_i
iff $G_i \in \Pi(O_i)$. □

Theorem 2 *For inputs $I_1 = (P_1, G_1, O_1)$ and $I_2 = (P_2, G_2, O_2)$, let φ be a pairing of P_1 and P_2. Let R_i be $G_i \leftarrow A_1, \ldots, A_m$ $(m \geq 1)$, where A_j are positive literals. Then, $I_i \overset{EBG}{\rightarrow}_{\varphi} R_i$ iff the following conditions hold:*

(a) There exists a $(P_i(\varphi))_r = \{(B \leftarrow B_1, \ldots, B_n) \in P_i(\varphi) \mid B \notin \Pi(O_i), n \geq 1\}$ such that $(P_i(\varphi))_r \vdash R_i$.

(b) There exists a substitution θ such that $\{R_i\} \cup \Pi(O_i)|_{M_i^*} \vdash G_i\theta$.

(c) $G_i \notin \Pi(O_i)$.

proof. ($\Rightarrow$) Suppose $I_i \overset{EBG}{\rightarrow} R_i$. Let $C_1, \ldots, C_n$ be the sequence of rules of the generalized derivation of depth n. Then, $\{C_1\}\cup, \cdots, \cup\{C_n\} \vdash R_i$. Furthermore, $C_1, \ldots, C_n \in (P_i(\varphi))_r$. Hence, $(P_i(\varphi))_r \vdash R_i$.

Since $A_1, \ldots, A_m \in \Pi(O_i)$, there exists a substitution θ such that $A_j\theta \in \Pi(O_i)|_{M_i^*}$. Hence, $R_i \cup \Pi(O_i)|_{M_i^*} \vdash G_i\theta$.

($\Leftarrow$) Suppose that the conditions (a),(b) and (c) hold. By (c) there exists no substitution γ such that $\Pi(O_i) \vdash G_i\gamma$. Then, by (a) and (b), there exists a substitution θ such that $A_j\theta \in \Pi(O_i)|_{M_i^*}$. Hence, there exists an explanation tree with the root $G_i\theta$ and the leaves $A_1\theta \ldots, A_m\theta$. Thus, we have $I \overset{EBG}{\rightarrow} R_i$. □

Thus, EBG by AR is a deductive reasoning from a reformation of a program which satisfies input conditions. A reformation of a program depends on φ.

5 A Realization of EBG by AR

The EBG by AR system is a unification of the EBG system and the analogical reasoning system. The unified system carries out EBG and analogical reasoning simultaneously. If there exists missing rules in domain theory, the system carries out EBG. If not, it carries out making up missing rules by means of analogical reasoning and EBG.

Unifiability of terms is essential for a realization of EBG by AR. Hence, we require a pairing φ to be a partial identity.

Now we present a definition of EBG by AR for Prolog programs according to [2, 5, 8, 11].

```
(C1) reason_ebg(Goal,GenGoal,Leaves,Pairing,Wa):-
       r_ebg(Goal,GenGoal,Leaves,Wa),reason(Goal,[ ],Pairing,Wa),!.
(C2) r_ebg(A,GA,L,Wa):-
       prolog(A,Wa),ebg(A,GA,L),!.
(C3) r_ebg(A,GA,L,Wa):-
       reason_ana_ebg(A,GA,L,Wa).
(C4) reason_ana_ebg(Leaf,GenLeaf,GenLeaf,Wa):-
       operational(Leaf),!,prolog(Leaf,Wa).
(C5) reason_ana_ebg((A,As),(G,Gs),(L,Ls),Wa):-
       r_ebg(A,G,L,Wa),r_ebg(As,Gs,Ls,Wa).
(C6) reason_ana_ebg(A,GA,L,Wa):-
       fact(Wa(GA:-GAs)),copy((GA:-GAs),(A:-As)),r_ebg(As,GAs,L,Wa).
(C7) reason_ana_ebg(A,GA,L,Wa):-
       prematch(A,(GB:-GBs),Wa),world(Wa,Wb),reason_B((GB:-GBs),Wb),
       make_clause(Wa-Wb,(GA:-GAs),(GB:-GBs),Pair),compact(Pair,Pair1),
       epic(Pair1),consistency(Pair1,Wa),copy((GA:-GAs),(A:-As)),
       r_ebg(As,GAs,L,Wa),!.
```

First suppose that a fact of assertion A and a rule $C \leftarrow B_1, \ldots, B_n$ $(n \geq 1)$ in P_i are represented by the following Prolog clauses:

$$fact(w_i, (A \leftarrow true)).$$

$$fact(w_i, (C \leftarrow B_1, \ldots, B_n)).$$

respectively, and stored in Prolog database, where w_i is a world name for an object S_i.

The predicate **reason_ebg** takes an instantiation of a goal concept as its first argument, a goal concept as its second argument, and a world name as its fifth argument, and returns a general definition of a goal concept in the third argument, and a partial identity of P_1 and P_2 in the fourth argument.

The predicate **ebg** carries out EBG. The predicate **reason** carries out analogical reasoning. The ordered set of clauses (C2) and (C3) carries out EBG, if possible. The ordered set of clauses (C4), (C5) and (C6) works as a pure-Prolog interpreter. The clause (C7) is a significant rule of EBG by AR. The part from **prematch** to **copy** in the body of (C7) carries out making up rules which are needed in domain theories.

Finally we present an example of EBG by AR.

Example

$W1$ consists of the following clauses:

Goal concept G_1: `ggf(X,Z)`

Domain Theory D_1:
```
ggf(X,Z):-gf1(X,Y),gf2(Y,Z).
gf(X,Z):-p(X,Y),f(Y,Z).
```

Training Example T_1:
```
f(b,c).        m(a,b).
f(d,e).        m(c,d).
```

Operationality Criterion O_1: `operational(G):-member(G,[f(_,_),m(_,_)]).`

$W2$ consists of the following clauses:

Goal concept G_2: `ggf(X,Z)`

Domain Theory D_2:
```
gf1(X,Y):-gf(X,Y).     p(X,Y):-f(X,Y).
gf2(X,Y):-gf(X,Y).     p(X,Y):-m(X,Y).
```

Training Example T_2:
```
f(bb,cc).        m(aa,bb).
f(dd,ee).        m(cc,dd).
```

Operationality Criterion O_2: `operational(G):-member(G,[f(_,_),m(_,_)]).`

The question to our EBG by AR system is

```
?- reason_ebg(ggf(aa,ee),ggf(X,Y),Leaves,Bind,W2).
```

The answers from the system are

```
X = X,
Y = Y,
Leaves = m(X,_18717), f(_18717,_15781), m(_15781,_21719), f(_21719,Y)
Bind = [a-aa,b-bb,c-cc,e-ee,d-dd]
```

Figure 2 shows that we can not construct explanation tree for $W2$ in the usual EBG system but can construct it in our EBG by AR system. It is the same with $W1$.

6 Conclusion

We have given a formalization of EBG. We have shown that EBG is a deductive reasoning from the program which satisfies the input conditions. As a result, we have given a formalization of EBG by AR and its realization, by unifying EBG and analogical

reasoning. Our EBG by AR is a method to carry out EBG even when domain theories are incomplete.

The analogy treated in the present paper is a pairing of terms. We can generalize our EBG by AR for an analogy defined by a pairing of function symbol and predicate symbol[17].

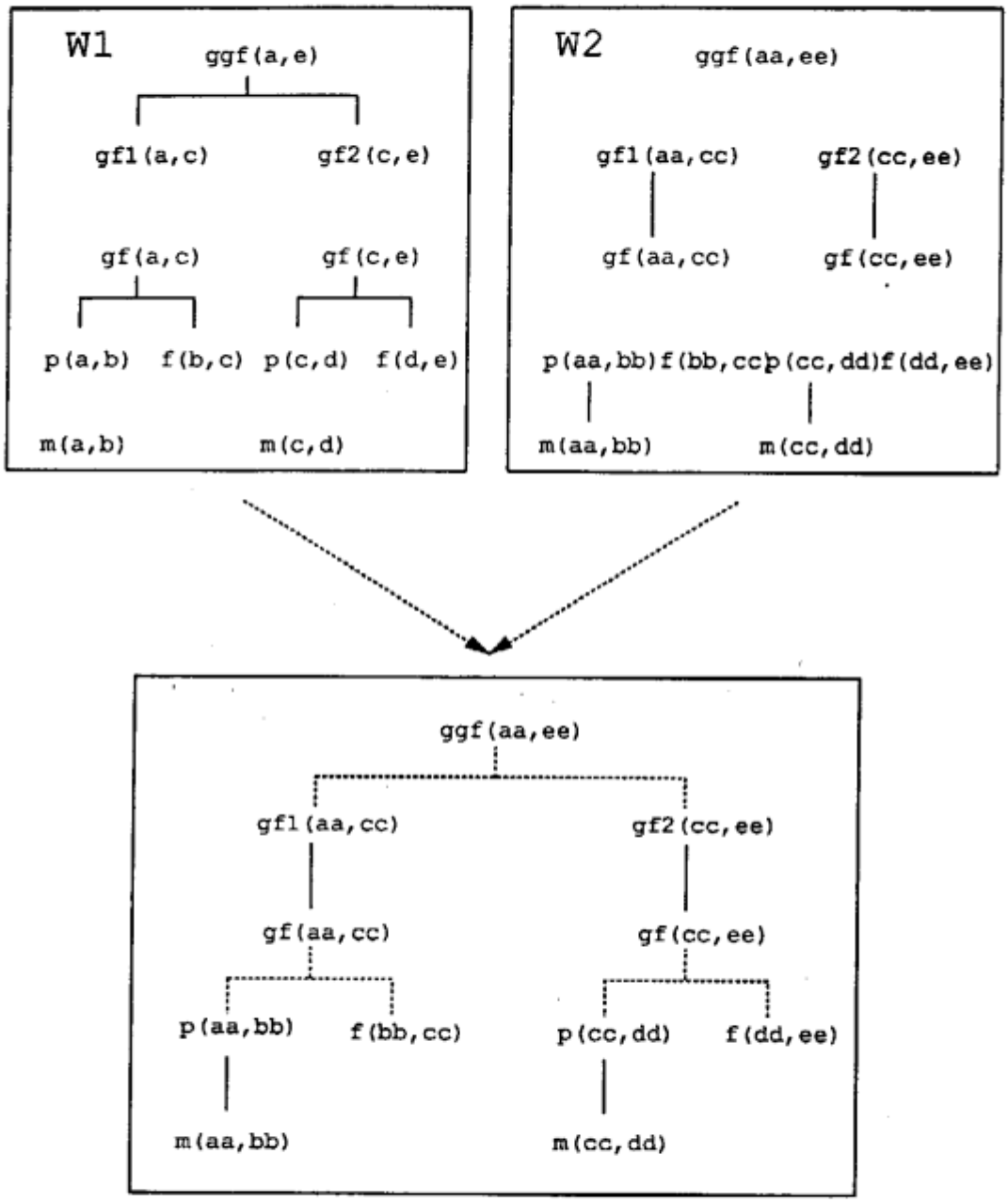


Figure 2: Explanation of W2 using EBG by AR.

References

[1] Ali, K.M.: Augmenting Domain Theory for Explanation-Based Generalization, *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 40–42 (1989).

[2] Arikawa, S. and Haraguchi, M.: A Theory of Analogical Reasoning, Ohsuga, S. and Saeki, Y., Editors, Knowledge Acquisition and Learning, pp. 221–251, *Ohm-sha*, Tokyo (1987), in Japanese.

[3] Ellman, T.: Explanation-Based Learning: A Survey of Programs and Perspectives, *ACM Computing Surveys*, Vol. 21, No. 2, pp. 163–221 (1989).

[4] Genest, J, Matwin, S, and Plante, B.: Explanation-Based Learning with Incomplete Theories: A Three-Step Approach, *Proceedings of the Seventh International Conference on Machine Learning*, pp. 286–294 (1990).

[5] Haraguchi, M. and Arikawa, S.: A Formulation of Analogical Reasoning and its Realization, *Journal of Japanese Society for Artificial Intelligence*, Vol. 1, No. 1, pp. 132–139 (1986), in Japanese.

[6] Haraguchi, M. and Arikawa, S.: A Foundation of Reasoning by Analogy — Analogical Union of Logic Programs, *Lecture Notes in Computer Science* 264, pp. 58–69 (1987).

[7] Haraguchi, M. and Arikawa, S.: Reasoning by Analogy as a Partial Identity between Models, *Lecture Notes in Computer Science* 265, pp. 61–87 (1987).

[8] van Harmelen, F. and Bundy, A.: Explanation-Based Generalization = Partial Evaluation, *Artificial Intelligence*, Vol. 36, pp. 401–412 (1988).

[9] Koppel, M.: ESBL: an Integrated Method for Learning from Partial Information, *Annals of Mathematics and Artificial Intelligence*, Vol. 4, No.3,4, pp. 323–343 (1991).

[10] Lloyd, J.W.: Foundation of Logic Programming (second edition), *Springer-Verlag* (1987).

[11] Matsubara, H., Hanada, K. and Akibe, S.: Explanation-Based Generalization $\neq$ Partial Computation, *Journal of Japanese Society for Artificial Intelligence*, Vol. 6, No. 2, pp. 276–279 (1991), in Japanese.

[12] Mitchell, T.M., Keller, R.M. and Kedar-Cabelli, S.T.: Explanation-Based Generalization: A Unifying View, *Machine Learning*, Vol. 1, No. 1, pp. 47–80 (1986).

[13] Numao, M.: Explanation-Based Learning — Approaches by using Domain-Specific Knowledge —, *Journal of Japanese Society for Artificial Intelligence*, Vol. 3, No. 6, pp. 704–711 (1988), in Japanese.

[14] Numao, M. and Shimura, M: Analogical Reasoning by Decomposing Explanation Structure, *Journal of Japanese Society for Artificial Intelligence*, Vol. 6, No. 5, pp. 716–724 (1991), in Japanese.

[15] Shavlik, J.W. and Towell, G.G.: Combining Explanation-Based Learning and Artificial Neural Networks, *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 90–92 (1989).

[16] Yamamura, M. and Kobayashi, S.: An Augmented EBL and its Application to the Utility Problem, *Proceedings of Twelfth International Conference on Artificial Intelligence*, pp. 623–629 (1991).

[17] Wakizono, R.: A Study on Analogical Reasoning and Abstraction, Master's thesis, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University (1990), in Japanese.

Fundamental Properties of Inductive Reasoning

Klaus P. Jantke
Technische Hochschule Leipzig
FB Mathematik & Informatik
Karl-Liebknecht-Strasse 132
Leipzig 7030, Germany

Abstract

Inductive reasoning is mainly faced to the problem of processing information which may be incomplete. Therefore, inductive inferences may be somehow speculative. Conclusions drawn have frequently to be considered hypothetical. There is no general way out, except giving up the potential incompleteness and, hence, leaving the whole area of inductive reasoning.

In order to cope with these inherent difficulties of inductive reasoning, one may require desirable properties like consistency of hypotheses. Other seemingly natural properties are motivated by efficiency considerations.

Within the talk, there are introduced and investigated several fundamental properties of inductive reasoning. A couple of results are discussed which illustrate both the importance and the restrictiveness of these properties. It is assumed that these results apply to inductive logic programming directly.

The talk is intended to initiate a deeper and partially new communication between scientists from the inductive logic programming community and those scientists working on other areas of inductive reasoning.

A Personal Approach to Linguistics Oriented Inductive Logic Programming

Gregers Koch

DIKU, Copenhagen University, Denmark.

Abstract

A new concept called Computational Logico-Semantic Induction is introduced. It is a special kind of inductive logic programming, and it may be considered a generalisation of the old concept grammatical inference that may in turn be characterised as a kind of computational syntactic induction. The concept is clarified by an example.

The possibility of automation is discussed in considerable detail. The implementation of computational logico-semantic induction has to do with the construction of a kind of blackbox to accept a traditional syntactic description of a linguistic universe. Besides the blackbox must accept as input a finite set of pairs (Ek,Fk) where Ek is a text from the linguistic universe, and Fk is the intended semantic representation corresponding to the input Ek. For instance, the Fk may be in the form of a logical formula or a logical code. Output from the blackbox should be a program that translates linguistic input E into logical output F in such a way that in particular the input Ek gives the output Fk. Here is required a complete match with the given examples.

Some possible principles for the construction of such a blackbox by logic programming methods are discussed. These principles are clarified by application to a few small sample texts. We conclude that this new concept of computational logico-semantic induction is extraordinarily promising.

Introduction

This paper investigates methods and tools for developing a specific kind of model of human language learning capability, by presenting a performative simulation model (here termed a computational logico-semantic induction system [16, 18]).

The same methods and tools may be applied for the purpose of implementing a wide variety of computational systems including certain kinds of rule-based expert systems and certain kinds of modern grammars (in particular the so-called unification grammars) [17].

The advantage of logico-semantic induction is its applicability in the context of constructing natural language interfaces as well as a variety of other user-friendly types of interfaces to expert systems and other computer systems.

We are studying the problem of constructing language acquisition models from specific data. That is, we could be claimed to be modelling an extremely advanced type of information processing systems, viz. human beings in the role of acquiring language capabilities. However, we are modelling the performative aspects only. No claim whatsoever is made as to the possible descriptive power of the resulting models from a psychological point of view (so we might call it purely antropomorphic information technology).

The focus of this paper is on logico-semantic induction which is a method for the systematic pattern identification and extraction in linguistic data sequences,

in particular at a semantic and a combined syntactic and logical level of interpretation. It provides a means for the automated analysis of verbal protocols, and it constitutes a method for the automated construction of a logico-semantic parser.

Logico-Semantic Induction and its automated variant Computational Logico-Semantic Induction designate a completely new method from the area of logic programming and natural language processing. In contrast to the majority of other inductive approaches the method here does not deal with induction in a space of possible assertions but instead with induction in a space of possible logico-semantic representations. Here is given a short introduction to the concepts. A more comprehensive discussion by this author may be found elsewhere.

The particular kind of inductive inference that we have in mind may be illustrated by means of a diagram. Along the first axis we shall map all the possible assertions or utterances (in some suitable encoding), and along the second axis we shall map all possible representations within the framework of a particular representational notation (and similarly in a suitable encoding). A semantic theory will then occur in the shape of a mapping from the axis of utterances into the axis of representations (as long as we presuppose unambiguity, otherwise it will be generalised to a relation).

For example, we might from the following facts

crow number 1 is black
crow number 2 is black
crow number 3 is black
etc.

make the attempt to induce the following more general assertion: [2]

all crows are black.

The type of induction advocated here is of a different kind: From the following conventions

text E1 has the logico-semantic representation F1
text E2 has the logico-semantic representation F2
text E3 has the logico-semantic representation F3
etc.

we should like to find a (possibly very limited) linguistic universe L and a (logical) program P such that for each text e in L its corresponding logico-semantic representation f is the result (output) of executing the program P with the given e as input. Here the example texts E1, E2, E3 etc. are all included in the linguistic universe L.

Computational Logico-Semantic Induction may be considered a generalisation of the old concept grammatical inference that may be characterised as a kind of computational syntactic induction [11].

The possibility of automation is discussed in considerable detail. The implementation of computational semantic induction has to do with the construction

of a kind of blackbox to accept a traditional syntactic description of a linguistic universe. Besides the blackbox must accept as input a finite set of pairs $<e_k,f_k>$ where e_k is a text from the linguistic universe, and f_k is the intended semantic representation corresponding to the input e_k. For instance, the f_k may be in the form of a logical formula or a logical code. Output from the blackbox should be a program that translates linguistic input e into logical output f where especially the input e_k gives the output f_k. Here is required a complete match with the given examples.

Some possible principles for such a blackbox are discussed. These principles are clarified by application to a few small sample texts. We conclude that this new concept of computational logico-semantic induction is extraordinarily promising.

This paper contains a brief discussion and sketches a solution. A more comprehensive discussion is in preparation [13, 14, 16].

Here we are concerned exclusively with parsing or textual analysis. Analogous considerations can be made concerning textual synthesis or generation.

This work on computational logico-semantic induction was performed under heavy influence by some of the leading approaches within logic grammars like those of A. Colmerauer [3, 4], V. Dahl [7, 8, 9], F. Pereira [23, 24, 25], P. Saint-Dizier [27, 28], and M. McCord [21, 22].

It may really be seen as an attempt to unify some rather diverging tendencies in the philosophy of language, namely Creswell's lambda-calculatoric theory [5, 6] and some montagovian ones [19, 10], and on the other hand, the first order logical theories from logic grammars [12, 16]. The contribution here seems to support any of these theories.

As an example we may investigate the following English sentence

(1) Mary believes that Peter loved a woman

Within the limits of a modestly extended first order predicate calculus we may assign to the sentence the following two interpretations or logico-semantic representations, respectively:

(2) ∃y[woman(y) & believe(pres,mary,love(past,peter,y))]

(3) believe(pres,mary,∃y[woman(y) & love(past,peter,y)])

An absolutely central problem of semantics (here called the logico-semantic problem) is to assign to each input text from the appropriate linguistic universe one or several formalized semantic representations. As formalizations we will here consider only logical formulae belonging to some particular logical calculus (like definite clauses or Horn clauses, first order predicate logic, some extended first order predicate logics, the lambda calculi, and Montague's intensional logic [19]).

The principles of implementation are quite clear and fairly well developed, as may be seen by studying the example below (another example may be found in [16]). But as far as an actual implementation is concerned, we are working on it albeit in a rather slow pace (due to lack of resources).

A Small Example

Now time is probably ripe to investigate the example mentioned above. This may be seen as a further development of the ideas discussed in [16]. If the syntactic description is the following little grammar

(4) Sent → Np Vp
Np → Det Noun | Prop
Vp → Tv Np | Vp-s that S

(in the last production rule we have used a categorial grammar notation) then we may look for a representative, also called an exhaustive text. Such an exhaustive sample text may be the following:

Mary believes that Peter loved a woman

Within the chosen semantic representational notation (a predicate calculus of arbitrary high order, PC_ω) we may prefer to use a kind of generalised quantifiers for representing some (two) possible interpretations of the sample text in the following way:

(5) a(y,woman(y),believe(pres,mary,love(past,peter,y)))

(6) believe(pres,mary,a(y,woman(y),love(past,peter,y)))

The two interpretations deviate by one having as a presupposition the existence of such a female and the other not having that presupposition. Montague grammars like PTQ would obtain the same distinction.

If we choose to consider the first formula (5) to be the intended representation, the method here will lead in a mechanical fashion to the logic program shown below (7), written in the form of a logic grammar.

The program constitutes just a syntactical description augmented with attributes or decorations as may be seen by ignoring the functional arguments (then quite simply the grammar of (4) occurs).

Let us see what happens more precisely in our method. The intended resulting formula (5) should be represented as a tree structure like that in figure 1. Then the following steps should be performed:

Step 1: Enumerate the boxes in the intended result structure. (In our example this means that the boxes will get the numbers from 1 to 7, as in figure 1).

Step 2: Construct the syntactic structure (by performing parsing or syntactic analysis).

Step 3: Create a match between the result structure and the syntactic structure. More precisely, make a connection from a numbered box in the result structure to the lexical category in the syntax structure to which the word (lexical or syncategorematic) belongs. This is an indication of the vertex in the syntax tree where that fraction of the result structure

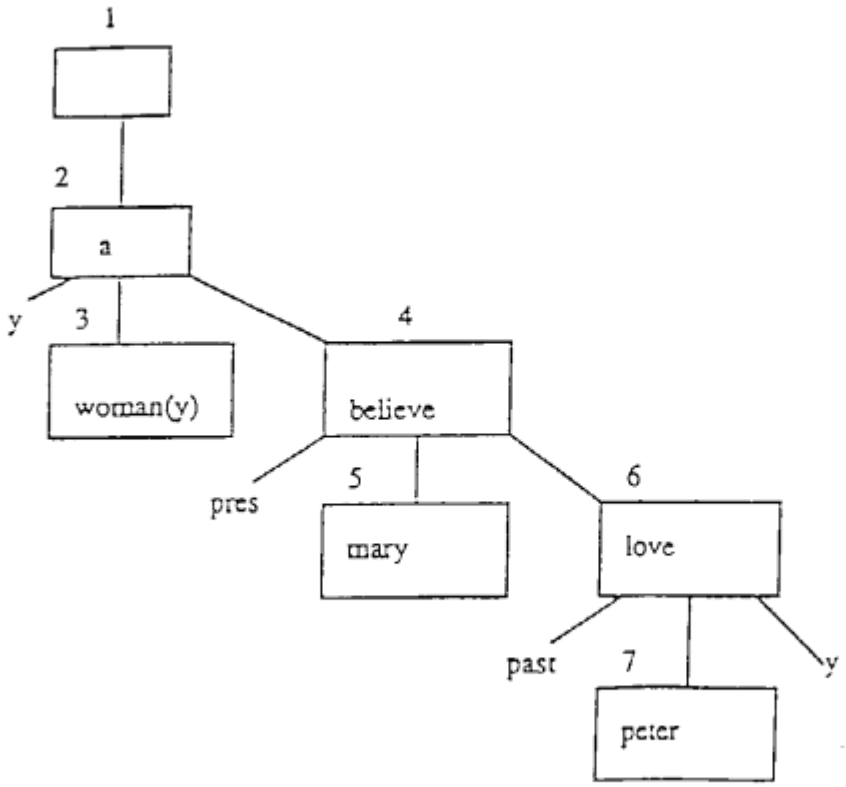


Figure 1:

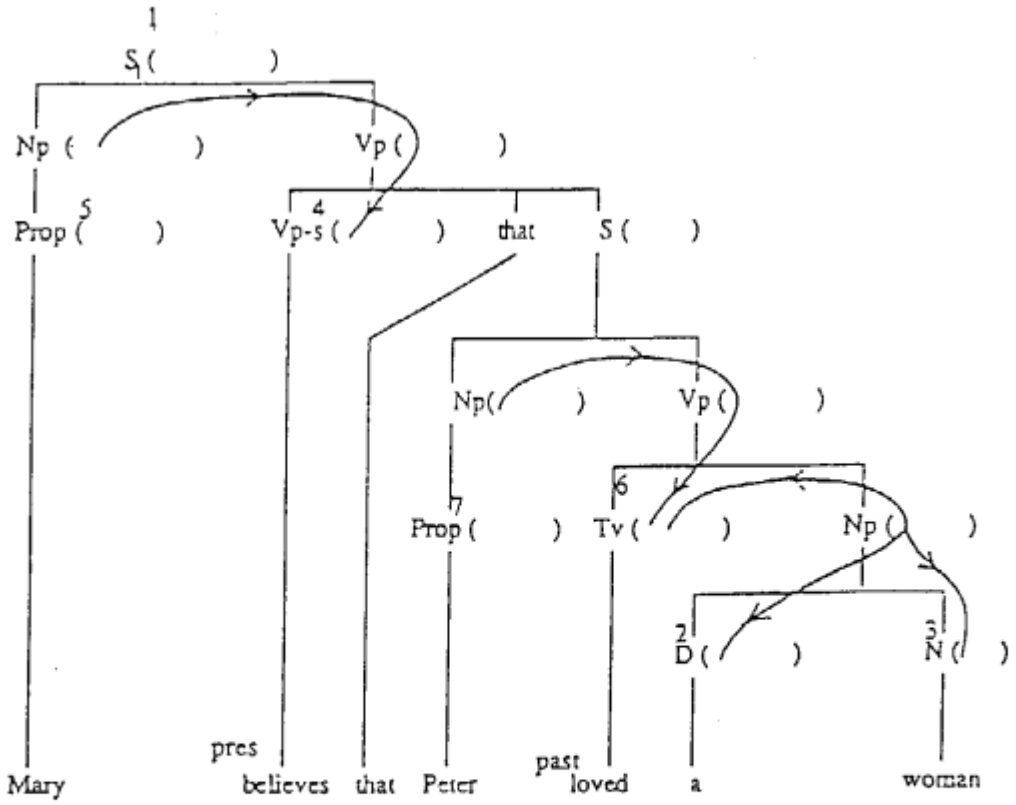


Figure 2:

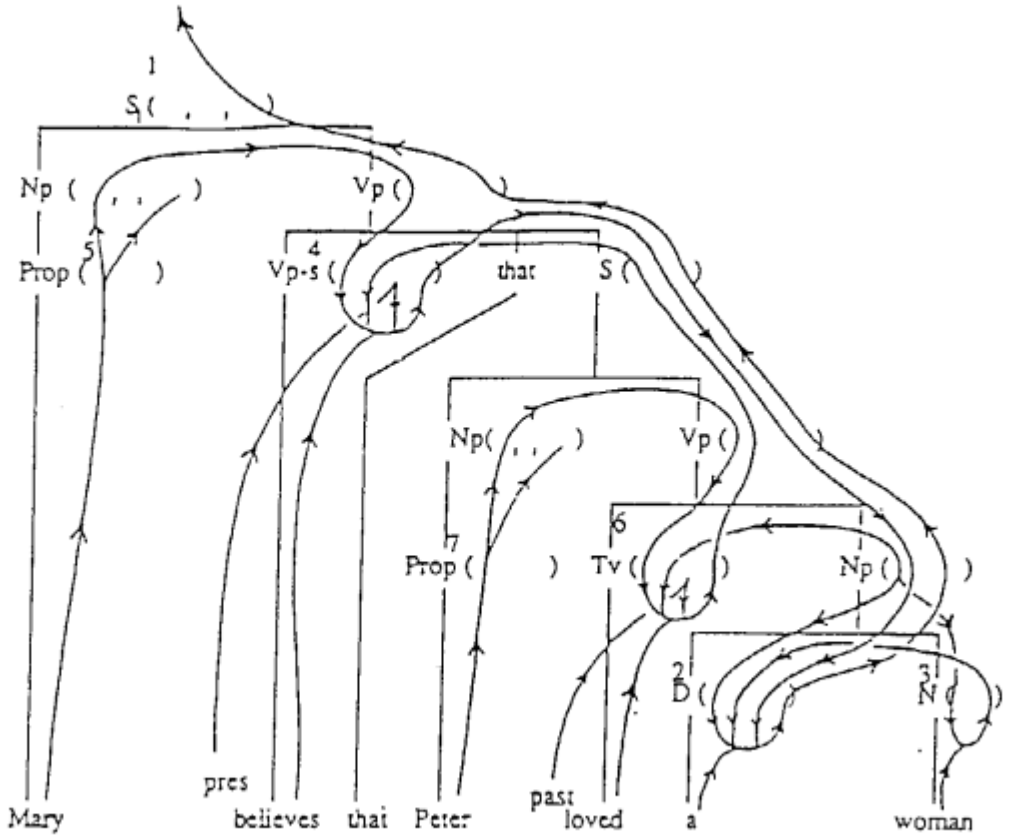


Figure 3:

having the relevant word as its top vertex, is being constructed as the result attribute of the vertex.

Step 4: Construct the flow from the so-called focus variables (form a new variable for each Np phrase, as in figure 2).

Step 5: Construct the flow in the lexical rules.

Step 6: Connect each pair of numbers corresponding to an edge in the result structure (here the tree structure should be respected, as in figure 3 where the following pairs are connected: 7–6, 5–4, 6–4, 3–2, 4-2, 2–1).

Step 7: Check the consistency concerning arity and local flow.

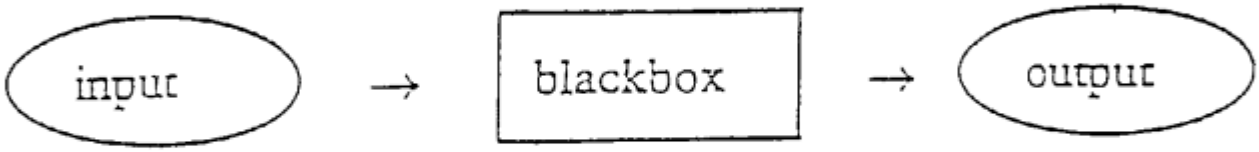
In our example the augmented syntactic structure will be like figure 3.

The resulting logic grammar will be the following:

(7) S(V,W,U) → Np(X,Y,Z),Vp(X,W,V,U)
Np(X,Y,Z) → Prop(X)
Np(X,Z,W) → D(X,Y,Z,W),N(X,Y)
Vp(Y,X1,Y1,V) → Vp-s(X,Y,Z,W),[that],S(W,Z,V)
Vp(Y,W,V,U) → Tv(X,Y,Z,W),Np(Z,V,U)
D(X,Y,Z,a(X,Y,Z)) → [a]
D(X,Y,Z,every(X,Y,Z)) → [every]

Concluding Remarks and Perspectives

As to which representation languages are acceptable with respect to this method, there seems to be a high degree of freedom so that we seem to be near the implementation of a general information theoretical or computer science paradigm like this:



Anyway, there exists a requirement that a kind of homomorphy property, a kind of compositionality should be available in the relationship between input and output. One or another variant of Frege's principle of compositionality should be obtained:

To the extent that our rules are of the form

$$p_0(G(y_1,\ldots,y_n)) \rightarrow p_1(y_1),\ldots,p_n(y_n)$$

we know about the semantic representation function Sem that

$$Sem(P_0)=G(Sem(P_1),\ldots,Sem(P_n))$$

where $$P_0=P_1 \frown P_2 \frown \ldots \frown P_n$$

provided that P_k is the fragment of the input text belonging to the syntax category p_k for all $k \in \{0,1,..,n\}$.

And this property is precisely one way of expressing Fregean compositionality.

One perspective of this approach is that it allows a generalisation into what we tend to call computational logico-semantic abstraction [18]. In this context it is profitable to make use of certain results from the modern computer science disciplines of logic programming, attribute grammars, and denotational semantic theories.

Another perspective concerns automated learning. Computational logico-semantic induction has the property that the system will be able to improve its linguistic performance (i.e., handling new information of a semantic nature) by adoption from a single occurrence of a grammatical rule. That must be effective automated learning par excellence!

So, besides concluding that the method of logico-semantic induction is not only new but also promising we are able to discuss AI-problems related to inductive learning from the following perspective: inductive reasoning as a way of managing linguistic information in logical systems. Hence in this case it is not really a question of empirical information, and of course its relationship to AI is always arguable (what is the precise content of AI?), but a surprisingly high degree of automated learning is actually obtainable.

References

[1] J.W. Bresnan [ed.]. 1982. *The Mental Representation of Grammatical Relations.* MIT Press.

[2] E. Charniak & D. McDermott. 1985. *Introduction to Artificial Intelligence.* Addison-Wesley.

[3] A. Colmerauer. 1978. Metamorphosis Grammars. L. Bolc [ed.]. *Natural Language Communication with Computers.* 133–189. Lecture Notes in Computer Science No. 63.

[4] A. Colmerauer. 1982. An Interesting Subset of Natural Language. K.L. Clark & S.-Å. Tärnlund [eds.]. *Logic Programming.* Academic Press.

[5] M.J. Cresswell. 1973. *Logics and Languages.* Methuen.

[6] M.J. Cresswell. 1985. *Structured Meanings, the Semantics of Propositional Attitudes.* MIT Press.

[7] V. Dahl. 1979. *Logical Design of Deductive Natural Language Consultable Data Bases.* Proc. Fifth International Conference on Very Large Data Bases. Rio de Janeiro, Brazil.

[8] V. Dahl. 1979. *Quantification in a Three-Valued Logic for Natural Language Question-Answering Systems.* Proc. IJCAI. Tokyo, Japan.

[9] V. Dahl & M. McCord. 1983. *Treating Coordination in Logic Grammars.* Am. Journ. Comp. Ling.

[10] D.R. Dowty, R.E. Wall & S. Peters. 1981. *Introduction to Montague Semantics.* D. Reidel.

[11] J.J. Horning. 1969. *A Study of Grammatical Inference.* Technical Report No. CS 139. Computer Science Department, Stanford University.

[12] G. Koch. 1985. *Who is a Fallible Greek in Logic Grammars.* 54–77 in [15].

[13] G. Koch. 1986. *Relating Definite Clause Grammars and Montague Grammars.* Institute of Computer Science, Technical University of Denmark. 20 pages.

[14] G. Koch. 1986. *The Application of Prolog for the Translation into a Semantic Representation.* Proc. Nordic Seminar on Machine Translation. EUROTRA-DK. Copenhagen. 175–189.

[15] G. Koch [ed.]. 1985. *Fifth Generation Programming vol. 1: Logic Programming in Natural Language Analysis.* Proceedings of Workshop in Copenhagen. Dec. 1984. DIKU report 85/2.

[16] G. Koch. 1987. *Automating the Semantic Component.* Information Processing Letters 24. 299–305.

[17] G. Koch. 1987. *LFG og Prolog.* Institute for Applied and Mathematical Linguistics, Copenhagen University.

[18] G. Koch. [Forthcoming]. *A Technical Perspective on Expert Systems, Modern Grammars, Semantic Abstraction, and their Implementations.* Proc. Fifth Symposyum on Empirical Foundations of Information and Software Science, Risø, Denmark, Nov. 1987.

[19] R. Montague. 1974. *The Proper Treatment of Quantification in Ordinary English.* In [20].

[20] R. Montague. 1974. *Formal Philosophy.* Yale University Press.

[21] M. McCord. 1982. Using Slots and Modifiers in Logic Grammars for Natural Language. *Artificial Intelligence*. 18,3:327–367.

[22] M. McCord. 1987. Natural Language Processing in Prolog. A. Walker [ed.]. *Knowledge Systems and Prolog*. Addison-Wesley.

[23] F.C.N. Pereira. 1983. *Logic for Natural Language Analysis*. SRI International. Technical Note 275.

[24] F.C.N. Pereira & D.H.D. Warren. 1983. *Parsing as Deduction*. SRI Technical Report 293. Stanford Research Institute.

[25] F. Pereira & D. Warren. 1980. Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*. 13,3:231–278.

[26] U. Reyle & W. Frey. 1983. *A Prolog Implementation of Lexical-Functional Grammar*. Proc. IJCAI, International Joint Conference on Artificial Intelligence. 693–695.

[27] P. Saint-Dizier. 1985. *On Syntax and Semantics of Modifiers in Natural Language Sentences*. In [15].

[28] P. Saint-Dizier. 1986. An Approach to Natural Language Semantics in Logic Programming. *The Journal of Logic Programming*. 3(4):329–356.

Institute of Datalogy
University of Copenhagen
DK 2100 Ø
Copenhagen, Denmark

Distinguishing exceptions from noise in non-monotonic learning

Ashwin Srinivasan,
Stephen Muggleton,
Michael Bain
The Turing Institute,
36 North Hanover Street,
Glasgow G1 2AD,
UK.

April 20, 1992

Abstract

It is important for a learning program to have a reliable method of deciding whether to treat errors as noise or to include them as exceptions within a growing first-order theory. We explore the use of an information-theoretic measure to decide this problem within the non-monotonic learning framework defined by Closed-World-Specialisation. The approach adopted uses a model that consists of a reference Turing machine which accepts an encoding of a theory and proofs on its input tape and generates the observed data on the output tape. Within this model, the theory is said to "compress" data if the length of the input tape is shorter than that of the output tape. Data found to be incompressible are deemed to be "noise". We use this feature to implement a compression-guided specialisation procedure that searches for the best-fitting theory for the data (that is, the one with the shortest input tape length). The approach is empirically evaluated on the standard Inductive Logic Programming problem of learning classification rules for the KRK chess end-game.

1 Introduction

Induction is an uncertain process. Scientific theories are ascribed various degrees of belief depending on how well they agree with known facts. As new information becomes available certain hypotheses may seem more likely and others less so. For instance consider the Julian calendar in which leap years were held to be necessary once every 4 years. This can be represented in Prolog with negation by failure as

normal(Year) :- year(Year), not leap4(Year).
leap4(Year) :- modulo(Year,4,0).

This rule is correct up to around one part in a hundred and so up until 1582 errors could simply be treated as noise. However after 1500 years the mismatch with astronomical measurements forced a revision of the calendar under Pope Gregory XIII. In the Gregorian calendar the rules can be written as follows.

leap4(Year) :- modulo(Year,4,0), not leap100(Year).
leap100(Year) :- modulo(Year,100,0), not leap400(Year).
leap400(Year) :- modulo(Year,400,0).

The aim of Inductive Logic Programming (ILP) is to automate the construction and revision of logical theories by using example facts and background knowledge [17]. In the case above, examples are of the form *normal(1581)* or *not(normal(1582))* and background knowledge would contain a definition of *modulo*. ILP methods based on closed-world specialisation [3] would progressively specialise the overgeneral clause *normal(Year)* by inventing (and generalising) new abnormality predicates (corresponding to *leap4*, *leap100* and *leap400*). This process is capable of generating the Gregorian calendar theory and has recently been used to construct a complete and correct solution for the standard KRK illegality problem from the machine learning literature [2]. However, a key issue remains to be addressed: there is no mechanism by which a non-monotonic learning strategy can reliably distinguish true exceptions from noise. For example, a strategy based on closed-world-specialisation would continue specialising until a correct theory is obtained. In noisy domains, this will necessarily result in fitting the noise. In this paper we explore the possibility of using a general information-theoretic model developed in [18, 21] to help distinguish noise from true exceptions. An important consequence of adopting this model is that theories found to be "compressive" (described below) are, with very high probability, significant. A simple search procedure is developed to find as compressive an explanation as possible for the data. Its results are evaluated empirically for the standard ILP problem of learning classification rules in the KRK chess end-game.

2 Information-Theoretic Evaluation of Hypotheses

In the 1950's Carnap [7] and others suggested "confirmation theories" aimed at providing a statistical underpinning to the problem of the plausibility of inductive inferences. Various difficulties and paradoxes were encountered with these approaches which meant that they were never applied within machine learning programs [16]. Instead, confidence in alternative hypotheses has for the most part relied on either *ad hoc* notions of simplicity (the Occam's razor principle) or on statistical tests of significance based on the coverage of a rule and prior probability estimates of the classes present in the data ([9, 10]).

The choice of the most compact hypothesis is the basis of Rissanen's "Minimal Description Length" (MDL) principle[25]. This states that the best theory for explaining a set of data is one which minimises the sum of:

1. the description length of the theory in bits and
2. the description length of the data when encoded using the theory.

Within machine learning the MDL principle has been applied by [11] to determine the the best sampling rate for character recognition and by [24] to the problem of learning decision trees. However, its application to first-order learning remains largely unexplored. It forms the motivation for the encoding measure used in [23]. However, the simplifications result in a number of problems (identified in [10]). Muggleton [18] addresses this issue using a model related to algorithmic information theory ([26, 13, 8]). In his approach, the significance of a hypothesis is evaluated by comparing the length of the input and output tapes of a reference Turing machine. The components to be minimised in the MDL approach are represented on the input tape as a Horn clause theory and a proof specification. The latter specifies how the examples on the

output tape are to be derived using the theory and background knowledge. (Figure 1: from [21]). A theory is deemed significant if the length of the input tape (in bits) is shorter than that

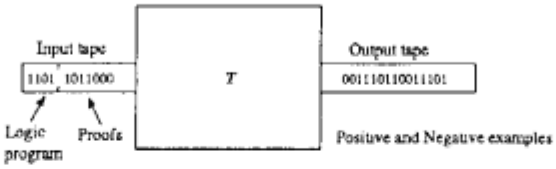


Figure 1: A Turing machine model for learning logic programs

of the output tape (the theory and proofs are said to compress the data). This model has been empirically evaluated in [21] and shown to be better suited to learning first-order theories than the statistical measure used in [9, 10]. In the following section we describe a method of using the compression obtained from this approach to guide the progressive correction of first-order theories within a non-monotonic framework.

3 Compression-Based Non-Monotonic Induction

We incorporate the Turing machine compression model in Figure 1 within the non-monotonic learning framework developed by Muggleton and Bain ([3, 2]). Their technique commences with an over-general logic program. This is then progressively corrected by a hierarchical decomposition strategy. At each level negated exception predicates are introduced (and generalised) to account for exceptions. Figure 2 shows an algorithm that performs the alternate operations of specialisation and generalisation characteristic of closed-world-specialisation.

It is worth noting here that:

1. As in [15], there is an assumption that the exceptions to a rule are fewer than the examples that satisfy it.

2. The call to *generalise* results in an attempt to induce a (possibly over-general) rule by a learning algorithm.

3. All rules are added to the theory. Further, all negative examples covered by an over-general clause are taken to be exceptions and the clause is specialised with a (new) abnormality predicate.

Each correction performed by the CWS algorithm is an attempt to improve the accuracy of the theory, at the expense of increasing its size. Clearly, if the correction was worthwhile, the gain in accuracy should outweigh the penalty incurred in increasing the theory size. In encoding terms, each correction increases the theory encoding on the input tape and decreases the proof encoding. In the model in Figure 1, a net decrease in the length of the input tape occurs when the correction succeeds in identifying some pattern in the errors (that is, the errors are not noise). The new theory consequently compresses the data further by exploiting this pattern. Using this feature, we evaluate the utility of updating a theory by checking for an increase in compression. We note the following consequences of using the compression model within the non-monotonic framework adopted:

1. Only compressive theories are deemed to be reliable in the model. Thus, while we can adopt the MDL principle of selecting the theory with the shortest input tape, we can be

```
start:
  PosE = positive examples of target concept
  NegE = negative examples of target concept
  return learn(PosE,NegE)

learn(Pos,Neg):
  ClauseList = []
  repeat
    C = generalise(Pos,Neg)
    if C ≠ []
      PosC = positive examples covered by C
      NegC = negative examples covered by C
      Pos = Pos - PosC
      Neg = Neg - NegC
      ClauseList = ClauseList + (C,PosC,NegC)
  until C = []

  Theory = []
  foreach (Clause,PosC,NegC) in ClauseList
    if |NegC| ≠ 0
      Theory = Theory + specialise(Clause,PosC,NegC)
    else
      Theory = Theory + Clause
  return Theory

specialise(HornClause,Pos,Neg):
  hd(V1, ..., Vn) = head of HornClause
  Body = body of HornClause
  ab = a new predicate symbol
  SpecialisedClause = hd(V1, ..., Vn) ← Body, not(ab(V1, ..., Vn))
  PosE = positive examples of ab formed from Neg
  NegE = negative examples of ab formed from Pos
  return SpecialisedClause + learn(PosE,NegE)
```

Figure 2: Non-monotonic inductive inference using closed-world-specialisation (CWS)

confident of not having fitted the noise only if the theory itself is compressive. Stated differently, we can be confident that highly compressive theories have avoided fitting the noise as much as possible.

2. With the closed-world assumption, all examples are covered. Consequently, the output tape has to be encoded only once. Input tapes for alternate theories are compared against this encoding.

3. Consider an over-general clause in the current theory. The proof encoding described in [21] ensures all variables of the clause are bound to ground terms. Specialising this clause involves adding a negated literal to its body. By appending this literal to the body, we are guaranteed that it will be ground. This ensures safety of the standard Prolog computation rule used by the Turing machine.

4. The proof encoding for each example has two parts: a choice-point specification and a proof tag. Since the negative literal appended to a clause can never create bindings, the choice-point specification remains unaltered. The size-accuracy trade-off referred to earlier therefore reduces to a trade-off between increasing theory size and decreasing tag size. Not having to recalculate the choice-point encoding for each specialisation is a major benefit as this is an extremely costly exercise.

While the aim is to obtain the most compressive subset of the clauses produced by the CWS algorithm, it is unnecessary to examine all subsets since clauses constructed as generalisations of an abnormality predicate cannot be considered independent of the parent over-general clause. For example, it makes no sense to consider the following set of clauses for explaining leap years:

normal(Year) :- year(Year), not leap4(Year).
leap400(Year) :- modulo(Year,400,0).

Despite this, there may still be an intractably large number of clause-sets to consider. Consequently, we adopt a greedy strategy of selecting clauses in order of those that give the most gain (in compression). This strategy has to confront two important issues: devising a reliable method of deciding on the "best" clause to add to the theory and the fact that adding this clause may not produce an immediate increase in compression.

A simple way to address the first problem is to select the clause that corrects the most errors. Since decreasing errors is the only way to shorten the input tape, the gains are larger for theories that make fewer errors. This works well if all clauses are of approximately the same descriptional complexity. A better estimate would account for the complexity of individual clauses as well. This can be done using average estimates of the cost of encoding predicates, functions and variables. In the experiments in the next section, this more sophisticated estimate has proved unnecessary. This is because the clauses fitting noisy data tend to correct fewer errors and therefore, considered later using the simpler estimate. For the other clauses, the gain from correcting errors dominates the loss from increased theory size.

To address the problem of local minima, it is clearly desirable to have a method of looking ahead to see if a (currently non-compressive) clause will be part of the final theory. To decide this, we calculate an estimate of the compression produced by the most accurate theory containing the clause. The clause is retained if this expected compression is better than the maximum achieved so far. Each time an actual increase in compression is produced, the theory is updated with all clauses that have been retained. Figure 3 shows how the estimate is calculated. The estimated compression will usually be optimistic because it it assumes that all errors can be

```
estimate(Theory):
  Ncorrect = number of examples correctly classified by Theory
  Nmaximum = number of examples that the learner can hope to classify correctly
  Outbits = length of output tape (in bits)
  OldTheory = length of Theory (in bits)
  OldTags = current length of correction tags (in bits)
  Choices = length of choice-point encoding (in bits)

  NewTheory = OldTheory × Nmaximum / Ncorrect
  NewTags = length of correction tags to correctly classify Nmaximum examples
  EstInbits = NewTheory + Choices + NewTags
  return (Outbits - EstInbits)
```

Figure 3: Estimating the compression from a theory

compressed. However within the compression model adopted, it is extremely unlikely to get any more compression from a theory that is completely correct on noisy data than from an incorrect one that leaves the noise uncompressed. Of course, one way to guarantee an optimistic estimate is to assume that there will be no increase in theory size (as opposed to the current scaled estimate). However, this gives no heuristic power and usually only prolongs a futile search for a correct theory. Figure 4 summarises the main steps in the compression-based selection of clauses as described here. The following points deserve attention:

1. At any given stage, only some clauses produced by CWS are candidates to be added to the theory (recall the earlier statement that over-general clauses have to be considered before their specialisations).

2. The "best" clause refers to the clause selected using the simple error-count measure, or the more sophisticated one that accounts for the estimated theory increase. To obtain the latter requires a knowledge of the number of predicate, function and variable symbols in the clause.

3. Consider the situation when the estimated compression from adding the "best" clause is no better than the compression already obtained. Figure 4 does not acknowledge the possibility that some of the other clauses can do better. It is possible to rectify this by progressively trying the "next best" clause until all clauses have been tried.

4. The procedure in Figure 4 is reminiscent of post-pruning in zero-order algorithms (the clauses are constructed first and then possibly discarded). A natural question that arises is whether it is possible to incorporate the compression measure within the specialisation process The analogy to zero-order learning algorithms is whether tree pre-pruning is feasible. The answer is yes, and in practice may be preferred as it avoids inducing all clauses. The price to pay is that it may not be possible to estimate reliably the utility of a clause.

4 Empirical Evaluation

We evaluate the utility of using compression as a reliable noise detector on the standard ILP problem of learning classification rules for the KRK chess endgame [19]. However, contrary to

```
start:
  ClauseList = clauses produced by CWS
  return select_clauses(ClauseList)

select_clauses(ClauseList):
  Theory = PartialTheory = []
  Compression = 0
  repeat
    PotentialClauses = clauses in ClauseList that can be added to theory
    C = "best" clause in PotentialClauses
    if C ≠ []
      PartialTheory = PartialTheory + C
      NewCompression = compression of PartialTheory
      if NewCompression > Compression
        Theory = PartialTheory
        Compression = NewCompression
      else
        EstCompression = estimate(PartialTheory)
        if EstCompression ≤ Compression return Theory
  until C = []
  return Theory
```

Figure 4: Compression-based selection of clauses produced by CWS

normal practice, we chose to learn rules for KRK-legality (as opposed to KRK-illegality). This provides an extra level of exceptions for the specialisation method. Given background knowledge of the predicates *lt/2* and *adj/2*, Figure 5 shows the target theory. It is possible to achieve an accuracy of about 99.6% without accounting for the second level of exceptions. In fact, the theory shown in Figure 6 is about 98% correct.

For our experiments, we adopt a simple noise model termed the *Classification Noise Process* (CNP) [1]. In this, a noise of η implies that (independently) for each example, the sign of the example is reversed with probability η. This is not the only random noise process possible. For example, a noise of η in our model corresponds to a class-value noise of 2η in that adopted by [22] and Donald Michie (private communication) advocates a process that preserves the underlying distribution of positive and negative examples. Finally, although the procedure described in Figure 4 is not dependent on any particular induction algorithm, the results quoted here use Golem ([20]).

Figure 7 tabulates the percentage accuracy of the most compressive theory for different noise levels. Here "accuracy" refers to accuracy on an independent (noise-free) test set of 10000 examples. Since the compression model only guarantees reliability for compressive theories, nothing can be said about those for which compression is less than 0 (irrespective of their accuracy on the test set). In Figure 7, an entry of "_" denotes that the theory obtained is non-compressive on the training data and consequently, no claim is made regarding its accuracy on the test set. The results highlight some important points. Compressive theories do appear to avoid fitting the noise to a large extent. The price for this reliability is reflected in the amount of data required. In comparison, it is possible that other techniques may require fewer examples. However, they either require various parameters to be set ([10]), use *ad hoc* constraints ([23]) or need an additional

```
% legal(WK_file, WK_rank, WR_file, WR_rank, BK_file, BK_rank)
legal(A,B,C,D,E,F) :- not ab00(A,B,C,D,E,F).
ab00(A,B,C,D,C,E) :- not ab11(A,B,C,D,C,E).
ab00(A,B,C,D,E,D) :- not ab12(A,B,C,D,E,D).
ab00(A,B,C,D,E,F) :- adj(A,E), adj(B,F).
ab00(A,B,A,B,C,D).
ab12(A,B,C,B,D,B) :- lt(A,D), lt(C,A).
ab12(A,B,C,B,D,B) :- lt(A,C), lt(D,A).
ab11(A,B,A,C,A,D) :- lt(B,D), lt(C,B).
ab11(A,B,A,C,A,D) :- lt(B,C), lt(D,B).
```

Figure 5: A complete and correct theory for KRK-legality

```
legal(A,B,C,D,E,F) :- not ab00(A,B,C,D,E,F).
ab00(A,B,C,D,C,E).
ab00(A,B,C,D,E,D).
ab00(A,B,C,D,E,F) :- adj(A,E), adj(B,F).
```

Figure 6: An "approximately correct" theory for KRK-legality

data set for pruning ([6]). Further, most of them are unable to offer any guarantee of reliability (the approach followed in [10] can select clauses above a user-set significance threshold). In this respect, our empirical results mirror PAC ([27]) results for learning with noisy data in propositional domains ([1]): with increasing noise, more examples are needed to obtain a good theory. It is also worth noting that the conditions covered by the second level of exceptions (the cases in which the White King is in between the White Rook and Black King) occur less than 4 times in every 1000 examples. This is only picked up in the noise-free data set of 10000 examples (in which there were 38 examples where the rules applied).

Extending the PAC analogy further, Figure 8 shows the results from a different perspective. For different levels of noise, this figure shows the number of training examples required for the "approximately correct" theory of Figure 6 to be compressive. For example, at least 170 examples are required to obtain a compressive theory that is 98% accurate on noise-free data. While these numbers are approximate (they are obtained by extrapolating the compression produced by the theory for the different training sets in Figure 7) they do indicate the general trend of requiring larger example sets for increasing noise levels.

5 Conclusion

The task of distinguishing between exceptions and noise is an issue that is typically ignored in the literature on non-monotonic reasoning. It is, however, of fundamental importance for a learning program that has to construct theories using real-world data. One way to approach the problem is to see if the exceptions to the current theory exhibit a pattern. The compression model

Noise (%)	Training Set Size					
	100	250	500	1000	5000	10000
0	-	99.7	99.7	99.7	99.7	100
5	-	98.1	98.1	99.7	99.7	99.7
10	-	-	98.1	98.1	99.7	99.7
15	-	-	98.1	98.1	99.7	99.7
20	-	-	-	98.1	99.7	99.7
30	-	-	-	-	98.1	98.1
40	-	-	-	-	-	98.1

Figure 7: Test-set accuracy for the most compressive theory

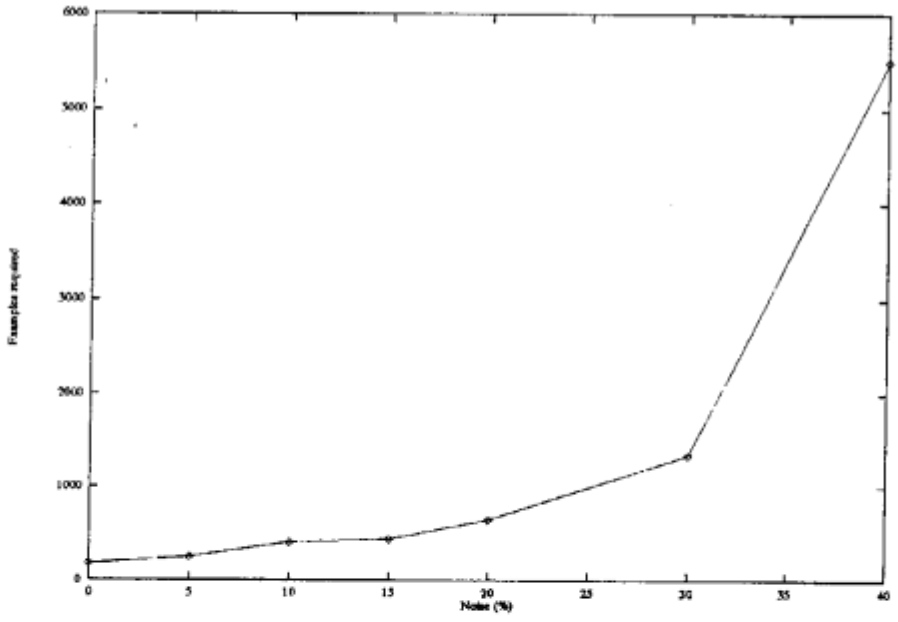


Figure 8: Examples required for a 98% correct and compressive theory

we have used in this paper uses an information-based approach to check whether the pattern detected warrants specialising the theory. While it can be formulated as an implementation of the Minimal Description Length principle, more significant is the fact that theories found to be compressive in the model are unlikely to have detected chance patterns. Our empirical results suggest that by selecting the most compressive theory, it is possible (given enough data) to reliably avoid fitting most of the noise. Clearly, it would be desirable to confirm these results with controlled experiments in other domains. In practice, the method has found interesting rules on an independent problem of pharmaceutical drug design ([12]). Finally, the results also lend support to the link between compressive theories for first-order concepts and their PAC-learnability. While various authors have shown such a connection exists ([4, 5, 14]), it would be nice to show that their concept of compression fits that used here.

Acknowledgements. The authors would like to thank Donald Michie and the ILP group at the Turing Institute for their helpful discussions and advice. This work was carried out at the Turing Institute and was supported by the Esprit Basic Research Action project ECOLES, the IED's Temporal Databases and Planning project and the SERC Rule-Base Systems Project. Stephen Muggleton is supported by an SERC post-doctoral fellowship.

References

[1] D. Angluin and P. Laird. Learning from noisy examples. *Machine Learning*, 2(4):343–370, 1988.

[2] M. Bain. Experiments in non-monotonic learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 380–384, San Mateo, CA, 1991. Morgan Kaufmann.

[3] M. Bain and S. Muggleton. Non-monotonic learning. In D. Michie, editor, *Machine Intelligence 12*. Oxford University Press, 1991.

[4] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 273–282, 1986.

[5] R. Board and L. Pitt. On the necessity of occam algorithms. Uiucdcs-r-89-1544, University of Illinois at Urbana-Champaign, 1989.

[6] C.A. Brunk and M.J. Pazzani. An investigation of noise-tolerant relational concept learning algorithms. In L. Birnbaum and G.C. Collins, editors, *Proceedings of the Eighth International Workshop on Machine Learning*, San Mateo, 1991. Morgan Kaufmann.

[7] R. Carnap. *The Continuum of Inductive Methods*. Chicago University, Chicago, 1952.

[8] G. Chaitin. *Information, Randomness and Incompleteness - Papers on Algorithmic Information Theory*. World Scientific Press, Singapore, 1987.

[9] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–283, 1989.

[10] S. Dzeroski. *Handling Noise in Inductive Logic Programming*. University of Ljubljana, (M.Sc. Thesis), Ljubljana, 1991.

[11] Q. Gao and M.Li. An application of minimum description length principle to online recognition of handprinted numerals. In *IJCAI-89*, Detroit, MI, 1989. Kaufmann.

[12] R. King, S. Muggleton, and M.J.E. Sternberg. Drug design by machine learning. *submitted to Journal of the National Academy of Sciences*, 1991.

[13] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Prob. Inf. Trans.*, 1:1–7, 1965.

[14] M. Li and P.M.B. Vitanyi. Inductive reasoning and Kolmogorov complexity. In *Proceedings of the Fourth Annual IEEE Structure in Complexity Theory Conference*, pages 165–185, 1989.

[15] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[16] H. Mortimer. *The Logic of Induction.* Ellis Horwood, Chichester, England, 1988.

[17] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[18] S.H. Muggleton. A strategy for constructing new predicates in first order logic. In *Proceedings of the Third European Working Session on Learning*, pages 123–130. Pitman, 1988.

[19] S.H. Muggleton, M.E. Bain, J. Hayes-Michie, and D. Michie. An experimental comparison of human and machine learning formalisms. In *Proceedings of the Sixth International Workshop on Machine Learning*. Kaufmann, 1989.

[20] S.H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.

[21] S.H. Muggleton, A. Srinivasan, and M.E. Bain. Compression, significance and accuracy. *to appear: International Machine Learning Conference*, 1992.

[22] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[23] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[24] J.R. Quinlan and R.L. Rivest. Inferring decision trees using the Minimum Description Length principle. *Information and Computation*, 80:227–248, 1989.

[25] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[26] R.J. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:376–388, 1964.

[27] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

Implementations, Experiments and Applications

Handling noise in Inductive Logic Programming

Sašo Džeroski *
Institut Jožef Stefan
Jamova 39, 61111 Ljubljana, Slovenia
and
The Turing Institute
36 North Hannover Street, Glasgow G1 2AD, Scotland, UK

Ivan Bratko
University of Ljubljana, Faculty of Electrical Engineering and Computer Science
Tržaska 25, 61111 Ljubljana, Slovenia
and
Institut Jožef Stefan
Jamova 39, 61111 Ljubljana, Slovenia

Abstract

As the field of inductive logic programming matures to the stage of being applicable to practical problems, the question of handling imperfect data becomes increasingly more important. We have developed the ILP system mFOIL, which is based on the FOIL approach and uses Bayesian probability estimates to handle imperfect data. The paper presents mFOIL with its noise-handling mechanisms and discusses its performance on the benchmark problem of learning illegal chess endgame positions from noisy examples. It also describes the application of mFOIL to the more practical problems of learning rules for finite element mesh design and learning qualitative models of dynamic systems.

*Sašo Džeroski is currently visiting The Turing Institute and is supported by a British Council scholarship. Most of the research presented in this paper was financially supported by the Slovene Ministry of Science and Technology. Part of the work presented was done at the Turing Institute.

1 Introduction

As the field of inductive logic programming (ILP, Muggleton 91) matures to the stage of being applicable to practical problems, the issue of handling imperfect data becomes increasingly more important. Namely, learning in real-life domains often means learning from imperfect data. The various kinds of data imperfections include: random errors (*noise*) in training examples, too sparse training examples (from which it is difficult to reliably detect correlations), and unsuitable concept language (which depends on the background knowledge and may not facilitate the formulation of an exact description of the target concept).

Systems which successfully deal with imperfect data usually have a single mechanism, called *noise-handling* mechanism, to cope with all three kinds of imperfection mentioned above. FOIL (Quinlan 90) includes a noise-handling mechanism based on a simple scheme of encoding the induced clauses. However, several problems with this mechanism have been identified (Džeroski 91, Džeroski and Lavrač 91, Lavrač and Džeroski 92) stemming from the defficiencies of the encoding scheme used (Muggleton, Srinivasan and Bain 92). LINUS (Lavrač, Džeroski and Grobelnik 91) was shown to perform better than FOIL on the problem of learning illegal positions in a chess endgame from noisy examples (Džeroski and Lavrač 91, Lavrač and Džeroski 92). As LINUS transforms the ILP problem to propositional form and then uses attribute-value learning systems, it relies on the noise-handling mechanisms of the latter.

In the mFOIL system, described in this paper, noise-handling techniques used in attribute-value systems are directly incorporated within the FOIL approach. The noise-handling techniques include Bayesian probability estimates, such as the Laplace estimate and the m-estimate (Cestnik 90, 91), and significance based stopping criteria. Section 2 presents mFOIL in detail, including its search space, search heuristics and stopping criteria. In Section 3, the performance of mFOIL, FOIL and LINUS is compared on the benchmark ILP problem of learning illegal chess endgame positions from noisy examples. The application of mFOIL, FOIL and GOLEM to the problem of finite element mesh design is described in Section 4. Finally, Section 5 gives a brief description of the application of mFOIL to the problem of learning a qualitative model of the dynamic system of coupled containers (U-tube).

2 Handling noise in mFOIL

mFOIL is largely based on the FOIL approach. There are, however, several important differences. As far as the search space is concerned, mFOIL uses some additional information, such as the symmetry and different variables (rectified literals) constraints, to reduce it. The main difference is that, instead of the entropy-based information gain heuristic, a more direct error estimate is used as a search heuristic. This may be the Laplace estimate or the more sophisticated m-estimate, which takes into acount prior probabilities of the positive ($\oplus$) and negative ($\ominus$) examples. In addition, mFOIL uses a beam-search strategy as opposed to the hill-climbing search used in FOIL. Finally, a stopping criterion based on statistical significance, similar to the one used in CN2 (Clark and Niblett 89), has been implemented in mFOIL.

Similar to LINUS and unlike FOIL and GOLEM (Muggleton and Feng 90), mFOIL can use non-ground background knowledge. On the other hand, several simplifications have been made as compared to FOIL. First, unlike FOIL (Quinlan 91), mFOIL handles determinate literals as any other literal. Second, instead of using partial orderings to prevent infinitely recursive clauses, a very simple constraint is used. Namely, a clause $Head \leftarrow Body$ must not contain a literal identical to $Head$ in its body, which does not always prevent infinite recursion. Finally, while FOIL is implemented in C, mFOIL is implemented in *prolog* (Quintus Prolog) and is thus fairly slow.

2.1 Search space

The search space of mFOIL is determined by the predicates from the background knowledge and their corresponding declarations. The background knowledge may take the form of a *prolog* program, which may contain non-ground and non-unit clauses, as in LINUS. FOIL and GOLEM, on the other hand, can use only ground facts as background knowledge. Background predicates have to be defined in such a way that each call to a predicate always instantiates (binds) all of its unbound output arguments.

The training examples have the form of gound facts. Structured terms may appear in training examples and background knowledge, but are treated as constants by mFOIL. This in fact means that $f(a,b)$ and $f(c,d)$ may only be generalized to a variable A, and not to a structured term $f(X,Y)$, in the clauses generated by mFOIL. To enable generalizations of the latter form, the transformations of flattening and unflattening (Rouveirol 90) may be used, which transform function symbols to predicate symbols and vice versa.

For each predicate in the background knowledge the types of its arguments have to be specified. Type restrictions are taken into account when constructing possible literals. In order to further reduce the search space of possible literals, additional information from the declarations is used, including input/output modes (designation of arguments as input/output), possible symmetries of predicates and rectification of literals.

Input/output modes, similar to the ones in GOLEM and FOIL are used in mFOIL. An input argument is processed so that a new variable cannot be introduced in its place. Either a new or an old variable may be put in place on an output argument. If no mode declaration is given for a predicate, mFOIL assumes that all of the arguments may be either input or ouput. However, at least one of the variables in a literal must be old, regardless of mode declarations. This constraint is enforced in FOIL as well.

A binary predicate p is symmetric if the truthvalues of *p(X,Y)* and *p(Y,X)* are equal for any binding of X and Y. This means that only one of the literals *p(X,Y)* and *p(Y,X)* need be considered when adding literals to the body of a clause. In general, a predicate may be symmetric in several pairs of arguments.

A constraint that all variables appearing in a literal be different is applied in mFOIL, thus further reducing the number of possible literals arising from a predicate. Namely, it usually turns out that literals like *p(X,X)* are non-discriminating, i.e. have truthvalue *false* (or *true*) for all possible values of X. Thus, they impose an unnecessary computational overhead in the learning process. Literals that do not contain identical variables as arguments are named *rectified* literals. By default, only rectified literals are allowed in mFOIL. This constraint may be eliminated by setting the value of a parameter in mFOIL.

2.2 Heuristics

The main difference between mFOIL and FOIL is that, instead of the entropy-based information gain heuristic in FOIL, a more direct error estimate is used as a search heuristic in mFOIL. This may be the Laplace estimate or the more sophisticated m-estimate (Cestnik 90, 91), both of which have proved to handle noise successfully in attribute-value learning systems (Clark and Boswell 91, Džeroski, Cestnik and Petrovski 92).

If a clause covers n training examples, out of which s are positive, its expected accuracy is estimated as

$$ExpectedAccuracy = \frac{s+1}{n+2}$$

by the Laplace error estimate. The m-estimate of the expected accuracy is given by

$$ExpectedAccuracy = \frac{s + m \times p^+}{n+m}$$

where p^+ is the a priori probability of the class $\oplus$. This may be estimated by relative frequency from the whole training set. By default, the Laplace estimate is used as a search heuristic.

It should be noted that the probability estimates in mFOIL are computed directly from the examples in the original training set, as opposed to the use of a local training set of extended tuples for this purpose in FOIL. This decision is justified by the following argument. Suppose a single noisy training example, erroneously classified as positive, is covered by a clause. This example may be extended to, for instance, ten tuples in the local training set. Estimating the expected accuracy from ten positive tuples would yield a result high enough to be accepted under most conditions. Thus a clause might be built covering a single erroneous example.

The Laplace and m-estimate used as search heuristic allow for a kind of pruning similar to the gain pruning in FOIL (Quinlan 90). This kind of pruning is described here, although it is not yet implemented in mFOIL. Suppose a literal L with new variables is to be added to a clause. If the clause specialized in this way covers s positive examples, the best we can achieve with further refinements is a clause that covers s positive and no negative examples. The heuristic value of such a clause would be $\frac{s+1}{s+2}$ and $\frac{s+m\times p^+}{s+m}$, if estimated by Laplace and the m-estimate respectively. If this value is less than the heuristic value of the best partial clause found so far, no replacement of new variables in L with old will produce a better clause. Consequently, such replacements need not be considered, thus saving considerable computational effort.

In fact, any well behaved heuristic function facilitates this kind of pruning. Suppose that the heuristic value of a clause covering s positive and t negative examples is estimated by a a function $\mathcal{H}(s,t)$. The pruning process described above is than justified if $\frac{\partial\mathcal{H}(s,t)}{\partial s} > 0$ and $\frac{\partial\mathcal{H}(s,t)}{\partial t} < 0$.

2.3 Search strategy and stopping criteria

A covering approach, similar to the one in FOIL, is used in mFOIL. This means that clauses are built repetitively, until one of the stopping criteria is satisfied. After a clause is built, the positive examples covered by it are removed from the training set. The search strategy in mFOIL is beam search, which at least partially alleviates the problem of getting into undesirable local maxima typical for greedy hill-climbing search.

The search for a clause starts with the clause with empty body. During the search, a small set of promising clauses found so far is maintained (the beam), as well as the best *significant* clause found so far. The default size of the beam is five clauses. At each step of the search, the refinements of each clause in the beam (obtained by adding literals to their bodies) are evaluated using the search heuristic, and the best of their improvements constitute the new beam. To enter the new beam a clause has to be *possibly significant*. The search for a clause terminates when the new beam becomes empty (no possibly significant improvements of the clauses in the current beam have been found at the current step). The best significant clause found so far is retained in the concept description if its expected accuracy is better that the default accuracy (the probability of the more frequent of the classes $\oplus$ or $\ominus$), estimated from the entire training set by relative frequency.

The significance test used in mFOIL is similar to the one used in CN2 (Clark and Niblett 89) and is based on the likelihood ratio statistic (Kalbfleish 79). If a clause covers n examples, s of them positive, the value of the statistic can be calculated as follows. Let p^+ and p^- be the prior probabilities of class $\oplus$ and $\ominus$, estimated from the entire training set. In addition, let $q^+ = \frac{s}{n}$ and $q^- = 1 - q^+$. Then we have

$$LikelihoodRatio = 2 \times n \times (q^+ log(\frac{q^+}{p^+}) + q^- log(\frac{q^-}{p^-}))$$

This statistic is distributed approximately as χ^2 with one degree of freedom. If its value is

above a specified significance threshold, the clause is deemed significant. The default significance threshold is 6.64, corresponding to a 99 % level of significance.

A kind of pruning, implemented in mFOIL, is based on the following argument. Suppose a partial clause covers s positive examples. The best we can hope to achieve with refining this clause, is a clause that covers s positive and no negative examples. The likelihood ratio statistic would in this case have the value $-2s \times log(p^+)$. If this value is less than the significance threshold, no significant refinement of this clause can be found. Otherwise, the clause is *possibly significant.*

The search for clauses is terminated when too few positive examples (possibly zero) are left for a generated clause to be significant or when no significant clause can be found with expected accuracy greater than the default. Demanding an expected accuracy (estimated by the heuristic value) higher than the default accuracy may cause constructing no clauses at all in domains where negative examples are prevalent. In such cases, the criterion of accuracy might be ommited and the expected accuracy of the constructed clauses disregarded, provided that they are significant. The search would then terminate when no further significant clause could be constructed. Afterwards, the clauses with very low accuracy might be discarded if desired.

3 Learning illegal chess endgame positions from noisy examples

This section discusses the performance of FOIL and mFOIL on the task of learning illegal chess endgame positions from noisy examples. Various amounts of noise were artificially added to the correct examples in an incremental fashion. mFOIL using the m estimate as search heuristic achieved better classification accuracy than FOIL. FOIL and LINUS were compared earlier on training sets with various amounts of noise, which was added nonincrementally (Džeroski and Lavrač 91). The performance of FOIL and mFOIL is also compared with these results.

The problem of learning illegal positions in the chess endgame domain White King and Rook versus Black King (KRK endgame) is described by Muggleton et al. (89) and Quinlan (90). The target relation *illegal(WKf,WKr,WRf,WRr,BKf,BKr)* states whether the position where the White King is at *(WKf,WKr)*, the White Rook at *(WRf,WRr)* and the Black King at *(BKf,BKr)* is not a legal White-to-move position.

The background knowledge for this problem consists of essentially two relations, *adjacent(X,Y)* and *less_than(X,Y)*, indicating that rank/file X is adjacent to rank/file Y and rank/file X is less than rank/file Y, respectively. According to the type constraints, each of these relations is replaced by two predicates, one for each type of arguments. Thus, the background knowledge consists of the following predicates: *adjacent_file(X,Y)* and *less_file(X,Y)* with arguments of type *file* (with values *a* to *h*), *adjacent_rank(X,Y)* and *less_rank(X,Y)* with arguments of type *rank* (with values *1* to *8*), and equality $X=Y$, used for both types of arguments. The relations *adjacent_file(X,Y)*, *adjacent_rank(X,Y)* and $X=Y$ are *symmetric.*

Experimental setup

We largely followed the experimental design described by Džeroski and Lavrač (91), except that noise was added incrementally. There were five training sets of 100 positions each and one testing set of 5000 positions (Muggleton et al. 89). Three series of experiments were conducted, introducing noise in the training examples in three different ways. First, noise was added in the values of all the arguments of the target relation *illegal*, then in the values of the class variable, $\oplus$ and $\ominus$, and finally, in both the argument and the class variable values.

Each series of experiments was performed using one of the three ways of introducing noise and a chosen noise level. The chosen amount of noise was first introduced into the training sets. Five sets of clauses were then induced, one for each of the five training sets, and their accuracy was tested on the 5000 examples set which remained intact (non-noisy). The accuracies given are the averages of the accuracies of the five clause sets.

In our experiments, x % of noise in argument A means that in x % of examples, the values of argument A were replaced by random values of the same type from a uniform distribution. For example, 5 % of noise in argument A means that its value was changed to a random value in 5 out of 100 examples, independently of noise in other arguments. The class variable was treated as an additional argument when introducing noise. The percentage of noise introduced varied from 5 % to 80 % as follows: 5 %, 10 %, 15 %, 20 %, 30 %, 50 %, and 80 %. Background knowledge and testing examples were noise-free.

In earlier experiments with FOIL and LINUS on the KRK endgame, noise was added to the training examples independently for each percentage, i.e., in a nonincremental fashion. Suppose we had a training set of 100 examples. To add 10 % of noise in the values of argument A, all correct values were taken and 10 of them were randomly selected and corrupted. In the experiments described in this section, noise was added incrementally. First, 5 values were corrupted out of the 100 correct values. Next, out of the 95 remaining nonnoisy values. additional 5 were corrupted to achieve 10 % of noisy values. Five more examples (out of the uncorrupted 90) are replaced with noisy to achieve 15 % of noise in argument A.

A recent version of FOIL (2.1) was used in the experiments. It was run with its default parameters. The parameters in mFOIL were set as follows. The beam size was set to five (default) and the significance threshold was set to zero (no significance was tested). The construction of clauses was stopped when a clause with expected accuracy (estimated by the search heuristic) less than the default accuracy was constructed. The Laplace estimate and the m-estimate, with several different values for m, were used as search heuristics. The values for m were as follows: 0, 0.01, 0.5, 1, 2, 3, 4, 8 and 16.

The three series of experiments with mFOIL were repeated with a significance threshold of 99 %. Finally, some experiments were performed on training examples with nonincremental noise. LINUS using ASSISTANT (Cestnik, Kononenko and Bratko 87), FOIL and mFOIL with $m = 0.01$ were applied in this case to examples corrupted with all three kinds of noise.

Results

The results of the three series of experiments performed with FOIL and mFOIL for different kinds of noise and different values of m are given in Table 3. Noise affected adversely the classification accuracy of both FOIL and mFOIL. The classification accuracy decreased as the percentage of noise increased: the most when it was introduced in both the values of the arguments and the class variable, and the least when introduced in the values of the class variable only.

As noise was introduced incrementally, accuracy dropped monotonically when the amount of noise increased. In earlier experiments with nonincremental noise (Džeroski and Lavrač 91) an increase in accuracy was observed when noise increased from 15 % to 20 %. As such an increase is not present in the results of experiments with incremental noise. we may conclude that the icrease was due to the nonincremental nature of the noise added. and not to some specific properties of the learning systems involved.

It can be noted that even with noise in the arguments only, the achieved classification accuracy drops below the default accuracy (2/3) for high levels of noise. This indicates that the training examples are still being overfitted. To illustrate this phenomenon, we adapt an argument from Clark and Boswell (91). At 100 % noise in the arguments, the arguments an the class are completely independent. The maximally overfitted rules (one per example) would

	Noise							
Error estimate	0	5	10	15	20	30	50	80
Laplace	95.14	90.11	80.41	78.32	73.66	72.57	67.91	59.20
$m = 0.0$	95.53	88.27	82.45	76.50	74.98	72.80	64.27	55.86
$m = 0.01$	95.57	91.87	84.59	80.06	74.95	72.61	68.32	59.10
$m = 0.5$	95.57	91.71	84.04	79.92	76.49	72.42	68.20	58.91
$m = 1.0$	95.33	91.86	81.52	79.54	74.74	73.53	68.54	60.34
$m = 2.0$	94.21	89.53	82.16	77.72	73.52	74.04	68.64	61.21
$m = 3.0$	94.21	88.71	80.84	76.78	72.72	73.06	66.79	60.50
$m = 4.0$	93.78	86.98	79.88	75.66	74.30	72.91	64.96	61.47
$m = 8.0$	86.92	79.44	70.98	72.14	71.50	66.45	65.23	65.40
$m = 16.0$	76.22	76.22	67.72	68.16	66.07	66.30	65.38	66.30
best m	95.57	91.87	84.59	80.06	76.49	74.04	68.64	66.30
FOIL	94.82	91.64	80.45	76.81	73.79	71.19	63.25	57.58

(a) noise in the arguments

	Noise							
Error estimate	0	5	10	15	20	30	50	80
Laplace	95.14	92.71	88.90	86.40	84.58	83.75	73.74	60.25
$m = 0.0$	95.53	94.19	88.86	85.74	85.39	82.19	75.37	55.89
$m = 0.01$	95.57	94.26	92.02	89.96	88.14	86.01	79.20	64.17
$m = 0.5$	95.57	94.26	91.34	89.28	88.37	85.88	78.54	65.04
$m = 1.0$	95.33	94.02	90.78	88.74	87.55	85.12	77.97	64.67
$m = 2.0$	94.21	93.01	89.36	87.76	86.96	84.37	76.94	63.44
$m = 3.0$	94.21	91.80	87.38	86.70	84.39	84.07	76.69	63.87
$m = 4.0$	93.78	92.09	87.73	87.88	84.61	85.66	76.26	64.54
$m = 8.0$	86.92	85.81	81.65	85.60	79.84	80.84	75.61	64.81
$m = 16.0$	76.22	76.22	76.22	78.22	78.02	81.67	76.56	63.96
best m	95.57	94.26	92.02	89.96	88.37	86.01	79.20	65.04
FOIL	94.82	94.23	90.07	88.31	87.53	84.54	71.86	60.65

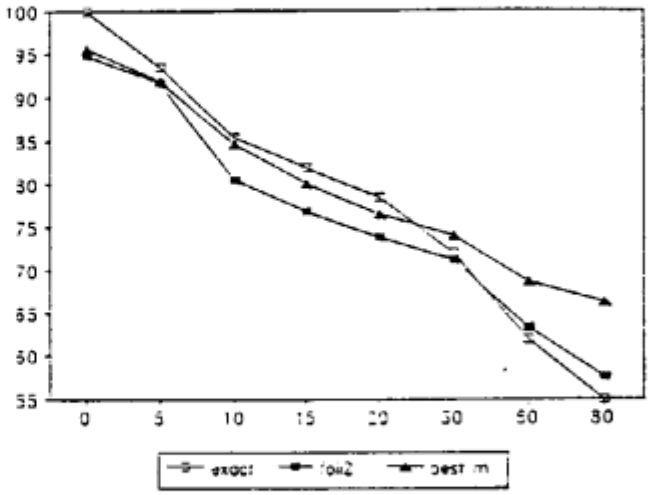
(b) noise in the class variable

	Noise							
Error estimate	0	5	10	15	20	30	50	80
Laplace	95.14	85.36	75.76	75.08	71.69	64.22	61.38	57.58
$m = 0.0$	95.53	85.56	77.92	72.68	71.95	64.64	60.62	56.22
$m = 0.01$	95.57	89.64	80.16	76.58	73.01	68.64	64.53	59.84
$m = 0.5$	95.57	89.62	79.83	75.38	73.09	68.80	66.27	59.65
$m = 1.0$	95.33	88.72	79.17	76.36	74.74	69.65	65.68	58.98
$m = 2.0$	94.21	88.53	79.48	76.10	72.38	65.73	64.42	60.79
$m = 3.0$	94.21	85.74	80.79	73.44	72.53	66.09	65.11	61.73
$m = 4.0$	93.78	85.66	76.05	74.75	71.74	67.18	65.00	60.52
$m = 8.0$	86.92	80.21	72.16	72.33	70.57	67.19	65.14	59.54
$m = 16.0$	76.22	76.22	71.77	71.79	67.79	68.21	62.76	60.24
best m	95.57	89.64	80.79	76.58	74.74	69.65	66.27	61.73
FOIL	94.82	89.28	76.94	72.88	71.02	66.14	61.33	57.88

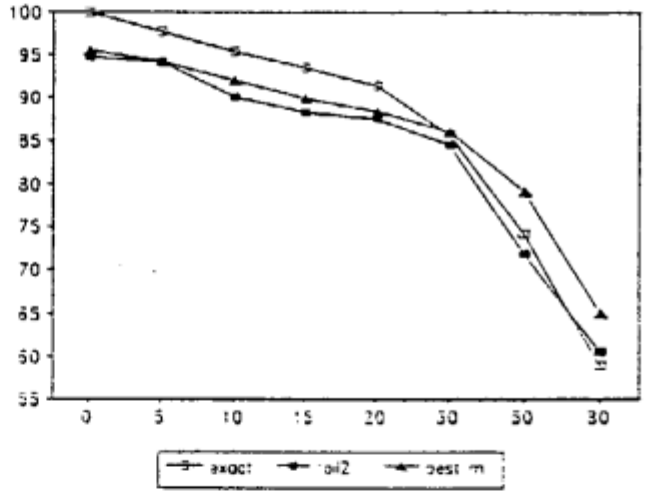
(c) noise in the arguments and the class variable

Table 1: Classification accuracy of mFOIL and FOIL on the testing set.

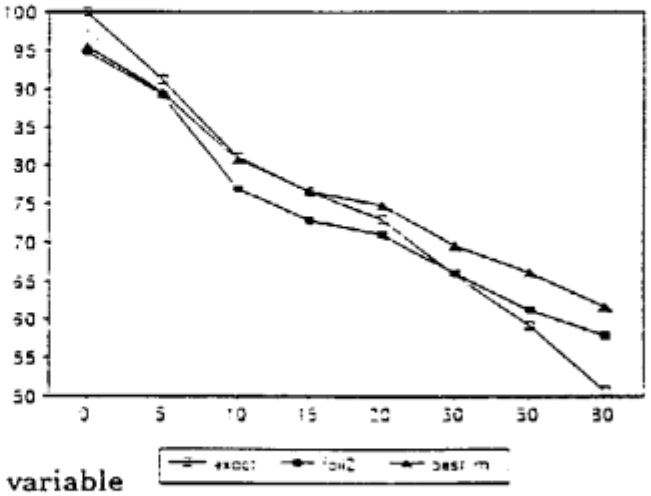
then be correct with probability 1/3 if the class is $\oplus$ and with probability 2/3 if the class is $\ominus$ (the default probabilities of positive and negative examples). The probability of a correct answer would then be $\frac{2}{3} \times \frac{2}{3} + \frac{1}{3} \times \frac{1}{3} = \frac{5}{9}$, or 55 %, which is lower than the 66 % default accuracy. The overfitting observed in our experiments reflects behaviour between these two extremes.



(a) noise in the arguments



(b) noise in the class variable



(c) noise in the arguments and the class variable

Figure 1: Accuracy of FOIL and mFOIL with best m on the KRK endgame

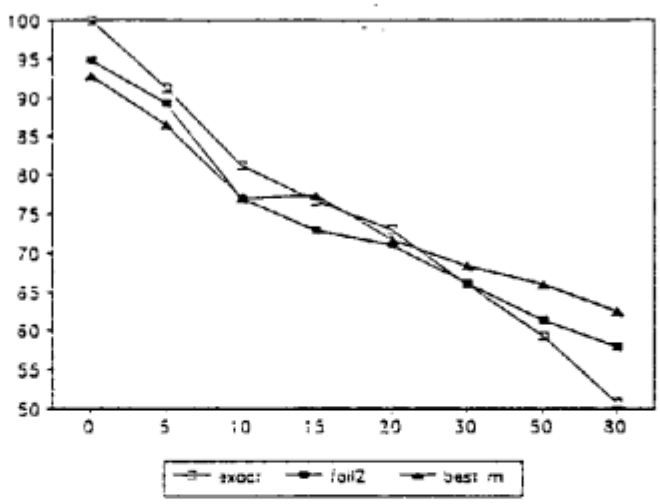


Figure 2: mFOIL with best m and 99 % significance threshold: noise in arguments and class

If the best m is chosen for each percentage of noise, the results achieved by mFOIL are better than the ones achieved by FOIL or mFOIL with the Laplace estimate. Moreover, if mFOIL uses the m-estimate with $m = 0.01$, it still performs better than with the Laplace estimate, as well as better than FOIL. The differences between mFOIL with $m = 0.01$ and FOIL at 10 % and 15 % of noise of all kinds are significant at least at the 90 % level.

Figure 1 gives the accuracies on the test set for mFOIL with the best m and FOIL, for each kind of noise. The curves labeled 'exact' represent the percentage of training examples (possibly noisy) correctly classified by the correct definition of the relation illegal. When noise is introduced in both arguments and class up to 20 %, the accuracy achieved by mFOIL with best m is quite close to the 'exact' curve, which means that mFOIL handles noise well.

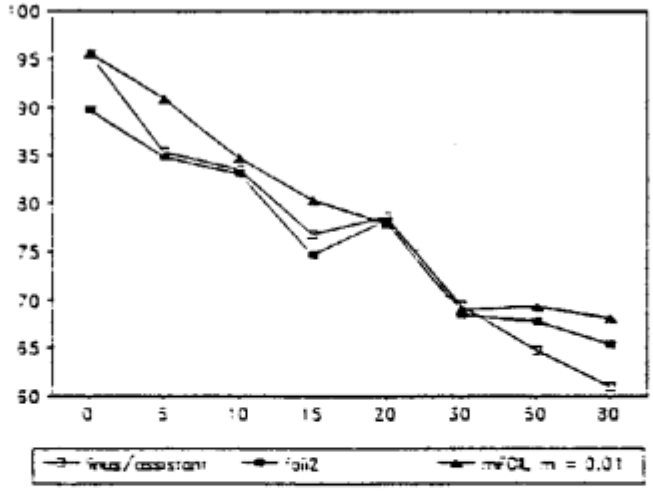


Figure 3: Accuracy on training examples with nonincremental noise in arguments and class

If a 99 % significance threshold is applied the performance of mFOIL with best m decreases. Similar results have been reported for CN2 (Clark and Boswell 91). However, it still performs better than FOIL for noise above 15 % in arguments or both arguments and class. A comparison

of accuracies of mFOIL with best m and FOIL, for noise in arguments and class is given in Figure 2.

Finally, we compare the performance of mFOIL with $m = 0.01$, and a zero significance threshold, FOIL and LINUS using ASSISTANT on training examples with nonincremental noise. The results for noise in arguments and class are given in Figure 3. mFOIL achieved better classification accuracy than both FOIL and LINUS using ASSISTANT.

4 Finite element mesh design

The problem of learning rules for finite element (FE) mesh design was first studied by Dolšak and Muggleton (92), where GOLEM was given examples from three structures, a hydraulic press cylinder, a paper mill and a hook. A comparison of the performance of FOIL, LINUS and GOLEM on the same data is given by Džeroski and Dolšak (91). Dolšak (91) gathered data about three additional, more complex, structures. In this section, we describe the application of mFOIL, FOIL and GOLEM on the mesh design problem, given the data about the hydraulic press, the paper mill and the three additional structures. We first briefly review the learning problem, then describe the experimental setup and finally compare the performance of the three systems.

The learning problem

The finite element method is used extensively by engineers and scientists to analyse stresses in physical structures. Figure 4 shows a typical instance of such a structure, with a corresponding finite element mesh. The structure shown is a cross-section of a cylinder from a hydraulic press used in the leather industry (Dolšak and Muggleton 92). The main problem in designing finite element meshes is to decide on the appropriate resolution for modelling each part of the structure.

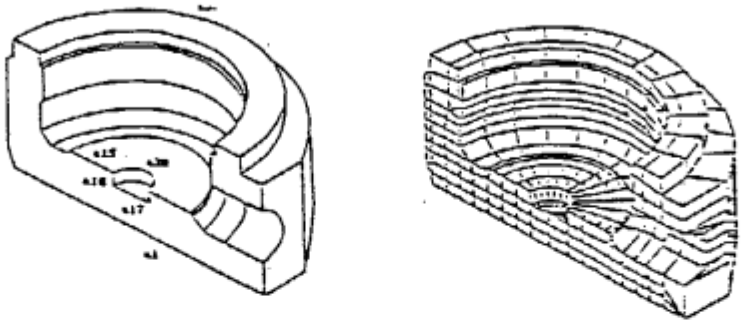


Figure 4: A typical structure and its corresponding FE mesh

The resolution of a FE mesh is determined by the number of elements on each of its edges. The problem of learning rules for determining the resolution of a FE mesh can, thus, be formulated as a problem of learning rules to determine the number of elements on an edge. The training examples have the form $mesh(Edge, Number\_of\_elements)$, where $Edge$ is an edge label (unique for each edge) and $Number\_of\_elements$ is the number of elements on the edge denoted by label $Edge$. In other words, the target relation $mesh$ has arguments of type $edge$ (which consists of all edge labels) and $elements$, where $elements = \{1..12, 17\}$.

	Number of elements													
Structure	1	2	3	4	5	6	7	8	9	10	11	12	17	$\sum$
A	21	9	3	2	3			1			2	13	1	55
B	9	9		1		11	4	4						42
C	6	6	2					14						28
D	14	13		2								28		57
E	23	36	11	5	9	4			6	1		1		96

Table 2: Distribution of number of edges w.r.t. number of elements

In our experiments, five different structures were used for learning, described in more detail in Dolšak (91). The positive training examples were derived from these structures. The negative training examples were generated under a modified closed world assumption. They included most facts of the form $mesh(e, n)$, which were not among the positive examples. However, for the positive example $mesh(c15, 8)$, the facts $mesh(c15, 7), mesh(c15, 9)$ and $mesh(c15, 10)$ are not among the negative examples. A detailed description of the training data is given in Dolšak (91). There were alltogether 278 positive examples and 2840 negative examples. The distributions of the edges according to the number of elements is given in Table 2.

The background knowledge described some of the factors that influence the resolution of a FE mesh, such as the type of edges, boundary conditions and loadings, as well as the shape of the structure (relations of neighborhood and oppositness). According to its importance and geometric shape, an edge can belong to one of the following types: *important_long*, *important*, *important_short*, *not_important*, *circuit*, *half_circuit*, *quarter_circuit*, *short_for_hole*, *long_for_hole*, *circuit_hole*, *half_circuit_hole* and *quarter_circuit_hole*. With respect to the boundary conditions an edge can be *free*, *one_side_fixed*, *two_side_fixed* or *fixed*. Finally, according to the loadings an edge is *not_loaded*, *one_side_loaded*, *two_side_loaded* or *continuously_loaded*.

Training examples	*Background knowledge*		
$\oplus$	*Edge type*	*Boundary condition*	*Loading*
mesh(a1, 17).	*important_long*(a1).	*fixed*(a1).	*not_loaded*(a1).
mesh(a15, 4).	*important_short*(a15).	*fixed*(a15).	*cont_loaded*(a15).
mesh(a16, 1).	*short_for_hole*(a16).	*fixed*(a16).	*cont_loaded*(a16).
mesh(a17, 2).	*long_for_hole*(a17).	*fixed*(a17).	*cont_loaded*(a17).
mesh(a36, 12).	*half_circuit*(a36).	*two_side_fixed*(a36).	*not_loaded*(a36).
mesh(a38, 12).	*half_circuit_hole*(a38).	*two_side_fixed*(a38).	*cont_loaded*(a38).
$\ominus$	*Geometry of the structure*		
mesh(a1, 1).	*neighbour*(a15, a38).	*opposite*(a15, a1).	*same*(a36, a37)
mesh(a15, 12).	*neighbour*(a16, a38).	*opposite*(a17, a1).	*same*(a37, a38)
mesh(a36, 1).	*neighbour*(a15, a16).	*opposite*(a36, a37).	
...	*neighbour*(a16, a17).	*opposite*(a37, a38).	

Table 3: An ILP formulation of the FE mesh design problem

Background knowledge about the shape of a structure includes the symmetric relations *neighbour* and *opposite*, as well as the relation *equal*. The latter states that two edges are not only opposite, but are also of the same length and shape, such as concentric circles. Unlike the unary relations that express the properties of the edges, the relations *neighbour*, *opposite* and *equal* are binary. The arguments of all background predicates are of type *edge*. For each predicate, the first argument is an input argument and the second is an output one. An excerpt from the training examples and background knowledge, describing the labeled edges from Figure 4 is given in Table 3.

Experimental setup

For each of the five structures, a set of clauses was derived from the background knowledge and examples for the remaining four structures and then tested on the structure (leave-one-out). In the classification process, the induced rules have to assign the correct number of element to each of the edges. More precisely, to determine the number of elements on edge E, the goal $mesh(E, N)$ is called, where E is bound and N is not. If N is assigned the correct number, the rules scores one. If it is assigned an incorrect number, or is not assigned a value at all, the rules score zero.

As the recursive clauses that were built in preliminary experiments caused infinite loops, no recursive clauses were allowed in mFOIL and GOLEM, and recursive clauses built by FOIL were discarded afterwards. For the classification process, the rules were ordered according to the Laplace estimate of their expected accuracy. Clauses with accuracy less than 80 % were discarded (in FOIL, this is done automatically).

mFOIL used the background knowledge as described above. In addition, literals of the form $X = v$, where v is a constant, were allowed for variables of type *element*. The Laplace estimate was used as search heuristic. Clauses were constructed until no more significant clauses could be found. The default beam size (five) and significance threshold (99 %) were employed.

As FOIL does not have literals of the type $X = v$, relations of the form $is\_v(X)$ were added to the background knowledge for each constant of type *element*. The default parameters were used in FOIL2.1. A few of the induced rules were recursive (as no partial orders exist in this domain, the recursive literals came into the clauses throgh the determinate literal facility of FOIL2.1). However, they were discarded as they caused infinite loops.

In the experiments with GOLEM, some preprocessing of background relations was necessary. Due to the restrictions of introducing new variables and mode declarations in GOLEM, it was necessary to split each of the relations *neighbour*, *opposite* and *equal* to several subrelations. The transformation is described in detail in Dolšak and Muggleton (92) and Dolšak (91). The noise parameter (number of negative examples that may be covered by a clause) was set to five. The *rlggsample* parameter was set to 50, and the *testsample* was set to 500000. These values were suggested by one of the developers of GOLEM.

FOIL's run time on each training set amounted to five minutes on a SUN SPARC 1. mFOIL, implemented in *prolog*, took about two hours for the same task. GOLEM, with settings as described above, took a little more than one hour.

Results

Several clauses induced by mFOIL are given below. Similar clauses have been induced by FOIL and GOLEM.

```
mesh(A,B) :- B=1, not_important(A).
mesh(A,B) :- quarter_circuit(A), B=9.
mesh(A,B) :- B=2, short(A),
             opposite(A,C), not_important(C).
mesh(A,B) :- two_side_fixed(A), B=6,
             opposite(A,C), cont_loaded(C), half_circuit(C).
```

mFOIL had problems with the relation *neighbour*. This predicate does not discriminate between positive and negative examples. Therefore, the accuracy gain is zero. Consequently, this predicate is never used in the induced clauses. In FOIL, which extends the tuples, the gain is also small, but can be nonzero. Another problem in FOIL is that the large number of background relations increases the number of bits needed to encode a clause. Thus, less clauses are allowed to be built.

	FOIL	*mFOIL*	*GOLEM*
A	17	22	17
B	5	12	9
C	7	9	5
D	0	6	11
E	5	10	10
$\sum$	34	59	52
%	12	21	19

Table 4: Number and percentage of examples correctly classified by FOIL, mFOIL and GOLEM in the FE mesh design domain

The number of test examples correctly classified by FOIL, mFOIL and GOLEM is given in Table 4. mFOIL and GOLEM performed better than FOIL, their performance being comparable. However, the classification accuracies achieved are hardly encouraging. As stated by Dolšak (91), a close inspection of Table 2 reveals that each of the objects has some unique characteristics. These can not be captured when learning from the other structures. In addition, in mFOIL and FOIL, the neighbour relation is not used appropriately in the induced clauses, which means that essential information is not taken into account.

To determine the possible gain of the proper use og the *neighbour* relation, we conducted a simple experiment with mFOIL. The starting clause $mesh(E,N) \leftarrow$ which is refined by specialization, was replaced by the clause $mesh(E,N) \leftarrow neighbour(E,F)$. This enabled mFOIL to use the properties of the neighbouring edges in the induced clauses. While the number of correctly classified edges for objects A to D remained approximately the same, the number of correctly classified edges of object E increased from 10 to 37. This suggests that a significant improvement is possible if information about neighbouring edges is taken into account properly.

To improve the above results, a larger set of structures should be used for learning, which would provide for some redundancy. Information about the neighbors of an edge should also be taken into account in an appropriate manner. To encourage the use of the *neighbor* relation, a lookahead is needed, which would allow for nondeterminate literals with zero or small gain. Such lookahead is already implemented in FOIL for determinate literals with zero or small gain. In a similar way, nondeterminate literals with nonnegative gain might be added to a clause. In the mesh design domain, the cost imposed by this extension should not be prohibitive.

5 Learning qualitative models

Bratko, Muggleton and Varšek (92) presented an application of GOLEM to the problem of learning of qualitative models in the QSIM (Kuipers 86) formalism. A qualitative model of the coupled containers (U-tube) system was generated from four positive examples and six near misses generated by an oracle. The induced model was shown to be dynamically (but not statically) equivalent to the correct model. In this section, we describe the application of mFOIL to the problem of learning a qualitative model of the U-tube system, which consists of two containers connected with a pipe.

Bratko, Muggleton and Varšek (92) suitably formulated the problem of learning qualitative models in the ILP framework. For the U-tube system, the target relation is the relation $legalstate(La, Lb, Fab, Fba)$, where La, Lb denote the levels of water in the containers and Fab, Fba denote the flows from one container to the other. Under suitable assumptions, the problem of learning the legality of states is equivalent to the problem of learning the dynamics

of the system.

The background knowledge consists of the predicates $add(F1, F2, F3)$, $mult(F1, F2, F3)$, $minus(F1, F2)$, $m\_plus(F1, F2)$, $m\_minus(F1, F2)$ and $deriv(F1, F2)$, which correspond to the QSIM constraint primitives. Corresponding values were ommitted to constrain the difficulty of the learning problem. All arguments of the background predicates are of the same type; they are compound terms of the form $Func : QualValue/DirOfChange$.

In our experiments, we used the four positive examples given by Bratko, Muggleton and Varšek (92). However, instead of the six oracle-generated negative examples, we used a set of 543 randomly generated negative examples (Žitnik 91). The results of learning with GOLEM from the same sets of positive and negative examples and several different arrangements of the background knowledge are given in Žitnik (91).

The background knowledge for mFOIL was in the form of a *prolog* program, defining the QSIM theory and consisting of the predicates mentioned above. For comparison with Bratko, Muggleton and Varšek (92), the *mult* relation was excluded from the background knowledge. All of the background predicates, except *deriv* are *symmetric*. For example, $add(X, Y, Z)$ is eauivalent to $add(Y, X, Z)$. The same holds for the predicate *mult*. Similarly, $minus(X, Y)$ is equivalent to $minus(Y, X)$. All arguments of the background knowledge predicates were considered input.

No.	Model
1	minus(Fab,Fba), add(Lb,Fab,La), m_minus(La,Fba), deriv(Fab,Fba)
2	minus(Fab,Fba), add(Lb,Fab,La), m_minus(La,Fba), deriv(Lb,Fab)
3	minus(Fab,Fba), add(Lb,Fab,La), m_minus(La,Fba), deriv(La,Fba)
4	minus(Fab,Fba), add(La,Fba,Lb), deriv(La,Fba), m_minus(Lb,Fab)
5	minus(Fab,Fba), add(La,Fba,Lb), deriv(La,Fba), m_plus(Lb,Fba)
6	**minus(Fab,Fba), add(La,Fba,Lb), deriv(Lb,Fab), deriv(La,Fba)**
7	minus(Fab,Fba), add(La,Fba,Lb), deriv(Fab,Fba), deriv(La,Fba)
8	minus(Fab,Fba), add(La,Fba,Lb), deriv(Fba,Fab), deriv(La,Fba)
9	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(Fba,Fab)
10	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(Fab,Fba)
11	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(Lb,Fab)
12	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(La,Fba)
13	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(Fba,Fab)
14	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(Fab,Fba)
15	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(Lb,Fab)
16	**minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(La,Fba)**
17	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(Fba,Fab)
18	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(Fab,Fba)
19	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(Lb,Fab)
20	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(La,Fba)

Table 5: Qualitative models for the U-tube system generated by mFOIL

The Laplace estimate was used as a search heuristic in mFOIL and the default significance level (99 %) was employed. With a beam size of five (default), two clauses were generated. After we increased the beam size to 20, a single clause was generated:

```
legalstate(La,Lb,Fab,Fba) :-
        minus(Fab,Fba),
        add(Lb,Fab,La),
        m_minus(La,Fba),
        deriv(Fab,Fba).
```

This clause correctly distinguishes between the given positive and negative examples, but is not equivalent to the correct model. However, there are 19 other clauses (models) in the beam, all of them distinguishing correctly between the given positive and negative examples. According to statistical criteria and their complexity (length), these are considered to be equivalent by mFOIL. Table 5 gives all of the models from the beam.

Model 6 in the beam is equivalent to the correct 'classical' model dynamically. It is also statically equivalent, if the corresponding values are not considered. The same holds for model 16 in the beam (Žitnik, personal communication). Out of the possible 194481 possible states, these models cover 130 states (among which 32 are really positive). When all 32 positive examples and the same 543 negative examples were given to mFOIL, it was able to generate, among other clauses in the beam, the two models from Table 5 which are equivalent to the correct model, even with a beam of size 10 and the *mult* relation in the bacground knowledge.

For comparison, from the same examples, and background knowledge as in Bratko, Muggleton and Varšek (92), GOLEM induced the following model (Žitnik 91), where the condition `deriv_simplified(D,E)` actually means `deriv(Lb,Fab)`.

```
legalstate(la:A/B, lb:C/D, fab:E/B, fba:F/D) :-
  deriv_simplified(D,E),
  legalstate(la:A/G, lb:C/H, fab: I/G, fba:J/H).
```

There are several problems with using GOLEM to learn qualitative models. As GOLEM can use only ground facts in the background knowledge, ground facts had to be generated from the non-ground *prolog* program. The number of facts generated was so large that the *add* constraint had to be replaced by three of its sub-constraints, in order to reduce the number of ground facts generated. Another problem with GOLEM is that an induced model can have several interpretations, due to the fact that GOLEM may choose to generalize and introduce new variables (as in the term `fba:J/H` above) whose meaning may be difficult to grasp.

Similar problems appear when using FOIL (Žitnik 91). Some of these problems are absent in LINUS, as it has typed variables, and can use non-ground background knowledge. However, LINUS cannot introduce new variables, which can prevent it from learning an appropriate model when some important variables are missing in the initial description of the problem.

None of the above problems appears in mFOIL, as it can use non-ground background knowledge and the typing of variables prevents unclear generalizations, while still having the possibility to introduce new variables. It should be noted, however, that new variables that may be introduced by the background knowledge predicates are likely to be non-discriminating and thus some kind of lookahead would be needed to treat them properly.

6 Summary and discussion

To summarize, mFOIL uses noise-handling techniques adapted from attribute-value systems in an ILP framework. These include the use of Bayesian probability estimates as search heuristics and significance based stopping criteria. Their use has improved noise-handling as compared to FOIL and LINUS, as demonstrated on the benchmark problem of learning illegal chess endgame positions from noisy examples.

Similarly, mFOIL performs better than FOIL on the problem of learning rules for finite element mesh design. Its performance is comparable (slightly better) to the performance of GOLEM. The fact that the performance of both mFOIL and GOLEM is actually unsatisfactory is primarily due to the small number and diversity of the structures used for training and testing. In addition, the inappropriate treatment of nondiscriminating literals in mFOIL may be improved by using some kind of lookahead, as is experimentally verified.

In the domain of learning qualitative models, the ability of mFOIL to use non-ground background knowledge proved useful. The models generated by mFOIL are represented directly

by QSIM constraints, and not their subcomponents, which makes their interpretaion much easier than the interpretation of models generated by FOIL and GOLEM. Among the models generated by mFOIL, two are equivalent to the classical model both statically and dinamically. As mFOIL can introduce new variables, it may also be suitable for learning models for systems where some important variables are missing in the system description.

It should be noted that the significance criterion used in mFOIL does not take into account the background knowledge given. It is based purely on the number of positive and negative examples covered by a clause and not on its length or the particular literals appearing in its body. Another significance criterion, based on a general encoding scheme, is presented by Muggleton, Srinivasan and Bain (92). As the main difficulties with noise-handling in FOIL stem from the defficiencies of the encoding scheme used, using an improved encoding scheme might significantly improve its performance. Further work will address this problem and investigate the use of a better encoding scheme within FOIL.

References

Bratko, I., Muggleton, S. and Varšek, A. (1992) Learning qualitative models of dynamic systems. To appear in: Muggleton, S.H. (ed) Inductive logic programming. London, UK: Academic Press.

Cestnik, B. (1990) Estimating probabilities: A crucial task in machine learning. Proc. European Conference on Artificial Intelligence, ECAI 90, Stockholm, Sweden.

Cestnik, B. (1991) Estimating probabilities in machine learning. Ph.D. thesis, Faculty of electrical engineering and computer science, University of Ljubljana, Slovenia. (In Slovene).

Cestnik, B., Kononenko, I. and Bratko, I. (1987) ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In: Bratko, I. and Lavrač, N. (eds) Progress in machine learning. Wilmslow: Sigma Press.

Clark, P. and Boswell, R. (1991) Rule induction with CN2: some recent improvements. Proc. Fifth European Working Session of Learning, EWSL 91. Porto, Portugal: Springer-Verlag.

Clark, P. and Niblett, T. (1989) The CN2 induction algorithm. Machine Learning, 3 (4), 261-284.

Dolšak, B. (1991) Constructing finite element meshes using artificial intelligence methods. M.Sc. thesis. Faculty of technical sciences, University of Maribor, Slovenia. (In Slovene)

Dolšak, B. and Muggleton, S.H. (1992) The application of inductive logic programming to finite element mesh design. To appear in: Muggleton, S.H. (ed) Inductive logic programming. London, UK: Academic Press.

Džeroski, S. (1991) Handling noise in inductive logic programming. MSc Thesis, University of Ljubljana, Slovenia.

Džeroski, S., Cestnik, B., and Petrovski, I. (1992) The use of Bayesian probability estimates in rule induction. Submitted for publication.

Džeroski, S. and Dolšak, B. (1991) A comparison of relation learning algorithms on the problem of finite element mesh design. XXVI Yugoslav conference of the society for ETAN. Ohrid, Yugoslavia. (In Slovene)

Džeroski, S. and Lavrač, N. (1991) Learning relations from noisy examples: An empirical comparison of LINUS and FOIL. Proc. Eighth International Workshop on Machine Learning. Evanston, IL: Morgan Kaufmann.

Kalbfleish, J. (1979) Probability and statistical inference (Vol. 2). New York: Springer Verlag.

Kuipers, B. (1986) Qualitative simulation. Artificial Intelligence, 29, 289-338.

Lavrač, N. and Džeroski, S. (1992) Inductive learning of relations from noisy examples. To appear in: Muggleton, S.H. (ed) Inductive logic programming. London, UK: Academic Press.

Lavrač, N., Džeroski, S. and Grobelnik, M. (1991) Learning nonrecursive definitions of relations with LINUS. Proc. Fifth European Working Session on Learning, EWSL 91. Porto, Portugal: Springer-Verlag.

Muggleton, S.H. (1991) Inductive logic programming. New Generation Computing 8 (4), 295-318.

Muggleton, S.H., Bain, M., Hayes-Michie, J. and Michie, D. (1989) An experimental comparison of human and machine learning formalisms. Sixth International Workshop on Machine Learning. Ithaca, NY: Morgan Kaufmann.

Muggleton, S.H. and Feng, C. (1990) Efficient induction of logic programs. First Conference on Algorithmic Learning Theory. Tokyo: Ohmsha.

Muggleton, S.H., Srinivasan, A., and Bain, M. (1992) Compression, significance and accuracy. To appear in: Proc. Ninth International Conference on Machine Learning.

Quinlan, J.R. (1990) Learning logical definitions from relations. Machine Learning 5 (3), 239-266.

Quinlan, J.R. (1991) Determinate literals in inductive logic programming. Proc. Eighth International Workshop on Machine Learning. Evanston, IL: Morgan Kaufmann.

Rouveirol, C. (1990) Saturation: Postponing choices when inverting resolution. Proc. European Conference on Artificial Intelligence, ECAI 90, Stockholm, Sweden.

Žitnik, K. (1991) Machine learning of qualitative models. Technical report IJS-DP-6239, Jozef Stefan Institute, Ljubljana, Slovenia.

Use of heuristics in empirical inductive logic programming *

Nada Lavrač, Bojan Cestnik and Sašo Džeroski
Jožef Stefan Institute
Jamova 39, 61000 Ljubljana, Slovenia
Phone: (+38)(61) 159 199, Fax: (+38)(61) 161 029
nada.lavrac@ijs.ac.mail.yu

Problem Area: concept learning
General Approach: empirical methods
Evaluation Criteria: theoretical analysis

Abstract

Inductive Logic Programming (ILP) has only recently addressed the problem of learning from noisy data. The main goal of the paper is to improve the understanding of noise-handling mechanisms by giving an analysis of proposed heuristics for dealing with noise. It is argued that the proposed accuracy and information gain heuristics can be used as search heuristics and stopping criteria in clause construction, as well as simplification criteria in post-processing of clauses. Furthermore, these heuristics can be improved by applying latest advances in estimating probabilities from the distribution of covered positive and negative examples, in particular by using the m-estimate. The problem of learning illegal positions in a chess endgame is used to illustrate and analyse how different search heuristics split the training set according to the distribution of positive and negative examples, indicating that the m-estimate is most appropriate for predicting the best split.

*This research was funded by the Slovenian Ministry of Science, Research and Technology. Nada Lavrač is grateful to Luc De Raedt and Maurice Bruynooghe for their support of this work and to the Belgian State - Science Policy Office for grant AI/03/05 enabling to write this paper in the stimulating environment of the Katholieke Universiteit Leuven. Authors are grateful to Luc De Raedt, Peter Flach, Igor Mozetič and Gunther Sablon for their comments on the paper.

1 Introduction

Concept learning can be viewed as heuristic search of the space of concept descriptions [16]. The choice of the description language of concepts L_C determines the search space, called the *hypothesis space*. In *Inductive Logic Programming* (*ILP*) [17], the search space is determined by the language of *logic programs* where an induced hypothesis consists of a set of program clauses. Non-interactive *empirical ILP systems*, which induce a single hypothesis from a large collection of examples, use some restricted form of logic programs to gain efficiency. For example, FOIL [23] induces function-free program clauses; GOLEM [19] allows only determinate literals which restricts the way of introducing new variables; QuMAS [18] and LINUS [13] are restricted to deductive hierarchical database clauses.

In empirical ILP, the following is given [9]: languages L_E, L_B and L_C of training examples, background knowledge and induced hypothesis: a set of positive and negative *examples* ($\oplus$ and $\ominus$ ground facts) of an unknown predicate p; *background knowledge* - a knowledge base of predicate definitions q_i (other than p) specifying information about arguments of the examples; a *coverage relation* between L_C and L_E w.r.t. L_B: and a *quality criterion* defined on predicate definitions in L_C. The task is to find a definition for p as a set of clauses, expressible in L_C, satisfying the quality criterion. The induced *target predicate definition* consists of clauses $p(X_1, \ldots, X_n) \leftarrow L_1, \ldots, L_n$. The body $L_1, \ldots, L_n$ is a conjunction of positive literals $q_i(Y_1, \ldots, Y_k)$ and/or negative literals *not* $q_i(Y_1, \ldots, Y_k)$. The quality criterion usually requires that the hypothesis is consistent and complete; when learning from imperfect data this is relaxed, to avoid overly specific hypotheses.

Empirical ILP aims at overcoming the main limitation of classical ID3- and AQ-like algorithms namely the limited expressiveness of attribute-value languages, which do not allow for effective use of background knowledge. As a consequence, ILP can potentially tackle a broader set of problems. However, to solve important real-world problems it must also cope with imperfect data. This problem has only recently been addressed in ILP [23, 12, 2, 10].

Domains involving imperfect data can be *noisy*, *incomplete* and *inexact*: data can also have *missing argument values* in training examples [12]. *Noise* is due to random errors in training examples and/or background knowledge. *Incompleteness* means

training examples which are too sparse to allow reliable correlation detection. *Inexactness* is due to the inappropriate or insufficient background knowledge which does not allow for formulation of an exact definition of the target predicate. Usually, inductive learning systems use the same *noise-handling mechanisms* for dealing with noisy, incomplete and inexact data, typically designed to prevent the construction of hypotheses overfitting the data.

Having selected the description language, the structure of the search space and the search strategy of an ILP system, its success of dealing with imperfect data lies in the choice of *heuristics* used to evaluate the *quality* of a clause. In top-down empirical ILP systems, heuristics used in clause construction consist of *search heuristics* used to select the next literal to be added to the body of a clause, and of *stopping criteria* used to stop the search of literals/clauses. In post-processing, based on *simplification criteria*, induced clauses are refined by eliminating literals, which results in shorter clauses with higher classification accuracy on unseen cases.

Usually, the relative frequency and the Laplace estimate are used as probability estimates in different heuristics. To avoid their limitations, the Bayesian *m-estimate* [4, 7] can be used; it can deal with a small number of examples and can take into account different prior probabilities of classes.

The development of practically applicable ILP algorithms requires a clear understanding of the mechanisms for dealing with imperfect data - their roles, effects and underlying assumptions. To this end, the main goal of the paper is an analysis of heuristics for dealing with noise. Based on the generalization as search paradigm, Section 2 proposes a common framework for describing empirical ILP algorithms performing top-down search of the hypothesis space. Section 3 introduces (weighted) accuracy and information gain as heuristics for evaluating the quality of a clause. Section 4 gives different probability estimates: relative frequency, Laplace estimate and m-estimate. Analysis of applying (weighted) accuracy and information gain as search heuristics in a chess endgame is given in Section 5. Finally, Section 6 gives an overview of some heuristics actually used by individual ILP systems.

2 A generic top-down empirical ILP algorithm

The proposed framework is based on the generalization as search paradigm [16] and the work on generic AQ-like descriptions of concept learning algorithms [11, 9]. Having selected L_E, L_B and L_C, an ILP system can be described in terms of the structure of the search space (specialization/generalization operators); the search strategy; the search heuristic for selecting the next literal to be added to/removed from the body of the current clause; the criterion for stopping the search of literals; the criterion for stopping the construction of the hypothesis; and the method of clause post-processing.

In empirical ILP, the search of the hypothesis space can proceed top-down (FOIL, LINUS, mFOIL [10], FOCL [2]) or bottom-up (GOLEM). In top-down search, based on specialization operators, various search strategies can be applied.

A generic top-down empirical ILP algorithm consists of three main steps: *pre-processing* of the training set, *construction* of a hypothesis and its *post-processing*:

- *pre-process Training_set*
- *initialize Hypothesis* — $Hypo := \emptyset$
- *repeat {covering}*
  - *initialize Clause* — $Clause := T \leftarrow Q$ *and* $Q := true$
  - *repeat {specialization}*
    - *specialize Clause* — $Q := Q, L$
  - *until necessity stopping criterion is satisfied*
  - *add Clause to Hypothesis* — $Hypo := \{Hypo.Clause\}$
  - *remove positive examples covered by Clause from Training_set*
- *until sufficiency stopping criterion is satisfied*
- *post-process Hypothesis according to simplification criterion*

Pre-processing handles missing argument values in training examples and, when no negative examples are given, the generation of negative examples.

The *construction* of a target predicate definition can be seen as consisting of two repeat loops, referred to as *covering* and *specialization*. *Specialization* starts with a clause with an empty body and continues by heuristically searching for a literal L to be added to the *current clause* $T \leftarrow Q$, leading to *new clause* $T \leftarrow Q'$, for $Q' = Q, L$.[1] The two loops are controlled by two *stopping criteria*, used to decide whether to stop adding

[1] It is assumed that all specializations occur by adding a literal to the body of clause $T \leftarrow Q$, including the specialization in which a variable in the head is replaced by a constant. Otherwise, $T' \leftarrow Q'$ should be used.

literals to a clause and whether to stop adding clauses to the hypothesis. In domains with perfect (exact) data, the *sufficiency criterion* requires completeness (coverage of all positive examples) and the *necessity criterion* requires consistency (no covered negative example). In domains with imperfect data, the two criteria are implemented as heuristics for evaluating the clause quality aimed at avoiding overly specific hypotheses.

Post-processing refines the hypothesis by removing literals from a clause and by removing clauses. In domains with imperfect data, heuristic *simplification criteria* are used to reduce complexity and improve the accuracy of the hypothesis when classifying unseen cases. Post-processing of single clauses could be performed immediately after the specialization loop.

3 Accuracy and information gain heuristics

Quinlan [21] introduced an entropy/information-based search heuristic in the construction of decision trees. Various other measures for estimating the "goodness of split" were proposed by other authors [1, 14]. An empirical comparison of search heuristics by Mingers [14] showed that there is little difference between them and that their use reduces the size rather than improves the accuracy of induced hypotheses. However, experiments by Buntine and Niblett [3] showed that this holds in some domains while in the others substantial differences may arrise also in the accuracy. Furthermore, the accuracy depends also on the choice of heuristics applied in post-processing of rules/decision trees/clauses [2, 15, 22, 23], such as heuristics for post-pruning of decision trees.

Heuristics use different probability estimates. It was shown by Cestnik [5] that the method for estimating probabilities used in the heuristics has a greater impact on the accuracy than the actual form of the heuristics (accuracy, entropy, gini-index). Therefore, in order to improve accuracy in ILP, we propose to use simple heuristics defined in this section, and a reliable probability estimation method as proposed in Section 4.

What follows is a proposal of heuristics to be used in ILP as search heuristics and stopping criteria in clause construction, as well as simplification criteria in post-processing. Proposed heuristics are functions that evaluate the *quality* of a clause.

Let n be the number of examples in the initial training set, $n^{\ominus}$ of which are positive.

Furthermore, let $n(Q)$ denote the number of examples in the current training set T_Q. $n^{\oplus}(Q)$ of which are positive and $n^{\ominus}(Q)$ are negative. The current training set T_Q is the set of examples covered by clause $T \leftarrow Q$.

The simplest measure of the quality of clause $T \leftarrow Q$ is its expected *accuracy* defined as probability that an example covered by the clause is positive [2]:

$$A(Q) = p(\oplus|Q) \tag{1}$$

Different probability estimates, outlined in Section 4, can be used to compute $p(\oplus|Q)$, for example, the relative frequency of covered positive examples $\frac{n^{\oplus}(Q)}{n(Q)}$. Expected accuracy can be employed in its variant, named the expected *error estimate*, which is computed as $1 - A(Q)$.

Another measure of the quality of a clause is its expected *informativity*:

$$I(Q) = log_2\ p(\oplus|Q) \tag{2}$$

To better suit the ILP framework, the proposed heuristic differs from the entropy-based informativity [21] which could be used instead. The goal in ILP is namely to build clauses covering as many positive examples as possible (preferably excluding negative examples) and not to build descriptions that would maximally discriminate between classes $\oplus$ and $\ominus$. As for usual entropy, it may be used if we want to generate a description of class $\ominus$ as well.

Given current clause $T \leftarrow Q$ and new clause $T \leftarrow Q'$ with body $Q' = Q, L$, *accuracy gain* $AG(Q, L)$ and *information gain* $IG(Q, L)$ are defined as follows:

$$AG(Q, L) = A(Q') - A(Q) = p(\oplus|Q') - p(\oplus|Q) \tag{3}$$

$$IG(Q, L) = I(Q') - I(Q) = log_2\ p(\oplus|Q') - log_2\ p(\oplus|Q) \tag{4}$$

In heuristics the use of weights is proposed. Without taking into account the number of examples covered by a clause, the heuristics can namely favor very specific clauses with high gain [24]. Thus we introduce weights in formulas (5) and (6). It we weight accuracy and information gain by the relative frequency of positive examples

[2] In ILP terms, $p(\oplus|Q)$ means $p(T|Q)$, i.e., the probability that the head of clause $T \leftarrow Q$ is implied by its body. For correct clauses, $p(T|Q)$ is 1, but this is not necessarily true for clauses induced from noisy examples.

covered at an individual specialization step $\frac{n^{\oplus}(Q')}{n^{\ominus}(Q)}$, we obtain *weighted accuracy gain* $WAG(Q, L)$ and *weighted information gain* $WIG(Q, L)$:

$$WAG(Q, L) = \frac{n^{\oplus}(Q')}{n^{\ominus}(Q)} \times (p(\ominus|Q') - p(\ominus|Q)) \tag{5}$$

$$WIG(Q, L) = \frac{n^{\oplus}(Q')}{n^{\ominus}(Q)} \times (log_2\ p(\ominus|Q') - log_2\ p(\ominus|Q)) \tag{6}$$

The main purpose of introducing weighted gains is to find the balance between the gain and the amount of examples covered by the clause.

In Section 5, the behavior of the above heuristics is analysed. The heuristics actually used in ILP systems, some of which are outlined in Section 6, are their (more complicated) variants.

4 Probability estimates used in the heuristics

Heuristics for evaluating clause quality use probabilities that have to be estimated from the current training set T_Q. Computation of probabilities is based on the distribution (*split*) of positive $n^{\oplus}(Q)$ and negative $n^{\ominus}(Q)$ examples covered by the clause. Usually, relative frequency is used to approximate probability. However, the reliability of this approximation decreases with the decreased size of T_Q; in the extreme case of only one positive example in T_Q the estimate of $p(\oplus|Q)$ is 1. When data is noisy this estimate is too optimistic. To avoid this problem, Laplace's law of succession can be used [20]. It states: if in the sample of N trials there were n successes, the probability of the next trial being successful is $\frac{n+1}{N+2}$, assuming a uniform initial distribution of successes and failures. To avoid this assumption, Cestnik [4] proposed the m-estimate derived by a Bayesian estimation procedure: after n successes in N trials, the probability of success in the next trial is estimated as $\frac{n+p_a m}{N+m}$, where p_a is the prior probability of success and m is a parameter of the method.

Let clause $T \leftarrow Q$ cover $n(Q)$ examples, $n^{\oplus}(Q)$ of which are positive. In ILP, the probability estimates to be used in the heuristics of Section 3 are defined as follows:

Relative frequency:

$$p(\oplus|Q) = \frac{n^{\oplus}(Q)}{n(Q)} \tag{7}$$

Its use is appropriate when the number of covered examples is large, and inappropriate, when it is small (e.g., [4]). Problems arise when $n(Q) = 0$ (division by 0). If $n(Q) > 0$

and $n^{\ominus}(Q) = 0$ then $p(\ominus|Q)$ becomes 0, even if $n(Q)$ is small, meaning that the estimate is not reliable.

Laplace estimate:

$$p(\oplus|Q) = \frac{n^{\oplus}(Q) + 1}{n(Q) + 2} \tag{8}$$

In ILP there are only two classes $\oplus$ and $\ominus$. In general, in a domain with k classes, the corresponding formula is $p(C|Q) = \frac{n^{C}(Q)+1}{n(Q)+k}$. The Laplace estimate is more reliable then the relative frequency when dealing with a small number of examples; however, it relies on the assumption of a uniform prior probability distribution of the two (k) classes, which is rarely met in practice.

m-estimate:

$$p(\oplus|Q) = \frac{n^{\oplus}(Q) + m \times p_a(\oplus)}{n(Q) + m} \tag{9}$$

This estimate takes into account prior probabilities of classes. The prior probability $p_a(\ominus)$ can be estimated by the relative frequency of positive examples in the initial training set: $\frac{n^{\oplus}}{n}$. The parameter m can be set subjectively; it expresses our confidence in the experimental evidence (training examples). The actual value of m should be set according to the amount of noise in the examples (larger m for more noise). As m grows toward infinity, the m-estimate approaches the prior probability of the corresponding class.

By using the m-estimate in ILP, our goal is to provide a firm theorethical background based on Bayesian analysis and to build hypotheses which are more accurate when classifying unknown cases [4, 5]. The m-estimate is a general probability estimate. By appropriately setting its two parameters m and $p_a(\ominus)$, the other two estimates are obtained: relative frequency $p(\ominus|Q) = \frac{n^{\ominus}(Q)}{n(Q)}$ (for $m = 0$), and the Laplace estimate $p(\ominus|Q) = \frac{n^{\ominus}(Q)+1}{n(Q)+2}$ (for $m = 2$, $p_a(\ominus) = \frac{1}{2}$).

5 Experimental analysis of heuristics

The problem of learning illegal chess endgame positions is used to illustrate and analyse the use of proposed (weighted) accuracy and information gain as search heuristics in ILP.

In the chess endgame problem White King and Rook versus Black King the target relation *illegal(A,B,C,D,E,F)* states that the position in which the White King is at

(A,B), the White Rook at *(C,D)* and the Black King at *(E,F)* is not a legal White-to-move position. Arguments A, C and E are of type *file* (with values a to h), and B, D and F are of type *rank* (with values 1 to 8). An example illegal position is $illegal(g,6,c,7,c,8)$. The task is to learn the definition of the predicate *illegal/6* from examples of legal and illegal chess endgame positions and from background knowledge predicates which indicate that rank/file X is adjacent, adjacent or equal, or less than rank/file Y, as well as the equality of ranks/files. Thus, background knowledge consist of $adj\_file(X,Y)$, $aeq\_file(X,Y)$ and $less\_file(X,Y)$ with arguments of type *file*, of $adj\_rank(X,Y)$, $aeq\_rank(X,Y)$ and $less\_rank(X,Y)$ with arguments of type *rank*, and of equality $X=Y$, used for both types of arguments.

Partitions of the training set

Suppose that, from an initial training set of 100 examples [23], an ILP algorithm has generated the following partial clause $T \leftarrow Q$:

$$illegal(A,B,C,D,E,F) \leftarrow adj\_file(A,E) \tag{10}$$

covering the 15 examples given in Table 1, 6 of which are positive and 9 negative. This situation is denoted by a 6-9 split of the current training set T_Q. Note that one of the examples is "noisy": it is incorrectly classified as legal (negative) although it is, in fact, illegal (positive).

Positive examples	*Negative examples*
$illegal(5,2,5,7,4,2)$	$illegal(3,6,6,6,4,7)$ *(noisy)*
$illegal(7,3,7,1,6,4)$	$illegal(5,8,1,6,4,5)$
$illegal(3,4,5,3,4,5)$	$illegal(5,6,7,6,6,3)$
$illegal(6,1,2,7,5,2)$	$illegal(4,3,2,4,5,8)$
$illegal(6,4,2,1,5,4)$	$illegal(3,8,7,3,4,5)$
$illegal(6,5,3,8,7,5)$	$illegal(1,2,7,1,2,8)$
	$illegal(8,2,5,5,7,8)$
	$illegal(3,4,5,6,2,7)$
	$illegal(8,2,2,1,7,7)$

Table 1: Examples covered by the partially built clause.

Adding literal L to the body of clause (10) will lead to a new clause with body $Q' = Q, L$ with new splits of $T_{Q'}$. There are 68 further splits of 6-9: 6-0, 5-0, ..., 1-0, 6-1, ...(7 ways of choosing among 6 positive examples, 10 ways of choosing among 9 negative examples, disregarding splits 0-0 and 6-9; therefore, $7 \times 10 - 2 = 68$). The splits are ordered according to decreasing accuracy (1) computed by the relative frequency estimate (7) (see the decreasing function for AG_{rf} in Figure 1): $6-0_1$, $5-0_2$, $4-0_3$, $3-0_4$, $2-0_5$, $1-0_6$, $6-1_7$, $5-1_8$, $4-1_9$, $6-2_{10}$, $3-1_{11}$, ...; the index denotes the split number. The splits on the x axis of Figures 1 and 2 correspond to this ordering.

Analysis of proposed search heuristics

What literal is to be added next to the body of clause (10) ? For given types and due to the symmetry of predicate arguments there are 48 possible applications of background predicates. Out of 68 splits, only 24 splits can actually occur by adding one of the 48 literals (since different literals may cause the same splits). Below are listed 8 of the 48 possible literals with their corresponding splits: $B = F$ $(3-0_4)$, $A = C$ $(2-0_5)$, $aeq\_file(A,C)$ $(2-0_5)$, $aeq\_rank(B,F)$ $(6-1_7)$, $adj\_rank(B,F)$ $(3-1_{11})$, $adj\_file(C,E)$ $(3-1_{11})$, $not\ less\_rank(B,D)$ $(3-6_{41})$, and $not\ aeq\_rank(B,F)$ $(0-8_{60})$.

Literal	*Split*	AG_{rf}	AG_{Lap}	$AG_{m=2}$	WAG_{rf}
$B = F$	$3-0_4$	1 : 0.600	4 : 0.388	5 : 0.340	11 : 0.120
$A = C$	$2-0_5$	1 : 0.600	6 : 0.338	9 : 0.273	17 : 0.080
$aeq\_rank(B,F)$	$6-1_7$	7 : 0.457	5 : 0.366	4 : 0.348	2 : 0.213
$adj\_rank(B,F)$	$3-1_{11}$	10 : 0.350	10 : 0.255	11 : 0.218	15 : 0.093
Literal	*Split*	IG_{rf}	IG_{Lap}	$IG_{m=2}$	WIG_{rf}
$B = F$	$3-0_4$	1 : 1.322	4 : 0.958	5 : 0.902	15 : 0.264
$A = C$	$2-0_5$	1 : 1.322	6 : 0.865	9 : 0.763	22 : 0.176
$aeq\_rank(B,F)$	$6-1_7$	7 : 1.100	5 : 0.918	4 : 0.918	2 : 0.513
$adj\_rank(B,F)$	$3-1_{11}$	10 : 0.907	10 : 0.695	11 : 0.639	17 : 0.242

Table 2: Clause quality measured by AG, WAG, IG and WIG using different probability estimates. Numbers preceding ":" denote the rank of splits within each individual heuristic.

For four literals, Table 2 gives their corresponding splits and the quality of clause $T \leftarrow Q'$ after adding L. Clause quality is computed with accuracy gains AG and WAG

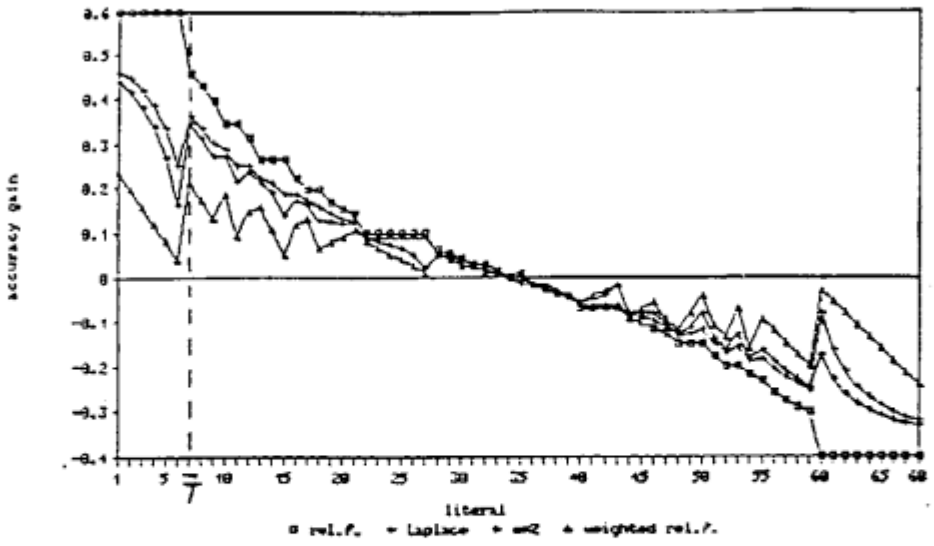


Figure 1: Values of heuristics AG_{rf}, AG_{Lap}, $AG_{m=2}$ and WAG_{rf} on 68 splits, ordered according to decreasing AG_{rf}. Seventh split 6-1. corresponding to literal $aeq\_rank(B, F)$, is evaluated as best by $AG_{m=2}$ and WAG_{rf}.

and information gains IG and WIG using different probability estimates. Index rf denotes relative frequency, Lap Laplace estimate. and $m = 2$ denotes m-estimate with m set to 2 and probability $p_a(\oplus) = \frac{1}{3}$ (relative frequency of positive examples in the initial training set of 100 examples). For example, $4 : 0.348$ (in row 3. column 5) means that split $6 - 1_7$ has fourth best rank when evaluated by $AG_{m=2}$. and only seventh best rank if evaluated by AG_{rf} ($7 : 0.457$ in row 3. column 3).

Figures 1 and 2 show values of heuristics using different probability estimates on 68 splits. Table 2 and Figures 1 and 2 show that heuristics AG and IG using relative frequency evaluate "pure" splits 6-0 to 1-0 as best. This is questionable as small splits. such as 1-0. are unreliable. Knowing the chess endgame domain. the best literal to be added to clause (10) is $aeq\_rank(B, F)$. Its effect is a 6-1 split. reflecting one purposely corrupted example. By AG_{rf} this split is evaluated only as seventh best. On the other hand. heuristics $AG_{m=2}$, WAG_{rf}. $IG_{m=2}$, and WIG_{rf} (as well as heuristics WAG_{Lap}. $WAG_{m=2}$, WIG_{Lap} and $WIG_{m=2}$. which are not in Table 2) would in fact select this literal. For example. $AG_{m=2}$ selects the following best splits: 1: 6-0. 2: 5-0. 3: 4-0. 4: 6-1 (value 0.348 for literal $aeq\_rank(B, F)$ in row 3. column 7). 5: 3-0. etc. Since no literals that would cause 6-0 to 4-0 splits exist. the fourth split 6-1 is selected as best. leading to clause $illegal(A, B, C, D, E, F) \leftarrow adj\_file(A, E), aeq\_rank(B, F)$. Having

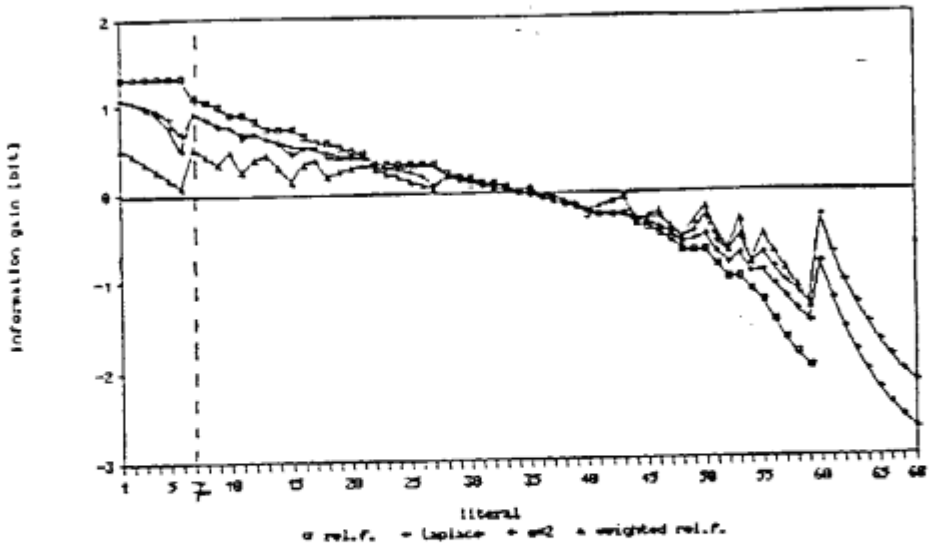


Figure 2: Values of heuristics IG_{rf}, IG_{Lap}, $IG_{m=2}$ and WIG_{rf} on 68 splits, ordered according to decreasing AG_{rf}.

set $m = 2$, the covered negative example is considered noisy.

Figure 1 (and 2) shows that heuristic AG_{rf} (and IG_{rf}) is substantially different from the other three heuristics which are, on the other hand, qualitatively similar. Heuristics AG_{Lap} and $AG_{m=2}$ are the "closest" according to a measure of closeness which was introduced to count the differences between the split ranks produced by the two heuristics. In our domain this is understandable since Laplace assumption of the uniform prior distribution of classes $\ominus$ and $\oplus$ is not too strongly violated, having a 6-9 partition of positive and negative examples.

Furthermore, using the same measure of closeness, it was shown that all WAG (and WIG) heuristics are "closer" to $AG_{m=2}$ (and $IG_{m=2}$) than other heuristics are. Since empirical evidence strongly suggests to use m-estimate in order to achieve better classification accuracy [4, 7, 5, 10], this leads also to the hypothesis that heuristics using weights perform better than unweighted ones, the best being WAG and WIG using m-estimate. As a rule of thumb, if AG is to be used then AG_m is recommended. Else, let WAG be used. To support this claim more firmly, more empirical evidence from the performance on real-life domains is needed.

6 Heuristics of selected ILP systems

Below is an outline of some of the heuristics used by selected ILP systems in construction and post-processing of clauses, relating them to accuracy and information gain, as well as probability estimates.

Heuristics used in clause construction

In clause construction, search heuristics are used to evaluate the quality of a clause in order to select literal L to be added to the body of current clause $T \leftarrow Q$. Stopping criteria are used to decide whether to stop adding literals to a clause and whether to stop adding clauses to the target predicate definition.

FOIL [23] uses a variant of the WIG heuristic (6) using relative frequency (7). Let T_Q contain $n^{\oplus}(Q)$ positive and $n^{\ominus}(Q)$ negative tuples and the choice of literal L give rise to new set $T_{Q'}$. If $n^{\oplus\oplus}(Q)$ positive tuples in T_Q are represented by one or more tuples in $T_{Q'}$, the information gained by selecting literal L amounts to $WIG(Q,L) = n^{\oplus\oplus}(Q) \times (I(Q') - I(Q)) = n^{\oplus\oplus}(Q) \times (log_2 \frac{n^{\oplus}(Q')}{n(Q')} - log_2 \frac{n^{\oplus}(Q)}{n(Q)})$. The difference between this heuristic and (6) is in the weights. In FOIL, $T_{Q'}$ is obtained as a set of extensions of tuples in T_Q that satisfy L. Therefore, if a positive literal with new variables is added to the body of the current clause, the size (arity) of tuples in T_Q increases in $T_{Q'}$. A tuple in T_Q can also give rise to more than one tuple in $T_{Q'}$.

To implement noise-handling, FOIL employs a stopping criterion based on the *encoding length restriction* [23] used to restrict the total length of an induced clause to the number of bits needed to explicitly enumerate the positive examples it covers. The construction of a clause is stopped when it covers only positive examples (is consistent) or when no more bits are available for adding literals to its body. The encoding length restriction allows for building too specific clauses which cover a small number of examples [12] which is inappropriate in noisy domains. The search for clauses stops when no new clause can be constructed under the encoding length restriction. The encoding length restriction does not employ probability estimates.

In mFOIL [10], the search heuristic is the expected accuracy (1) using the Laplace estimate (8) as in CN2 [8] or the m-estimate (9). Probability estimates are computed from the original training set, unlike the local training set of extended tuples in FOIL. This is justified as follows. Suppose a single noisy example, erroneously classified as

positive, is covered by a clause. This example may be extended to, for instance, ten tuples in the local training set; estimating the expected accuracy from ten positive tuples could then lead to a clause covering a single erroneous example.

A sufficiency stopping criterion in mFOIL is the *significance* of a clause. To avoid construction of too specific clauses, mFOIL uses a significance test (similar to the one used in CN2) which ensures that the distribution of examples covered by the clause is significantly different from that which would occur by chance. Significance test does not employ probability estimates. The search for clauses is stopped when all positive examples are covered, or when too few examples are left for a generated clause to be significant, or when no significant clause can be found with expected accuracy greater than the default.

LINUS [13] is an ILP environment which incorporates existing attribute-value learning algorithms ASSISTANT [6], an AQ-like algorithm NEWGEM, and CN2. Therefore, search heuristics and stopping criteria are imported from these systems. For example, in ASSISTANT an information-based search heuristic is applied, and the stopping criteria as used in pre-pruning of decision trees include attribute suitability (estimated by its informativity), class frequency and node weight.

Heuristics used in post-processing

In post-processing, induced clauses are refined according to a simplification criterion.

Clauses in FOIL are post-processed immediately after their specialization is completed: this is done by eliminating *irrelevant* literals. A literal is irrelevant if it can be removed from the clause without decreasing its expected accuracy (1). When the whole set of clauses is constructed, redundant clauses are removed.

Brunk and Pazzani [2] apply *reduced error pruning* to sets of clauses induced by FOCL. The set of training examples is split into a set for training and a set for pruning. Operators *delete last literal* and *drop clause* are independently applied to each clause of the induced definition. The refinement that yields the greatest improvement in accuracy is retained. The procedure is repeated until the application of any of the operators would decrease accuracy.

LINUS using ASSISTANT uses a form of reduced error pruning using the expected accuracy estimate $A(Q) = \frac{n^{\ni}(Q)-0.5}{n(Q)}$ proposed by Quinlan [22]; however, m-estimate should be used instead. An additional mechanism, available within ASSISTANT, is

[2] Brunk, C.A. and Pazzani, M.J. (1991) An investigation of noise-tolerant relation concept learning algorithms. Eighth International Workshop on Machine Learning. Evanston, IL: Morgan Kaufmann.

[3] Buntine, W. and Niblett, T. (1990) A further comparison of splitting rules for decision-tree induction. Technical Report. The Turing Institute, Glasgow, UK.

[4] Cestnik, B. (1990) Estimating probabilities: A crucial task in machine learning. Proc. of ECAI 90, Stockholm, Sweden.

[5] Cestnik, B. (1991) Estimating probabilities in machine learning. PhD Thesis,.Ljubljana University, Slovenia (in Slovene).

[6] Cestnik, B., Kononenko, I. and Bratko, I. (1987) ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In: Bratko, I. and Lavrač, N. (eds.) Progress in machine learning. Wilmslow: Sigma Press.

[7] Cestnik, B. and Bratko, I. (1991) On estimating probabilities in tree pruning. Fifth European Working Session on Learning, EWSL 91. Porto, Portugal: Springer-Verlag.

[8] Clark, P. and Boswell, R. (1991) Rule induction with CN2: Some recent improvements. Fifth European Working Session on Learning, EWSL 91. Porto, Portugal: Springer-Verlag.

[9] De Raedt, L. (1991) Interactive concept-learning. PhD Thesis. Katholieke Universiteit Leuven, Belgium.

[10] Džeroski, S. (1991) Handling noise in inductive logic programming. MSc Thesis. University of Ljubljana, Slovenia.

[11] Gams, M. and Lavrač, N. (1987) Review of five empirical learning algorithms within a proposed schemata. In: Bratko, I. and Lavrač, N. (eds.) Progress in Machine Learning. Chichester, UK: SIGMA Press.

[12] Lavrač, N. and Džeroski, S. (1991) Inductive learning of relations from noisy examples. In: Muggleton, S. (ed.) Inductive Logic Programming. Academic Press (in press).

[13] Lavrač, N., Džeroski, S. and Grobelnik, M. (1991) Learning nonrecursive definitions of relations with LINUS. Fifth European Working Session on Learning, EWSL 91. Porto, Portugal: Springer-Verlag.

[14] Mingers, J. (1989) An empirical comparison of selection measures for decision-tree induction. Machine Learning 3. 319-342. Kluwer Academic Publishers.

[15] Mingers, J. (1989a) An empirical comparison of pruning methods for decision tree induction. Machine Learning 4 (2). Kluwer Academic Publishers.

[16] Mitchell, T. (1982) Generalization as search. Artificial Intelligence 18, 203-226.

[17] Muggleton, S.H. (1991) Inductive logic programming. New Generation Computing 8 (4), 295-318.

[18] Mozetič, I. (1987) Learning of qualitative models. In: Bratko, I. and Lavrač, N. (eds.) Progress in Machine Learning. Chichester, UK: SIGMA Press.

[19] Muggleton, S.H. and Feng, C. (1990) Efficient induction of logic programs. First Conference on Algorithmic Learning Theory. Tokyo: Ohmsha.

[20] Niblett, T. and Bratko, I. (1986) Learning decision rules in noisy domains. Expert Systems 86. Cambridge University Press.

[21] Quinlan, J.R. (1986) Induction of decision trees. Machine Learning 1 (1), 81-106.

[22] Quinlan, J.R. (1987) Simplifying decision trees. International Journal of Man-Machine Studies. 27. 221-234.

[23] Quinlan, J.R. (1990) Learning logical definitions from relations. Machine Learning 5 (3), 239-266.

[24] Smyth, P., Goodman, R.M. and Higgins, C. (1990) A hybrid rule-based/Bayesian classifier. Proc. of ECAI 90. Stockholm, Sweden.

Constraint-directed Generalization for Learning Spatial Relations

Fumio Mizoguchi and Hayato Ohwada
Intelligent System Laboratory
Science University of Tokyo
Noda, Chiba (278), Japan

Abstract

The goal of this paper is twofold. First, the paper incorporates a new computational mechanism, *constraint-directed generalization*, into constraint logic programming. This mechanism is to find a set of constraints that subsume every constraint set, and therefore it can be viewed as a constraint logic programming version of least general generalization in inductive logic programming. Second, we propose a method for learning spatial relations using constraint-directed generalization. In this method, a set of quantitative data as examples are subsumed to a region which is represented as a set of linear constraints. This region is then translated to abstract terms, that corresponds to qualitative descriptions. The method supports acquision of spatial constraints about layout problems and constraint generation for transforming quantitative information to qualitative descriptions.

1 Introduction

Reasoning about spatial relations on real-world objects is important for Artificial Intelligence, Computational Geometry, Robotics, Graphics, CAD and so on. For example, qualitative reasoning about space and motion[3], and maintaining consistency for interactive graphics[8][11] involve computation of geometrical relations on parameterized objects. In this application, constraint is a useful notion rather than procedure and control structure. Programming is achieved by giving a set of constraints, and program execution is done by using constraint satisfaction, which assigns consistent values to variables. This programming style called constraint programming has been recently emerging, and various constraint languages have been developed[15][5][8].

Constraint programming supports declarative style of programming. However, it is difficult to give an appropriate set of constraints for solving problems, because spatial constraints are represented as equations and inequalities. For example, extracting constraints about layout problems and constructing the model of a robot manipulation environment are time-consuming tasks. The ultimate goal of this research is to provide an automatic constraint acquisition rather than describing constraints directly.

To solve the problem, we propose a computational mechanism for learning spatial relations from examples. This mechanism can be characterized as *constraint-directed generalization*, which finds a set of constraints subsuming every constraint set. A desired set of linear constraints are automatically obtained by using constraint-directed generalization. This set can then be transformed to qualitative descriptions through abstraction. Finally, spatial relations are represeted as logic programs.

From a standpoint of constraint programming, constraint-directed generalization is regarded as introducing a generalization operator into a constraint language. There are typical logical operations, *and* and *or*. While *and* operation constructs a region which is represented as a set of constraints, *or* does not. The generalization operator constructs the minimum region subsuming *or*-constructed region.

Acquiring spatial relations can be viewed as learning qualitative rules from a given set of quantitative data. Recent work on machine learning has focused on application to qualitative reasoning about physical systems. For example, Bratko *et. al.* demonstrated a learning method of qualitative models from the behaviors of a physical system in inductive logic programming framework[10][1]. The framework of constraint-directed generalization can be integrated with inductive logic programming, because constrained clauses are induced by incorporating the structure of constraints into inductive logic programming. The similar approach is found in [12].

The paper is organized as follows: First, we show a basic idea for learning spatial relations using constraint-directed generalization. We then design a constraint logic programming language for realizing constraint-directed generalization. Section 4 presents an integrated framework between constraint logic programming and inductive logic programming. Section 3 and 4 provides a basis for the subsequent sections. Section 5 formulates constraint-directed generalization. In this section, constraint logic programs are generated from examples and background knowledge which reflects feature descriptions of the problem. Section 6 translates constraint sets to qualitative descriptions through abstraction. Section 7 provides some extentions. Section 8 concludes the present study.

2 Basic Idea

Before describing a learning method, we illustrate a simple example for explanatory purpose. Figure 1 shows an object and an obstacle that are connected at a point. This figure represents translational and rotational freedom of the object.

Our goal is to generate a general rule for freedom from positive instances. Finding the movable region of freedom is not trivial; it must be derived from geometrical relations between the object and the obstacle. In the case of rotational freedom especially, it seems to be difficult to find such region depicted in Figure 1. Instead, it is easy to detect whether specific objects are movable or not, and therefore generating rules from such examples is meaningful.

We focus on the following properties in learning spatial relations:

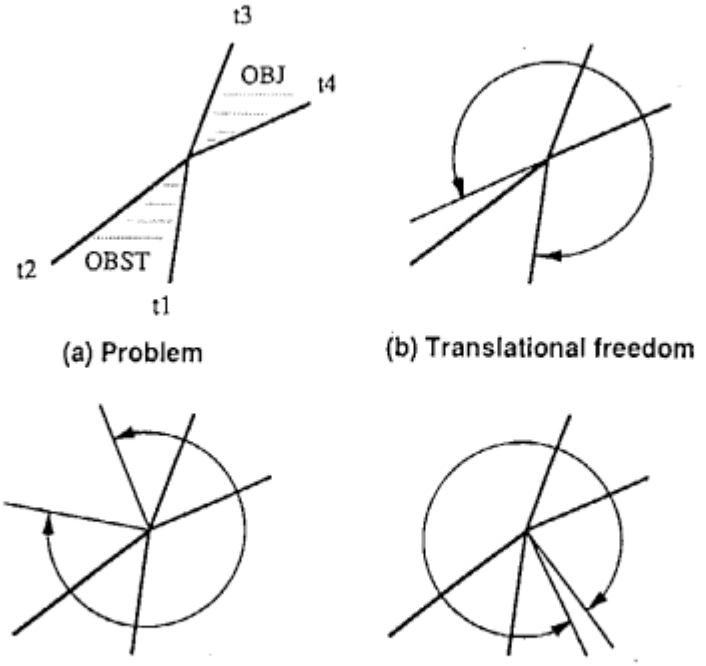


Figure 1: Spatial Relations [14]

1. Problem description

   It is important for problem descriptions to decompose a given problem into a set of meaningful sub-spaces. In Figure 1, the auxiliary line of a boundary of the object is essential for translational freedom. To solve this problem, we shall produce the extended problem description from a given problem. This mechanism is supported by an axiomatic system in constraint logic programming.

2. Operations on objects

   This property concerns with problem descriptions. Interactions among objects must be considered in producing the extended problem description. Since the object and the obstacle are connected at the edge, problem descriptions are extended how such objects interact.

3. Generating examples

   An efficient learning depends on how to generate and select a desired set of examples in the extended problem description. The reasoner finds a representative in this description, and poses a query whether it is positive or negative. In this process, coarse decomposition improves the efficiency for learning; a set of examples are sequentially produced.

4. Generalization to feasible regions

   This property finds the minimum region which subsumes every positive instances and is consistent with every negative instances. Since positive instances are given as a set of constraints,

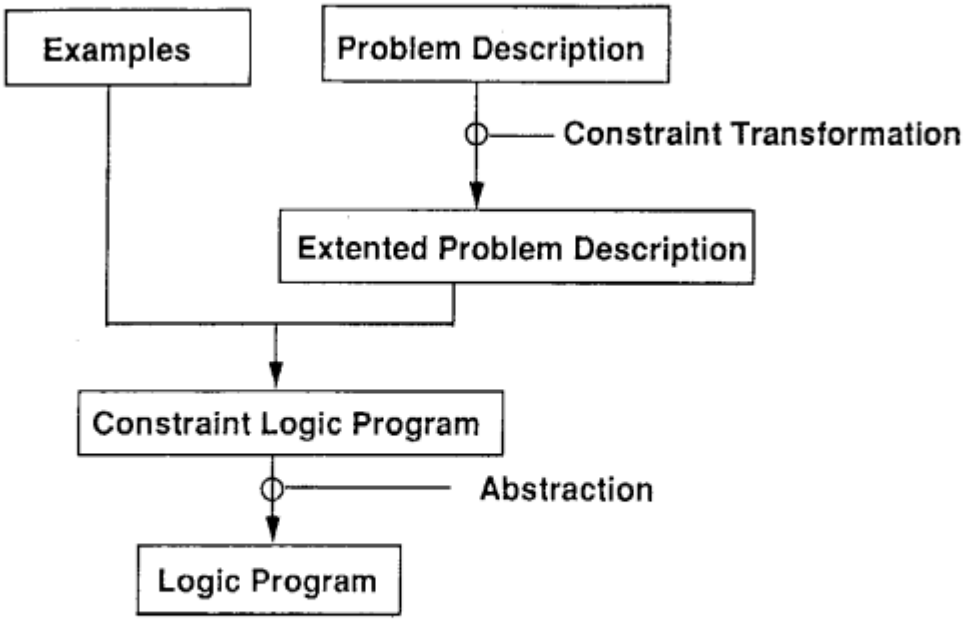


Figure 2: Constraint-directed Generalization

the generalization works as information compression.

5. Transforming constraint sets to symbolic descriptions
   A generalized region does not corresponds to problem descriptions. The transformation from constraint sets to symbolic descriptions is achieved by abstraction. This process produces a constraint logic program.

6. Verification
   The produced program is verified by posing queries where examples are selected from the extended problem description.

The learning method we propose is shown in Figure 2. Parts in arcs are procedures, and rectangle areas are input-outputs for those procedures.

3 Language Design

In this section, we design a constraint logic programming language for constraint-directed generalization.

3.1 Syntax and semantic domain

Constraint logic programming is formulated as many-sorted first-order logic[4]. A symbol such as function, predicate and variable is a sequence of a set of sorts. Let a set of function symbols, predicate symbols and variables be Σ, Π and V respectively. $\tau(\Sigma \cup V)$ denotes a set of terms. Π is divided into Π_P and Π_C. The former is a set of user-defined predicate symbols, while the latter a collection of built-in predicate symbols such as equality.

For $p \in \Pi_P, c \in \Pi_C, t_i \in \tau(\Sigma \cup V)$, We say $p(t_1, \ldots, t_n)$ is an atom, and $c(t_1, \ldots, t_m)$ is a constraint.

A constrained clause is of the form:

$$H \diamond C$$
$$H \diamond C \leftarrow B_1, \ldots, B_n$$

where H, B_i are atoms, and C is a constraint set.

In constraint logic programming, the structure gives an interpretation of Π and Σ. The structure $\Re$ specifies assignments of Π and Σ. Let θ be an assignment to variables. The solution $soln(c)$ of the constraint c is represented as follows:

$$soln(c) = \{\theta | \Re \models c\theta\}$$

In case of a constraint set C, the solution is described below.

$$soln(C) = \{\theta | \forall c \in C \ \Re \models c\theta\}$$

C is satisfiable if the solution of C is not empty.

$$\exists \theta \ \Re \models C\theta$$

Let $A \diamond C$ be a constrained atom. If C is satisfiable,

$$\{A\theta | \forall c \in C \ \Re \models c\theta\}$$

is called interpretation.

Let I be an interpretation. The clause

$$H \diamond C \leftarrow B_1, \ldots, B_n \tag{1}$$

is true, iff the following conditions are satisfied:

$$\exists \theta \ \Re \models C\theta$$
$$\{B_1\theta, \ldots, B_n\theta\} \subseteq I \rightarrow H\theta \in I$$

Lemma 1

$$H \diamond C_0 \leftarrow B_1 \diamond C_1, \ldots, B_n \diamond C_n \tag{2}$$

implies

$$H \diamond (C_0 \cup \ldots \cup C_n) \leftarrow B_1, \ldots, B_n \tag{3}$$

Proof: Suppose that (2) holds:

$$\exists\theta\ \forall C_i\ \Re \models C_i\theta$$
$$\{B_1 \diamond C_1\theta, \ldots, B_n \diamond C_n\theta\} \subseteq I$$

Thus,

$$H \diamond C_0\theta \in I$$

holds. Then the formula

$$\exists\sigma\ \Re \models (C_0 \cup \ldots \cup C_n)\sigma$$
$$\{B_1\sigma, \ldots, B_n\sigma\} \subseteq I$$

is satisfied. Therefore, (3) holds. □

The above lemma indicates that constraints can be eliminated from constrained clauses in the body.

3.2 Constraint system

We define the domains of sorts as follows:

- Herbrand universe
- Real
- Finite set of constraints

The first two is the same as the constraint logic programming language, CLP(R). The last indicates a finite set of constraints, that are constructed by Π_C and $\tau(\Sigma \cup V)$. In this formulation, terms in constraints may be recursive structure.

The constraint set Π_C consists of "$\{=, >, \geq\}$". A collection of function symbols includes the operator "$*$" and "$\oplus$". "$*$" finds the union of two constraint sets, and is regarded as the ordinal AND operation in constraint logic programming. "$\oplus$" finds the minimum set of constraints of two constraint sets, and thus we call the operator *generalization operator*. The minimum set of constraints is defined below.

Definition 1 *Let $\{C_1, \ldots, C_n\}$ be a collection of constraint sets. We say the constraint set C the least general constraint set (in short, LGCS) of $\{C_1, \ldots, C_n\}$, if the following conditions are satisfied:*

1. $C \geq C_i$
2. *For all D such that $D \geq C_i$, $D \geq C$*

where

$$C \geq D$$

indicates that

$$soln(C) \supseteq soln(D)$$

The two functions are closed, and constructs the following algebraic system:

Proposition 1 (Constraint system)

commutative law	$C_1 * C_2 = C_2 * C_1$
	$C_1 \oplus C_2 = C_2 \oplus C_1$
idempotent law	$C * C = C$
	$C \oplus C = C$
associative law	$(C_1 * C_2) * C_3 = C_1 * (C_2 * C_3)$
	$(C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3)$
	$C \oplus \{\} = C$
weak distribution law	$(C_1 \oplus C_2) * C_3 \geq (C_1 * C_3) \oplus (C_2 * C_3)$
	$(C_1 \oplus C_3) * (C_2 \oplus C_3) \geq (C_1 * C_2) \oplus C_3$

Proof: The formula about "$*$" is obvious. The commutative and idempotent law about "$\oplus$" is also clear. We prove the associative law about "$\oplus$" below.
Let C be the minimum set of constraints of $\{C_1, C_2, C_3\}$.

$$\begin{aligned} GC &\geq C_1 \oplus C_2 \\ GC &\geq C_3 \end{aligned}$$

The minimum set of $C_1 \oplus C_2$ and C_3 is more specific than any general constraint set of the three.

$$GC \geq (C_1 \oplus C_2) \oplus C_3 \tag{4}$$

However, $(C_1 \oplus C_2) \oplus C_3$ is general than C_1, C_2, C_3.

$$\begin{aligned} (C_1 \oplus C_2) \oplus C_3 &\geq C_1 \\ (C_1 \oplus C_2) \oplus C_3 &\geq C_2 \\ (C_1 \oplus C_2) \oplus C_3 &\geq C_3 \end{aligned}$$

Thus,

$$(C_1 \oplus C_2) \oplus C_3 \geq GC \tag{5}$$

By (4),(5),

$$GC = (C_1 \oplus C_2) \oplus C_3$$

Similary,

$$GC = C_1 \oplus (C_2 \oplus C_3)$$

We then prove the weak distribution law.

$$C_1 \oplus C_2 \geq C_1 \geq C_1 * C_3$$

Thus,

$$(C_1 \oplus C_2) * C_3 \geq C_1 * C_3 \tag{6}$$

Similarly,

$$(C_1 \oplus C_2) * C_3 \geq C_2 * C_3 \tag{7}$$

By (6),(7),

$$(C_1 \oplus C_2) * C_3 \geq (C_1 * C_3) \oplus (C_2 * C_3) \tag{8}$$

By (8),

$$\begin{aligned}(C_1 \oplus C_3) * (C_2 \oplus C_3) &\geq (C_1 * (C_2 \oplus C_3)) \oplus (C_3 * (C_2 \oplus C_3)) \\ &\geq (C_1 * C_2) \oplus (C_1 * C_3) \oplus (C_2 * C_3) \oplus C_3 \\ &= (C_1 * C_2) \oplus C_3\end{aligned}$$

□

Note that the distribution law does not hold. This is due to the fact that the generalization operator convers a larger region of original regions that are constructed by AND operator. Thus, in order to generalize a set of constraints, we first apply AND operator and the generalization operator in turn.

4 Inductive Constraint Logic Programming

In this section, we propose inductive constraint logic programming framework, which generalizes constrained clauses based on the structure presented in the previous section. Inductive logic programming proposed by Muggleton[10] is based on inverse resulution rules. These rules consist of V-Operator and W-Operator. Using the operators, Plotkin's RLGG(relative least general generalization)[13] can be realized.

In general, the statement "the formula A is general than the formula B with respect to C" is described as follows:

$$C \models A \rightarrow B \text{ and } C \not\models B \rightarrow C$$

In constraint logic programming, the above formula is extended as follows:

$$\Re \wedge C \models A \rightarrow B \text{ and } \Re \wedge C \not\models B \rightarrow A$$

where $\Re$ denotes the structure. Thus, RLGG is formulated below.

1. $\Re \wedge P \models RLGG \rightarrow (A \wedge B)$
2. $\Re \wedge P \not\models A \wedge B$
3. For any consistent C such that $\Re \wedge P \models C \rightarrow (A \wedge B)$
   $\Re \wedge P \models C \rightarrow RLGG$

We formulate inverse resolution and RLGG in constraint logic programming below.

4.1 Inverse resolution wih constraints

We focus on absorption rule, which is one of the functions of V-Operator. It is sufficient for spatial relations to deal with this rule, though other rules are formulated in a similar way. Before doing this, we define *partially derived clause.*

Definition 2 (Partially derived clause) *Suppose that a constrained clause is given as follows:*

$$H_1 \diamond C_1 \leftarrow A_1, \ldots, A_n \tag{9}$$

We say the above clause partially derived clause, if there is no clause such that

$$B_i \diamond C \leftarrow D_1, \ldots, D_m$$

where B_i is variant for A_i.

To obtain a partially derived clause from a constrained clause CL, we substitute atoms in the body of CL by resolution. This process corresponds to partial computation.

Proposition 2 *A constraint set C_1 of a constrained clause CL is more general than that C_2 of the partially derived clause:*

$$C_1 \geq C_2$$

Proof: A constraint is added to the original constraint set C_1 through resolution. Thus, C_1 is more general than C_2. □

The following proposition is a constraint logic programming version of absorption.

Proposition 3 (Absorption) *Let CL_1*

$$H_1 \diamond C_1 \leftarrow A_1, \ldots, A_n \tag{10}$$

be a partially derived clause. Suppose that the clause CL is derived by resolving CL_1 and CL_2 which contains an negative literal, and CL is of the form:

$$H \diamond C \leftarrow B_1, \ldots, B_m \tag{11}$$

The most specific clause $CL_2\downarrow$ constructed by V-operator is represented as follows:

$$H \diamond (C * C_1 * \theta_1) \leftarrow B_1, \ldots, B_m, H_1 \tag{12}$$

where θ_1 is a constraint set used for unification between $H_1 \diamond C_1$ and the negative clause in $CL_2\downarrow$.

Proof: Suppose that (10) and the negative clause are of the form:

$$\begin{aligned} CL_1 &= \{H_1 \diamond C_1\} \cup A \\ CL_2 &= \{H_2 \diamond C_2\} \cup D \end{aligned}$$

where the following conditions are satisfied:

$$\neg H_1 \diamond (C_1 * \theta_1) = \underset{Dk \in D}{D_k} \diamond (C_2 * \theta_2)$$

We put as follows:

$$\begin{aligned} CL_1' &= A \diamond (C_1 * \theta_1) \\ CL_2' &= \{H_2 \diamond C_2\} \cup (D - \{D_k\}) \diamond \theta_2 \\ E &= CL_1' - CL_2' \end{aligned}$$

Then the following equations are satisfied:

$$\begin{aligned} CL_2' &= CL - E \\ &= \{H_2 \diamond C_2\} \cup (D - \{D_k\}) \diamond \theta_2 \end{aligned}$$

where the inverse constraint set of θ_2 is denoted by θ_2^{-1}. Thus,

$$\theta_2 * \theta_2^{-1} = \{\}$$

This yields

$$\{H_2 \diamond C_2\} \cup D - \{D_k\} = (CL - E) \diamond \theta_2^{-1}$$

Therefore,

$$\begin{aligned} CL_2 &= (CL - E) \diamond \theta_2^{-1} \cup \{D_k\} \\ &= (CL - E) \diamond \theta_2^{-1} \cup \{\neg H_1\} \diamond (C_1 * \theta_1 * \theta_2^{-1}) \\ &= ((CL - E) \cup \{\neg H_1\} \diamond (C_1 * \theta_1)) \diamond \theta_2^{-1} \end{aligned}$$

The most specific clause$CL_2 \downarrow$ of CL_2 is obtained by the equation

$$E = \theta_2^{-1} = \phi$$

Hence,

$$CL_2 \downarrow = CL \cup \{\neg H_1\} \diamond (C_1 * \theta_1)$$

By Lemma 1,

$$\begin{aligned} CL_2 \downarrow &= H \diamond C \leftarrow B_1, \ldots, B_m, H_1 \diamond (C_1 * \theta_2) \\ &= H \diamond (C * C_1 * \theta_1) \leftarrow B_1, \ldots, B_m, H_1 \end{aligned}$$

□

Note that the most specific constraint set is used for finding $CL_2 \downarrow$.

4.2 Least general generalization of constrained clauses

As Muggleton presented, RLGG is derived by finding Lgg of a set of clauses that V-Operator constructs. We describe Lgg for constrained clauses below.

Let a constrained clause be

$$H \diamond C \leftarrow A_1, \ldots, A_n \tag{13}$$

Suppose that the atoms H and A_i are translated as H' and A_i' that have distinct variables. The formula (13) is translated as follows:

$$H' \diamond C' \leftarrow A_1', \ldots, A_n'$$

where C' includes the equality between fresh variables and terms. The above clause is called the normal form of (13).

Let two normal forms be

$$H_1 \diamond C_1 \leftarrow A_1, \ldots, A_n$$
$$H_2 \diamond C_2 \leftarrow B_1, \ldots, B_n$$

where H_1 and H_2 have the same predicate symbol, and these have distinct variables. In this case, the normal forms are unifiable. Suppose that θ is a mgu of H_1 and H_2. Lgg of the normal forms is obtained as follows:

$$H_1 \diamond (C_1\theta_1\sigma \oplus C_2\theta_2\sigma) \leftarrow D_1\sigma, \ldots, D_k\sigma, \ldots, D_q\sigma$$

where

$$\sigma = \cup_{i,j} \{A_i = B_j\}$$
$$D_k\sigma = A_i = B_j$$

So far, we do not describe a generalization procedure for constraint sets. This is done in the next section.

5 Constraint-directed Generalization

Finding the minimum set of constraints depends on the set of sorts. In Herbrand universe (ex. Prolog), two terms are compared and are replaced by a variable if the function symbols are different. In linear constraint domain, the minimum set is a convex hull that covers two constraint sets.

In general, spatial relations are specified with Euclid space R^n. Suppose that we consider a bounded space $S \in R^n$. A set of linear constraints construct a convex hull, which is the minimum polyhedral set covering the point sequence $(x_1, x_2, \ldots, x_m)$ in R^n. Thus, the minimum constraint set covers any convex hulls corresponding to given constraint sets. Due to the result of computational geometry, we can simply design a constaint solver for constraint-directed generalization.

Let a point in S be x_0, and a non-zero vector be $\vec{x}_i \in S(i = 1, m)$. A linear constraint is represented as follows:

$$\begin{aligned} x &= x_0 + \lambda_1 \vec{x}_1 + \cdots + \lambda_m \vec{x}_m \\ \lambda_i &\geq 0 \\ \sum \lambda_i &\leq 1 \end{aligned} \tag{14}$$

where x_0 is called the *starting point*.

Linear constraints are described within a constraint logic programs. As for Figure 1, a program may be defined as follows:

$$\begin{aligned} border(T1) &\diamond \{T1 = \{X = -50 * L, X = -100 * L, 0 \leq L \leq 1\}\}. \\ border(T2) &\diamond \{T2 = \{X = -100 * L, X = -50 * L, 0 \leq L \leq 1\}\}. \\ border(T3) &\diamond \{T3 = \{X = 25 * L, X = 100 * L, 0 \leq L \leq 1\}\}. \\ border(T4) &\diamond \{T4 = \{X1 = 100 * L, X2 = 100 * L, 0 \leq L \leq 1\}\}. \end{aligned}$$

However, it is impossible to find freedom of the object using the above definition. The definition does not include potential constraints that are meaningful for problem descriptions. For example, auxiliary lines such as symmetric and orthogonal one with respect to a boundary play a role for freedom. To solve the problem, we propose a method of constraint transformation in the next section.

5.1 Constraint transformation

Let the transformation system be T. T consists of the following rules:

- symmetric operator "$C \bowtie x_k$"
  A vector x_k is selected from a set of vectors. The sign of an element of x_k is reversed, and x'_k denotes such vector. The original constraint is transformed as follows:
  $$x = x_0 + \lambda_1 \vec{x}_1 + \cdots + \lambda_k \vec{x'}_k + \cdots + \lambda_m \vec{x}_m$$
- orthogonal operator "$C \perp x_k(s, t)$"
  A vector is selected from a set of vectors x_k. Two elements s, t of x_k are substituted for $-t, s$, and x'_k denotes such vector. The original constraint is transformed as follows:
  $$x = x_0 + \lambda_1 \vec{x}_1 + \cdots + \lambda_k \vec{x'}_k + \cdots + \lambda_m \vec{x}_m$$
- translational operator "$C \parallel y_0$"
  Another constraint
  $$y = y_0 + \lambda_1 \vec{y}_1 + \cdots + \lambda_m \vec{y}_m$$
  is selected. The transformed constraint is as follows:
  $$x = y_0 + \lambda_1 \vec{x}_1 + \cdots + \lambda_m \vec{x}_m$$

As for Figure 1, the symmetric transformation constructs boundary lines that are symmetric to the original line with respect to x- and y-axis. The orthogonal transformation is used for finding rotational freedom. The translational transformation plays a role of decomposing the original problem description through interactions of objects, though this transformation is not used in this case.

A transformation step is to construct a constraint set C_{i+1} from C_i through the transformation system T. This process is described as follows:

$$C_i \xrightarrow{T} C_{i+1}$$

Transformation steps construct from the original problem description C_0 to $C_i (i = 1, \infty)$. The extended problem description is represented as $\cup\, C_i$.

The constraint $inv(t3)$ in Figure 1 is obtained by transformation steps twice. This constraint is described below.

$$x_1 = -25\lambda, x_2 = -100\lambda$$

5.2 Generalizing constraint logic programs

A transformed constraint is also linear. Thus, the extended problem description PD is described as the form (14) within constraint logic programs. Positive E^+ and negative instances E^- are also described within constraint logic programs. Background knowledge K is specified as constraint logic programs. Constraint-directed generalization is specified as the tuple (PD, E^+, E^-, K), and is to find a set of clauses H which satisfies the following conditions:

1. $\Re \wedge PD \wedge K \models H \rightarrow E^+$
2. $\Re \wedge PD \wedge K \not\models H \rightarrow E^-$

As for Figure 1, the above formulation is described as follows:

PD :

$$\begin{aligned}
border(T1) &\diamond \{T1 = \{X = -50 * L, Y = -100 * L, 0 \leq L \leq 1\}\}. \\
&\vdots \\
border(T1 \bowtie X) &\diamond \{T1 = \{X = -50 * L, Y = 100 * L, 0 \leq L \leq 1\}\}. \\
&\vdots \\
border(T1 \bowtie [X, Y]) &\diamond \{T1 = \{X = 50 * L, Y = 100 * L, 0 \leq L \leq 1\}\}. \\
&\vdots
\end{aligned}$$

E^+ :

$$movable(C1) \diamond \{C1 = \{X = -4 * L, Y = -1 * L, 0 \leq L \leq 1\}\}. \quad (15)$$

$$movable(C2) \diamond \{C2 = \{X = -1 * L, Y = 3 * L, 0 \leq L \leq 1\}\}. \quad (16)$$

$$movable(C3) \circ \{C3 = \{X = 3 * L, Y = -1 * L, 0 \leq L \leq 1\}\}. \quad (17)$$

$$movable(C4) \circ \{C4 = \{X = L, Y = -4 * L, 0 \leq L \leq 1\}\}. \quad (18)$$

E^- :

$$\leftarrow movable(OBST) \circ \{OBST = \{X = -50 * L1 - 100 * L2 - 100 * L3,$$
$$Y = -100 * L1 - 100 * L2 - 50 * L3,$$
$$0 \leq L1, 0 \leq L2, 0 \leq L3, L1 + L2 + L3 \leq 1\}\}.$$

K :

$$clockwise(T_1, T_2) \leftarrow$$
$$border(T_1), border(T_2),$$
$$start(T_1) = start(T_2),$$
$$angle(T_1) > angle(T_2).$$

Notice that there is the relation $clockwise(T_1, T_2)$ in the background knowledge. This relation specifies that the boundary line T_2 is in clockwise direction of the boundary line T_1. The relation holds, if the starting points are the same ($start(T_1) = start(T_2)$) and the constraint $angle(T_1) > angle(T_2)$ holds.

We put the following condition for generalizing constraint sets.

Definition 3 (Generalization condition) *The starting points of two constraint sets must be the same for generalizing the constraint sets.*

Since all positive instances in Figure 1 are the same, these instances can be generalized.

We find a least general generalization of the positive instances. By focusing on the clauses (15) and (16), the absorption rule produces the following clauses:

$$movable(C1) \circ \{\cdots\} \leftarrow$$
$$clockwise(T1, T2), \ldots, clockwise(T4 \bowtie [X, Y], T2), \ldots, .$$
$$movable(C2) \circ \{\cdots\} \leftarrow$$
$$clockwise(T1, T2), \ldots, clockwise(T4 \bowtie [X, Y], T2), \ldots, .$$

The RLGG is constructed as follows:

$$movable(C1 \oplus C2) \circ \{\cdots\} \leftarrow$$
$$clockwise(T1, T2), \ldots, clockwise(T4 \bowtie [X, Y], T2), \ldots, .$$

So far, the target of generalization is an original problem description. The next stage is to generalize clauses for some problem descriptions. As shown previously, a generalized clause for a

problem description has the same conditional parts. Generalization of the conditional parts are also performed by giving different problem descriptions.

Generalized clauses for each problem description is as follows:

$$movable(C1 \oplus C2) \diamond \{\cdots\} \leftarrow$$
$$clockwise(T1, T2), \ldots, clockwise(T4 \bowtie [X, Y], T2), \ldots, .$$
$$movable(C1 \oplus C2) \diamond \{\cdots\} \leftarrow$$
$$clockwise(T1, T2), \ldots, clockwise(T4 \bowtie [X, Y], T2), \ldots, .$$

Each clause is generalized and the following clause is produced:

$$movable(C1 \oplus C2) \diamond \{\cdots\} \leftarrow$$
$$clockwise(T1, T2), \ldots, clockwise(T4 \bowtie [X, Y], T2), \ldots, .$$

Simplifying the conditional part yields the following clause:

$$movable(C1 \oplus C2) \diamond \{\cdots\} \leftarrow$$
$$clockwise(T4 \bowtie [X, Y], T2).$$

In this case, we put the same instances for each problem description. The above clause can also be obtained from different problem descriptions.

6 Abstraction

This section describes a method of abstraction, which eliminates constraint sets in constraint logic programs. This method can be viewed as transformation from constraint logic programs to ordinally logic programs. This is also viewed as producing qualitative descriptions from quantitative data. A subset of qualitative descriptions corresponds to the extended problem descriptions. In this description, a constraint may be signified as a symbol. This symbol is called *abstract term*. For example, the constraint set of the boundary line $T1$ is represented as the abstract term $t1$.

Definition 4 (Abstraction) *Abstraction is defined as tranforming constraint logic programs to logic programs whose domain is Herbrand universe. Let τ be the abstraction operator. The transformation rules are as follows:*

1. *Abstracting a constraint set*

   If an abstact term T is assigned to the constraint set C, the transformation rule is as follows:
   $C \xrightarrow{\tau} T$

   Otherwise C is transformed to the abstract term which corresponds the minimum constraint set covering C.

2. *Abstracting function*

   Let T_i be the abstraction of C_i. If the function f is assigned to the abstract term f', the transformation rule is specified as follows:

   $f(C_1, \ldots, C_n) \xrightarrow{\tau} f'(T_1, \ldots, T_n))$

3. *Abstracting constrained atom*

   Let Y_i be the abstraction of X_i. If p is assigned to the abstract term p', the transformation rule is specified as follows:

   $p(X_1, \ldots, X_n) \diamond C \xrightarrow{\tau} p'(Y_1, \ldots, Y_n)$

4. *Otherwise the abstraction of X is X itself.*

Note that a constraint set is eliminated in abstraction. This is interpreted as substituing a constrained atom to the abstract term.

We put abstract terms as follows:

$$\begin{aligned}
\{X = -4 * L, Y = -2 * L, 0 \leq L \leq 1\} &\xrightarrow{\tau} t1. \\
\{X = -1 * L, Y = -3 * L, 0 \leq L \leq 1\} &\xrightarrow{\tau} t2. \\
\{X = 5 * L, Y = 4 * L, 0 \leq L \leq 1\} &\xrightarrow{\tau} t3. \\
\{X = -1 * L, Y = -4 * L, 0 \leq L \leq 1\} &\xrightarrow{\tau} t4. \\
C1 \oplus C2 &\xrightarrow{\tau} comb(C1, C2) \\
C \bowtie [x, y] &\xrightarrow{\tau} inv(C)
\end{aligned}$$

The constraint logic program shown in the previous section is abstracted as follows:

$$movable(comb(t2, t3)) \leftarrow clockwise(t2, inv(t4)).$$

This rule is read as follows:

> If the boundary line $t2$ is in clockwise direction of the symmetric line of $t4$, the movable region is the composition of $t2$ and $t3$.

7 Extension

Producing examples is to find the representative of a constraint set. A small set of examples improves the efficiency of learning process. For this purpose, we construct a most simple set of the extended problem descriptions through constraint transformation.

We use the following heuristics in constraint transformation:

- The principle of parsimony

  Before producing complex constraint sets, simple constraint sets such as symmetric, orthogonal and translation constraints should be generated.

- Fairness
  The constraint transformation is fairly achieved for each constraint set.

Fairness does not focus on particular constraints. This heuristics does not have bias in producing the extended problem descriptions.

8 Concluding Remarks

This paper proposed a framework for learning spatial relations from examples using constraint-directed generalization. Constraint-directed generalization is regarded as introducing a constraint system into inductive logic programming. Inductive logic programming provides a general framework in that background knowledge is used for describing the problem domain. Our framework is also general, because the constraint system is purely mathematical, and is not dependent on the particular domain. This property provides an apparent framework for developing realistic inductive learning systems.

References

[1] Bratko, I., Muggleton, S. and Versek, A.: Learning Qualitative Models of Dynamic Systems, *Proc. of the Eighth International Machine Learning Workshop*, 1991, pp. 385-388.

[2] Brooks, R. A.: Symbolic Reasoning Among 3-D Models and 2-D Images, *Artif. Intell.*, Vol. 17 (1981), pp. 285-348.

[3] Forbus, K. D.: Interpreting Observations of Physical Systems, *IEEE Trans. of Man and Cybernetics*, Vol. 18, 1987.

[4] Gabbrielli, M. and Levi, G.: Modeling answer constraints in Constraint Logic Programming, *Proc. of the Eighth International Conference on Logic Programming*, 1991, pp. 238-252.

[5] Heintze, N. *et. al.*: The CLP(R) Programmers' Manual, Technical Report, Computer Science Dept, Monash University, 1987.

[6] Hentenryck, P. V. and Deville Y.: The Cardinality Operator: A new Logical Connective for Constraint Logic Programming, *Proc. of the Eighth International Conference on Logic Programming*, 1991, pp. 745-759.

[7] Jaffar, J. and Lassez, J. L.: Constraint Logic Programming, *Proc. of the 14th ACM Principles of Programming Languages Conference*, 1987.

[8] Maloney, J. H., Borning, A. and Freeman-Benson, B. N.: Constraint Technology for User-Interface Construction in ThingLab II, *Proc. of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1987, pp. 381-388.

[9] Mozetic, I. and Holzbaur, C.: Integrating Numerical and Qualitative Models within Constraint Logic Programming, *Proc. of the 1991 International Symposium on Logic Programming*, 1991, pp. 678-693.

[10] Muggleton, S.: Inductive Logic Programming, *New Generation Computing*, Vol. 8, 1991.

[11] Ohwada, H. and Mizoguchi, F.: A Constraint Logic Programming Approach for Maintaining Consistency in User-Interface Design, *Proc. of the 1990 North American Conference on Logic Programming*, 1990, pp. 139-153.

[12] Page, C. D. Jr and Frisch, A. M.: Generalizing Atoms in Constraint Logic, *Proc of the Second International Conference on Principles of Knowledge Representation and Reasoning*, 1991, pp. 429-440.

[13] Plotkin, G. D.: A Note on Inductive Generalization, *Machine Intelligence*, Vol. 5, 1970, pp. 153-163.

[14] Shoham, Y.: Naive Kinematics: one aspect of shape, *Proc. of IJCAI'85*, 1985, pp. 436-442.

[15] Sussman, G. J. and Steel, G. L.: CONSTRAINTS - A Language for Expressing Almost-hierarchical Descriptions, *Artif. Intell.*, Vol. 14 (1980), pp. 1-39.

[16] Winston, P. H.: Learning Structural Descriptions from Examples, In Winston, P. H. (ed.), *The Psychology of Computer Vision*, McGraw-Hill, 1975.

Automated Debugging of Logic Programs via Theory Revision *

Raymond J. Mooney and Bradley L. Richards
Department of Computer Sciences
University of Texas
Austin, TX 78712
mooney@cs.utexas.edu, bradley@cs.utexas.edu

February 14, 1992

Abstract

This paper presents results on using a theory revision system to automatically debug logic programs. FORTE is a recently developed system for revising function-free Horn-clause theories. Given a theory and a set of training examples, it performs a hill-climbing search in an attempt to minimally modify the theory to correctly classify all of the examples. FORTE makes use of methods from propositional theory revision, Horn-clause induction (FOIL), and inverse resolution. The system has has been successfully used to debug logic programs written by undergraduate students for a programming languages course.

1 Introduction

Most of the recent work on inductive logic programming has focused on the synthesis of logic programs from examples, e.g. FOIL [Quinlan, 1990] and GOLEM [Muggleton and Feng, 1990]. There has been very little work on automated debugging of logic programs since Shapiro's original work on the subject [Shapiro, 1983]. Although systems like FOIL and GOLEM generally make use of existing subroutines, e.g. learning `reverse` given `append`, the supplied definitions are extensional rather than intensional and these systems can only

*This research was supported by the National Science Foundation under grant IRI-9102926 and by the NASA Ames Research Center under grant NCC 2-629. The second author was supported by the Air Force Institute of Technology faculty preparation program.

learn new theories and cannot revise existing ones. However, unlike Shapiro's system, recent systems do not rely on an omniscient oracle that is capable of determining the truth of arbitrary ground atomic formulae. Consequently, current systems require much less user interaction.

This paper presents results on using methods for first-order theory revision to automatically debug logic programs without the use of an oracle. We are developing a theory revision system, FORTE [Richards and Mooney, 1991], that modifies an incomplete and/or incorrect Horn-clause domain theory to fit a set of training examples. When the domain theory is viewed as a logic program, theory revision corresponds to automated debugging from I/O pairs. The design of FORTE has been influenced by a number of previous developments. Many of its basic revision operators (delete antecedent, add antecedent, delete rule, and add rule) are based on similar operators in EITHER, a predecessor of FORTE for propositional Horn-clause theories [Ourston and Mooney, 1990; Mooney and Ourston, 1991]. FORTE also makes use of a FOIL-like algorithm to add new rules and antecedents, and several inverse-resolution operators [Muggleton and Buntine, 1988] to generalize and compress the theory. The system has recently been augmented with a path-finding method for overcoming local-maxima [Richards and Mooney, 1992] and modified to handle recursion.

To test the resulting system, we collected buggy Prolog programs from students in an undergraduate course on programming languages. Students were asked to hand in their initial versions of programs for finding a path in a graph and inserting a new element after a specified element in a list. The collected programs included a total of 12 distinctly different buggy programs. FORTE was able to correctly debug all but one of these programs based on a relatively small set of training examples.

Since induction of complete programs is a very difficult problem, we believe automated debugging is a more realistic goal with greater potential for practical utility. Induction of programs in other languages has not been particularly successful. Even the most sophisticated systems have been able to produce only relatively toy programs [Barstow, 1988]. Using inductive techniques to revise buggy programs has not been explored nearly as well, and we believe our initial results in oracle-free inductive logic-program debugging are quite promising.

The remainder of the paper is organized as follows. Section 2 presents an overview of FORTE's basic revision algorithm. Additional details on the revision algorithm are presented in [Richards and Mooney, 1991; Richards and Mooney, 1992]. Section 3 presents results on automatically debugging actual student programs. Section 4 presents results on debugging subroutines using only examples for the main program. Section 5 discusses related work and Section 6 presents directions for future research. Section 7 presents our conclusions.

2 FORTE's Theory Revision Algorithm

In order to revise a logic program, one must first detect an error. FORTE attempts to prove all positive and negative instances in the training set using the current program. As in most ILP systems, positive (negative) instances are tuples of constants that should (should not) satisfy the goal predicate. When a positive instance is unprovable, some program clause needs to be generalized. All clauses that failed during the attempted proof are candidates for generalization. When a negative instance is provable, some program clause needs to be specialized. All clauses that participated in the successful proof are candidates for specialization.

When an error is detected, FORTE identifies all clauses that are candidates for revision. The core of the system consists of a set of operators that generalize or specialize a clause to correctly classify a set of examples. Based on the error, all relevant operators are applied to each candidate clause. The best revision, as determined by classification accuracy on the complete training set, is implemented. This process iterates until the theory is consistent with the training set or until FORTE is caught in a local maximum, i.e. none of the proposed revisions improve overall accuracy.

FORTE maintains a list, for each program clause, of all errors for which the clause may be responsible. After trying to prove all instances, the clauses are ordered by the number of errors for which they may be responsible. Note that we treat specialization and generalization separately, so each clause may appear twice in this ordered list. Revision begins with the clause that provides the greatest potential benefit.

2.1 Specializing the theory

When one or more negative instances are provable, the theory needs to be specialized. There are two basic ways to do this: delete a clause or add new antecedents to an existing clause. Deleting a clause is a simple operation. The clause identified in the revision point is deleted from the theory, and the remaining theory is tested on the training set.

Adding antecedents to a clause is more complex. Our goal is to eliminate incorrect proofs of negative instances without eliminating many (or any) of the correct proofs of positive instances. It may be necessary to create several specializations of an existing clause. For example, we might specialize the propositional rule

```
a :- b, c
```

in two different ways

```
a :- b,c,d,e
a :- b,c,f
```

FORTE employs two competing methods for adding antecedents. The first is similar to FOIL in that it adds antecedents one at a time, choosing at any point the antecedent that provides the best information gain. Our information metric is different from FOIL's in that it considers only the number of positive and negative instances – not the number of proofs of positives and negatives.

The second algorithm is called relational pathfinding [Richards and Mooney, 1992]. We designed this method based on the assumption that relational concepts are usually represented by a small number of fixed relational paths connecting the arguments of a predicate. Relational pathfinding is reminiscent of Quillian's spreading activation [Quillian, 1968]. In essence, the variables already present in the clause are treated as nodes in a graph. Relations in the predicate are treated as edges (predicates with arity greater than two are simply edges with more than two ends). THe graph associated with the overly-general clause is frequently disconnected into separate subgraphs. Relational pathfinding seeks to find a path of additional relations that can be used to join the distinct subgraphs into a coherent whole.

2.2 Generalizing the theory

When one or more positive instances cannot be proven, the theory needs to be generalized. There are several ways to do this in FORTE: deleting antecedents from an existing clause, adding a new clause, or using the inverse resolution operators identification or absorption.

As with clause specialization, several generalizations of a clause may need to be generated in order to provide proofs for all of the positives without letting negatives become provable. There are two approaches to deleting antecedents from a rule. The first approach, which works well in simple cases, is to delete antecedents singly, based on a simple information metric. However, in some cases, we may need to delete several antecedents simultaneously in order to gain anything. A simple example of this is an "m of n" type problem. Given the clause

```
a :- b, c, d, e, f
```

it may be that all of the antecedents are important, but positive instances are distinguished by satisfying any three of them. In this case, we would need to create the rules

```
a :- b, c, d
a :- b, c, e
a :- b, c, f
a :- b, d, e
```

and so forth. However, deleting a single antecedent may not help, since none of the positives may satisfy a four antecedent rule like

```
a :- b, c, d, e
```

In order to add a new clause, FORTE copies the clause identified in the revision point. It then deliberately over-generalizes this clause by deleting all antecedents whose deletion allows the proof of one or more previously unprovable positives – even though their deletion may also allow proofs of some negative instances. This overly general rule is then given to the antecedent addition algorithms described in the section on specialization.

Lastly, FORTE can use the inverse resolution operators identification and absorption to generalize a program. Identification allows alternate definitions of a particular literal to come into play. For example, suppose we have the following two clauses in our program.

```
a :- b, c, d
a :- b, x
-------------
x :- c, d
```

Assume c in the first clause was a failure point in an attempted proof of a positive instance. Identification replaces the first clause with the third clause shown. This does not affect current proofs, but it allows other definitions of x to generalize proofs that use the first clause.

Absorption is, in effect, the complement of identification. Suppose we have, in our program, the first two clauses shown below.

```
a :- b, c, d
x :- c, d
-------------
a :- b, x
```

Absorption notes the common antecedents and replaces the first clause with the third one. Again, this does not endanger any existing proofs, and has the effect of allowing alternate definitions of x to come into play. Absorption is particularly useful for developing recursive clauses, since x and a may be the same predicate.

3 Debugging Student Programs

In order to test FORTE's debugging ability, sample logic programs were collected from students taking an undergraduate course on programming languages. Students were given an assignment to write several short logic programs. The first problem involved finding a path in a directed graph, a standard test problem for FOIL [Quinlan, 1990]. Below is a correct program for this problem.

```
path(A,B):-edge(A,B).
path(A,B):-edge(A,C),path(C,B).
```

The second involved inserting an element in a list immediately following a specified item. Below is a correct program for this problem: [1]

```
insert_after([A|B],A,C,[A,C|B]).
insert_after([A|B],C,D,[A|E]) :- C\=A, insert_after(B,C,D,E).
```

where `insert_after(X,A,B,Y)` inserts `B` after the first occurrence of `A` in the list `X`. Students were asked to hand in the initial versions of their programs before running and debugging them. We received 3 distinctly different buggy programs for `path`, and 9 for `insert_after`.

FORTE was given each of these 12 programs to debug together with a training set of correct input-output pairs. The training set for `path` was a complete set of positive and negative examples for an 11-node graph (15 positive, 106 negative). The training set for `insert_after` contained 10 positive examples for lists up to size 3 and 23 selected negative examples. For this problem, FORTE is also given a definition for `components`.

The system was able to correctly debug all but one of the 13 programs. Debugging time took an average of 68 seconds for the `path` programs and 350 seconds for the `insert_after` examples running in Quintus Prolog on a Sun Sparcstation. One program was not properly debugged due to a local maximum. The revisions included deleting incorrect clauses, adding additional literals to clauses, and adding totally new clauses. Below is one of the buggy student programs for `path`:

```
path(A,B):-edge(B,A).
path(A,B):-edge(A,B).
path(A,B):-edge(A,C),edge(D,B),path(A,C).
```

The first clause is incorrect since a directed path is desired. Since it covers a number of negatives and no positives, FORTE retracts this clause as its first revision. Since the student wrote the recursive clause with two edges and a path, the case of a path of length two is not handled. As a result, FORTE adds the rule:

```
path(A,B) :- edge(A,C), path(C,B).
```

Finally, FORTE decides to delete the student's original recursive clause since the new clause covers all of the examples it covers. The result is the simple correct program presented earlier.

One of the buggy student programs for `insert_after` is shown below:

```
insert_after([A|B],C,D,[A|E]):-insert_after(B,C,D,E).
insert_after([A|B],A,C,[A,C|D]):-insert_after(B,A,C,D).
```

[1] It should be noted that FORTE uses a function-free representation like FOIL. A term shown as `[A|B]` must actually represented by an additional literal, `components(X,A,B)`, in the body of the clause. FORTE translates its results into normal Prolog notation for readability.

This program is missing a base case, so the first revision FORTE makes is to add the clause:

```
insert_after([A|B],A,C,[A,C|B]).
```

Next, the system adds the antecedent `C` $\neq$ `A` to the student's first clause to prevent answers that never insert the desired item. Finally, the student's second clause is deleted because the instructions were to insert `C` only after the first occurrence of `A`. The result is the correct version of `insert_after` previously presented.

4 Additional Debugging Examples

Since the students in the previous experiment were writing their first Prolog programs, the examples could not be too difficult. Like many ILP systems, FORTE can actually induce complete programs for such simple problems. Given an empty initial theory and the same data used to refine the buggy programs, FORTE can construct complete and correct definitions for `path` and `insert_after`. Consequently, it is not particularly surprising that it can also debug programs for these problems.

However, FORTE can also debug programs which, due to fundamental limitations or resource constraints, existing ILP systems cannot induce without an oracle. Since they cannot create new predicates, systems like FOIL and GOLEM cannot induce programs with recursive subroutines unless they are given extensional definitions of these subroutines. For example, they cannot produce a program for `reverse` given only a background definition for `components` of a list – they need an extensional definition of append. Although FORTE is also unable to induce programs with recursive subroutines from scratch, it can debug many incorrect definitions.

As an example of fixing a recursive subroutine, consider the following buggy definition of `subset`.

```
member(A,[A|B]).
member(A,[B|C]).
subset([],A).
subset([A|B],C) :- member(A,C), subset(B,C).
```

The second clause for `member` is missing the recursive literal `member(A,C)` from its body. FORTE successfully added this antecedent given only 64 positive and negative examples of `subset` and a definition of `components` – explicit examples of `member` were not necessary.

5 Related Work

As previously mentioned, most recent work in ILP has concerned the complete induction of logic programs from examples [Quinlan, 1990; Muggleton and Feng, 1990; Kijsirikul *et*

al., 1991]. Shapiro's Prolog debugger, PDS6 [Shapiro, 1983], uses many techniques from his learning system, MIS; however, it requires a great deal of user interaction. The user must be available to answer membership queries as well as provide other detailed information. FORTE requires only a sufficient supply of examples; no oracle or additional user interaction is required.

Work in automated debugging for other programming languages has primarily employed *static* methods that compare a program to a formal specification [Katz and Manna, 1976] abstract program plan [Johnson, 1986], or existing correct program [Murray, 1988]. By comparison, PDS6 and FORTE are *dynamic*. They run a program on specific examples, detect errors, and use them to revise the program.[2] Consequently, dynamic methods require only partial, extensional definitions of programs. This is an important advantage since formal specifications are frequently unavailable. Systems that require an existing correct program (e.g. TALUS [Murray, 1988]) are primarily useful in tutoring environments, since a correct program is rarely available in other situations.

Most other work in theory revision is propositional in nature, and therefore inapplicable to logic programming [Ginsberg, 1990; Towell and Shavlik, 1991; Cain, 1991]. FOCL [Pazzani *et al.*, 1991] uses an initial theory to guide a FOIL-based system; however, it produces a flat, operationalized definition instead of a revised theory. A version of FOCL that performs theory revision has been developed [Pazzani and Brunk, 1990]; however, it requires significant user interaction. Finally, FOCL has not been tested on logic programming problems and it is unclear how its operationalization procedure would handle recursion.

6 Future Work

As with all existing ILP systems, the problems currently used to test FORTE are quite simple. Consequently, we plan to test FORTE's ability to debug more difficult programs. An interesting problem we are considering is debugging a Prolog implementation of ID3 [Bratko, 1990]. Since debugging is normally much easier than program synthesis, we believe FORTE should be able to handle larger problems than purely inductive systems. We also plan to test FORTE on other real-world problems such as revising qualitative models of complicated systems [Bratko *et al.*, 1991; Richards and Mooney, 1992] and revising the Chou-Fassman theory for protein folding [Maclin and Shavlik, 1991].

Like many other ILP systems (e.g. GOLEM, FOIL), FORTE is unable to create new predicates. Current predicate-invention methods such as inverse resolution [Muggleton and Buntine, 1988] are computationally very demanding and usually employ an oracle. Efficient oracle-free methods for predicate invention are needed to revise programs that require additional recursive subroutines.

[2]The terms *static* and *dynamic* are borrowed from [Murray, 1988].

We are also developing techniques for learning search heuristics [Mitchell, 1984; Cohen, 1990] to improve the efficiency of logic programs. Meta-rules for when to use a particular clause can be empirically learned using sample calls for which the clause ultimately failed or succeeded in leading to a final solution. Such examples can be extracted from the search conducted during the execution of a logic program [Cohen, 1990]. Existing ILP systems should be useful for learning search heuristics from these examples. As an example, consider the following exponential-time sorting program:

```
naivesort(X,Y) :- permutation(X,Y), ordered(Y).

permutation([],[]).
permutation([X|Xs],Ys) :- permutation(Xs,Ys1), remove(X,Ys,Ys1).

remove(X,[X|Xs],Xs).
remove(X,[Y|Ys],[Y|Ys1]) :- remove(X,Ys,Ys1).

ordered([_X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

The predicate `remove(X,Y,Z)` is true if removing *one* of the occurrences of the item `X` from the list `Y` leaves the list `Z`. The predicate `permutation` actually uses `remove` to insert an element into a list at every possible position. Using examples for when each of the clauses for `remove` leads to a success or failure, it should be possible to learn the following heuristics: the base case ultimately leads to a solution when `Xs` = [] or when `Xs` = `[Y|Ys]` and `X` $\leq$ `Y`; the recursive clause leads to a solution when `X` > `Y`. If these constraints are folded into the existing rules, the resulting definition for `remove` is:

```
remove(X,[X],[]).
remove(X,[X,Y|Ys],[Y|Ys]) :- X =< Y.
remove(X,[Y|Ys],[Y|Ys1]) :- X > Y, remove(X,Ys,Ys1).
```

When this new procedure is used by `permutation`, it always inserts the element `X` before the first element of `Ys1` that is greater than it. Upon inspection, it is clear that the result is an insertion sort, where `permutation` always returns a sorted permutation and the `ordered` check is redundant. Consequently, by learning heuristics for when to use each of the clauses for `remove`, we have turned an $O(n!)$ sorting algorithm into a $O(n^2)$ one! We are currently developing an ILP system that learns such heuristics using a FOIL-based inductive learner.

7 Conclusions

Automated program debugging is an area of ILP that has not been extensively explored. Shapiro's original work in this area has not been followed-up nearly as well as his work on

induction of complete programs. We believe that recent developments in first-order induction and theory revision hold great promise in developing dynamic automated debuggers for logic programming. Initial results on using our theory revision system, FORTE, to debug logic programs is quite promising. It is already capable of debugging actual student programs for simple problems without any user interaction. We plan to extend our tests to larger, more realistic problems and to develop effective learning methods for improving the speed as well as the accuracy of logic programs.

Acknowledgements

We would like to thank John Zelle for providing the details of the naive-sort example.

References

[Barstow, 1988] D. Barstow. Artificial intelligence and software engineering. In H.E. Shrobe, editor, *Exploring Artificial Intelligence*. Morgan Kaufman, San Mateo, CA, 1988.

[Bratko *et al.*, 1991] I. Bratko, S. Muggleton, and A. Varsek. Learning qualitative models of dynamic systems. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 385–388, Evanston, IL, June 1991.

[Bratko, 1990] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, Reading:MA, 1990.

[Cain, 1991] T. Cain. The DUCTOR: A theory revision system for propositional domains. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 485–489, Evanston, IL, June 1991.

[Cohen, 1990] W. W. Cohen. Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 268–276, Austin, TX, June 1990.

[Ginsberg, 1990] A. Ginsberg. Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 777–782, Detroit, MI, July 1990.

[Johnson, 1986] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Pitman Publishing, London, 1986.

[Katz and Manna, 1976] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the Association for Computing Machinery*, 19(4):188–206, 1976.

[Kijsirikul *et al.*, 1991] B. Kijsirikul, M. Numao, and M. Shimura. Efficient learning of logic programs with non-determinant, nondiscriminating literals. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 417–421, Evanston, IL, June 1991.

[Maclin and Shavlik, 1991] R. Maclin and J. W. Shavlik. Refining domain theories expressed as finite-state automata. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 524–528, Evanston, IL, June 1991.

[Mitchell, 1984] T. M. Mitchell. Toward combining empirical and analytic methods for learning heuristics. In A. Elithorn and R. Banerji, editors, *Human and Artificial Intelligence*. North-Holland, Amsterdam, 1984.

[Mooney and Ourston, 1991] R. J. Mooney and D. Ourston. A multistrategy approach to theory refinement. In *Proceedings of the International Workshop on Multistrategy Learning*, pages 115–130, Harper's Ferry, W.Va., Nov. 1991.

[Muggleton and Buntine, 1988] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, Ann Arbor, MI, June 1988.

[Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990. Ohmsha.

[Murray, 1988] W. R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing, London, 1988.

[Ourston and Mooney, 1990] D. Ourston and R. Mooney. Changing the rules: a comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 815–820, Detroit, MI, July 1990.

[Pazzani and Brunk, 1990] M. Pazzani and C. Brunk. Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning. In *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, October 1990.

[Pazzani *et al.*, 1991] M. Pazzani, C. Brunk, and G. Silverstein. A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 432–436, Evanston, IL, June 1991.

[Quillian, 1968] M. R. Quillian. Semantic memory. In M. Minksy, editor, *Semantic Information Processing*. MIT Press, Cambridge, MA, 1968.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Richards and Mooney, 1991] B. Richards and R. Mooney. First-order theory revision. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 447–451, Evanston, IL, June 1991.

[Richards and Mooney, 1992] B.L. Richards and R.J. Mooney. Learning relations by path finding. In *submitted paper*, 1992.

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

[Towell and Shavlik, 1991] G. Towell and J. Shavlik. Refining symbolic knowledge using neural networks. In *Proceedings of the International Workshop on Multistrategy Learning*, pages 257–272, Harper's Ferry, W.Va., Nov. 1991.

Correcting Multiple Faults in the Concept and Subconcepts by Learning and Abduction

Somkiat Tangkitvanich, Masayuki Numao and Masamichi Shimura
Department of Computer Science, Tokyo Institute of Technology
2-12-1 Oh-okayama, Meguro, Tokyo 152, Japan
e-mail: {sia,numao,shimura}@cs.titech.ac.jp

Keywords: theory refinement, learning from theory and examples, inductive logic programming, abduction

Abstract

The problem of correcting faults in the subconcepts of a theory is very important, but is so far given little attention from the study of theory refinement. This paper presents a system that automatically refines the theory expressed in the function-free first-order logic. Our system can efficiently correct multiple faults in all the concept and subconcepts of the theory, given only the classified examples of the concept. Our system first learns the correct operational definitions of the concept by using a combination of an explanation-based and an inductive learning algorithms. It then uses the operational definitions to infer the correct definitions of the concept and subconcepts by means of abduction. The system has been successfully tested in refining a theory in chemistry.

1 Introduction

Knowledge acquisition is recognized as a major bottleneck in the development of knowledge-based systems. The knowledge or theory elicited from an expert tends to be only approximately correct. By theory refinement, the approximate knowledge is transformed into a more complete and correct one. Automatic theory refinement increases the efficiency and quality in constructing a knowledge-based system.

Since the given theory is nearly correct and complete, it is generally assumed that the structure of the refined theory is similar to that of the given theory. In other words, the given and the refined theories should be defined by the same set of concept and subconcepts. Thus, the goal of theory refinement is to correct the faults in all the concept and subconcepts.

Although a large number of theory refinement systems have been developed, the problem of correcting faults in the subconcepts is given relatively little attention. In some existing systems (e.g., IOE[1], IVSM[3] and FOCL[4]), refining the theory is reduced to refining the operational definitions of the concept. However, refining only the operational definitions leaves the subconcepts remain incorrect. In other systems (e.g., PDS[6]), it is assumed that the training examples of the subconcepts are also given. However, giving the examples of all the concept and subconcepts burdens the expert too much.

The problem of our interests is to correct faults in all the concept and subconcepts of the theory when only the examples of the concept are given. We present a system called $\mathcal{RLA}$ (Refinement by Learning and Abduction) that can correct a combination of faults in a relational theory, or a theory represented in the function-free first-order logic.

The approach used in our system is based on learning and abduction. First, the system learns the correct operational definitions of the concept by using a combination of an explanation-based and an inductive learning algorithms. It then uses these operational definitions to infer the correct definitions of all the concept and subconcepts by abduction.

The remainder of the paper is organized as follows. Section 2 describes how to learn the correct operational definitions of the concept from its examples and an approximate theory. Section 3 explains how the operational definitions are used to to infer the correct theory. The next section gives an example of how the system refines a simple theory. Then, the paper reports the results of testing the system by using a theory in chemistry. The final section compares $\mathcal{RLA}$ with related systems.

2 Learning Correct Operational Definitions

In $\mathcal{RLA}$, the operational definitions are learned by using a combination of an explanation-based and the FOIL inductive learning algorithms. During the operationalization, a modification of FOIL's Gain heuristic is used to detect faults in the theory.

2.1 FOIL

FOIL[5] is an automatic inductive learning system that learns function-free Horn clauses from the examples represented in the form of tuples. In the outermost level, FOIL uses the covering strategy. In the inner level, it uses the divide-and-conquer strategy to grow a clause by adding literals one by one to the body part.

FOIL uses an information-based heuristic to select the most promising literal during a clause construction. The whole purpose of a clause is to characterize a subset of the positive tuples. Therefore, it is appropriate to focus on the information provided by signalling that a tuple is one of the positive kind. If the current training set, T_i, contains T_i^+ positive tuples and T_i^- negative tuples, the information required for this signal from T_i is given by

$$I(T_i) = -log_2(T_i^+/(T_i^+ + T_i^-)).$$

If the selection of a particular literal L_i extends T_i to T_{i+1}, the information given by the same signal is similarly

$$I(T_{i+1}) = -log_2(T_{i+1}^+/(T_{i+1}^+ + T_{i+1}^-)).$$

Suppose that T_i^{++} of the positive tuples in T_i have extensions in T_{i+1}. The total information we have gained regarding the positive tuples in T_i is given by the number of them that have extensions, T_i^{++}, multiplied by the information gained regarding each of them, i.e.,

$$Gain(L_i) = T_i^{++} \times (I(T_i) - I(T_{i+1})).$$

According to Quinlan, Gain is negative if the positive tuples are less concentrated in

T_{i+1} than in T_i, and is small if either the concentrations are similar or few tuples in T_i satisfy L_i. Using the Gain heuristic, FOIL searches from general to specific by adding a literal that mostly discriminates the positive from negative tuples.

2.2 Detecting Incorrect Literals

Since the given theory is only approximately correct, its rules may contain some incorrect literals. We now describe an approach to detect them. The approach is based on measuring the *usefulness* of a literal.

Definition 1 *The usefulness of a literal L_i*

A literal L_i is useful if it is in the space of literals of FOIL, and if

- *its normalized Gain, $(I(T_i) - I(T_{i+1}))$, is greater than the given Gain threshold, θ, or,*
- *it is a generate literal for another literal, and the combination of the two literals is useful.*

If a literal is not useful, it is useless.

A generate literal is a literal that when it is deleted from a clause, the body of the clause has some variables underivable from those in the head. In other words, a generate literal produces new variables for the other literals in the clause. In our implementation, θ has the default value of 0.25, but it is adjustable according to the certainty of the correctness of the given theory.[1] When θ is set to a large value, $\mathcal{RLA}$ refines the theory radically, i.e., it distrusts the given theory, tends to discard a rule and rebuilds a new one. On the other hand, if θ is set to a small value, $\mathcal{RLA}$ refines the theory conservatively. By varying θ, $\mathcal{RLA}$ can refine a theory radically or conservatively as desired.

[1] This default value corresponds to about 1.2 times increase in the concentration of the positive tuples in the total tuples.

The usefulness of a literal is measured by its normalized Gain, rather than its Gain, since the value of the normalized Gain does not depend on the size of the training examples. On the other hand, the value of Gain can be large in case of a large training set, even though the literal poorly discriminates the examples. By measuring the normalized Gain, we can detect the useless literals which are considered as certain kinds of faults. Examples of the useless literals are the literals that are irrelevant to the concept, and the literals which are redundant with the existing literals. Such useless literals can be easily detected since they exhibit a low normalized Gain. A non-terminating recursive literal with no established partial ordering among the variables is also a useless literal since it is not in the literal space of FOIL.

Next, we define the Gain of a clause, another important idea used in our algorithm.

Definition 2 *The Gain of a clause C*

$$Gain(C) = T_0^{++} \times (I(T_0) - I(T_C)).$$
$$I(T_0) = -log_2(T_0^+/(T_0^+ + T_0^-)).$$
$$I(T_C) = -log_2(T_C^+/(T_C^+ + T_C^-)).$$

where T_0^+ and T_0^- are the number of the positive and negative examples in the remaining training set T_0, respectively. T_0^{++} is the number of the positive examples in T_0 that have extensions in T_C. T_C^+ and T_C^- are the number of the positive and negative tuples that satisfy C, respectively.

According to the definition, the Gain of a clause C becomes greater as more positive examples and less negative examples satisfy the clause.

2.3 Learning Correct Operational Definitions

Now we are ready to describe the algorithm that learns the correct operational definitions. Algorithm1, as shown in Figure 1, is based on a combination of FOIL and an explanation-

based algorithms. It consists of three main steps: operationalization, specialization, and rule creation.

Algorithm1

Input: A theory Th defined by a set of concept and subconcepts, and a set of training examples

Output: 1. A set of operational definitions, $Op_i \cup L_i$'s, which are consistent with the training examples

2. A theory Th' which is derived from Th by deleting all the useless literals

1) Operationalization: Operationalize Th by expanding the subconcepts, resulting in the operational definitions, Op_i's.

1.1 Among the remaining rules of a subconcept, select the one that has biggest value of Gain. Consider only the useful literals when measuring Gain of a rule.

1.2 Do not expand an internal call of a recursion.

1.3 Let Th' be a theory derived from Th by deleting all the useless literals.

2) Specialization: Use FOIL to choose a set of literals L_i such that every $Op_i \cup L_i$ does not cover any negative examples.

3) Rule creation: If there are some positive examples uncovered by the existing operational rules, then use FOIL to create more rules by choosing sets of literal L_i's.

Figure 1: The algorithm for learning correct operational definitions

In the first step, the concept operational definitions, Op_i's, are generated by expanding the subconcepts. During the selection of the rules to expand, the algorithm removes the faults detected as useless literals. When all the useless literals are removed, rules having only these literals are deleted. Among the remaining rules, the rule with the greatest value of Gain is selected first.

In the second step, a set of literals L_i's are added to the overly general Op_i's so that the $(Op_i \cup L_i)$'s are no longer overly general. The literals are selected from the space of

all the operational literals and the literals that represent the subconcepts.

In the third step, more operational definitions of the concept are created if the existing ones do not cover all the positive examples.

3 Inferring the Theory from its Operational Definitions

In the last section, we described how to derive a theory with no useless literals and the correct operational definitions. In this section, we describe how they are used to infer the correct theory which is the output of the refinement system. The algorithm that performs this task is shown in Figure 2.

Algorithm2
Input: 1. A set of operational definitions, $Op_i \cup L_i$'s output from Algorithm1
2. A theory without any useless literals, Th', output from Algorithm1
Output: A theory Th'' which is defined by the set of concept and subconcepts of Th' but can be operationalized to $Op_i \cup L_i$'s

1) Find a correction set $C = C_1 \cup C_2$ where

$C_1 = \{add(sc_k, l_{ij})\}$ where the addition of the literal l_{ij} to the k^{th} rule of the subconcept sc offers an account for the addition of $l_{ij} \in L_i$ in the second step of Algorithm1, and

$C_2 = \{add(sc_{new}, l_{ij})\}$ where the addition of l_{ij} to a new rule of the subconcept sc offers an account for the addition of $l_{ij} \in L_i$ in the third step of Algorithm1.

2) For each individual correction $c \in C$ starting from the ones with lower preference values to the ones with higher preference values,

If the correction set $C-\{c\}$ can still derive all $Op_i \cup L_i$'s then $C = C-\{c\}$.

3) Refine Th' according to C, resulting in Th''.

Figure 2: The algorithm for inferring the correct theory

The main idea of the algorithm is to determine to which subconcept each literal added

during step 2 and 3 in Algorithm1 belongs. This is done by abduction; the reasoning that finds an explanation of a set of data. Here, the operational definitions that are underivable from the input theory are viewed as the data to be explained. An explanation is a set of corrections in the theory that makes the operational definitions derivable. An individual correction is represented in the form of $add(sc_n, l_{ij})$, which suggests the addition of the literal l_{ij} to the body of the n^{th} rule of the subconcept sc. The rule can be an existing rule or a new one.

First, the algorithm collects a set of the corrections. For an existing operational definition $Op_i \cup L_i$, an individual correction is selected from a set of $add(sc_k, l_{ij})$, where sc_k is a subconcept rule that is expanded to $Op_i \cup L_i$, and $l_{ij} \in L_i$. For a newly created operational definition, an individual correction is selected from a set of $add(sc_{new}, l_{ij})$, where sc_{new} is a new rule of sc, and $l_{ij} \in L_i$. An individual correction $add(sc_n, l_{ij})$ is selected if $l_{ij} \in L_i$, for all operational definitions, $Op_i \cup L_i$'s, expanded from sc_n.

At this point, if the theory is refined according to the correction set, all the operational definitions will be derivable. However, since there may be superfluous corrections, the algorithm removes them from the correction set. The algorithm attempts to remove the correction that has a lower preference value before the one that has a higher preference value. Since we usually prefer smaller number of corrections to larger one, it is appropriate to assign the preference value in a way that helps minimize the number of corrections. Assuming no mutual recursions, we specify the comparative preference values over any pair of corrections as:

- For existing rules sc_i and sc'_j, if sc calls sc' as a subconcept then $add(sc_i, l)$ is preferred to $add(sc'_j, l)$.
- For new rules sc_i and sc'_j, if sc calls sc' as a subconcept then $add(sc'_j, l)$ is preferred to $add(sc_i, l)$.

- Otherwise, the comparative preference values of $add(sc_i, l)$ and $add(sc'_j, l)$ are decided arbitrarily.

4 Example of a Refinement

As an example of how $\mathcal{RLA}$ works, consider Figure 3a and 3b which shows the correct cup theory and the theory to be refined, respectively. The theory to be refined has multiple faults: some are in the concept while others are in the subconcepts. Inputs to the Algorithm1 are the incorrect theory and enough number of the examples of cup. The classified examples of the `liftable` and `stable` subconcepts are not given. Some examples given are shown in Figure 4. With all the necessary inputs, the theory is refined in the following steps:

```
cup(X):-
   insulate_heat(X),
   stable(X),
   liftable(X),
cup(X):-paper_cup(X),
stable(X):-bottom(X,B),
   flat(B).
stable(X):-bottom(X,B),
   concave(B).
stable(X):-has_support(X).
liftable(X):-has(X,Y)
   handle(Y).
liftable(X):-
   small(X),
   made_from(X,Y),
   low_density(Y).
```

(a) Correct theory

```
cup(X):-
   red(X),/* irrelevant */
   stable(X),
   liftable(X),
/* missing rule */
stable(X):-bottom(X,B),
   flat(B).
stable(X):-bottom(X,B),
   concave(B).
/* missing rule */
liftable(X):-has(X,Y)
   handle(Y).
liftable(X):-
/* missing literal */
   made_from(X,Y),
   low_density(Y),
   elastic(Y)./* irrelevant */
```

(b) Theory to be refined

```
cup(X):-
/* missing literal */
   stable(X),
   liftable(X),
/* missing rule */
stable(X):-bottom(X,B),
   flat(B).
stable(X):-bottom(X,B),
   concave(B).
/* missing rule */
liftable(X):-has(X,Y)
   handle(Y).
liftable(X):-
/* missing literal */
   made_from(X,Y),
   low_density(Y).
```

(c) Theory without useless literals

Figure 3: Cup theory

Positive Examples

```
P1 :  cup(c1).  has(c1,h1).  handle(h1).  bottom(c1,b1).  flat(b1).  green(c1).
P2 :  cup(c2).  small(c2).  made_from(c2,aluminium).  red(c2)
        low_density(aluminium).  bottom(c2,b2).  concave(b2).
P3 :  cup(c3).  paper_cup(c3).  green(c3).
                              .......
```

Negative Examples

```
N1 :  neg(cup(n_c1)).  has(n_c1,n_h1).  handle(n_h1).  bottom(n_c1,n_b1).
        convex(n_b1).  red(n_c1).  light(n_c1).
N2 :  neg(cup(n_c2)).  made_from(n_c2,aluminium).  low_density(aluminium).
        bottom(n_c2,n_b2).  concave(n_b2).  big(n_c2).
                              .......
```

Figure 4: Training Examples

1. Firstly, the concept cup is operationalized by expanding `stable` and `liftable`. During rule selections, Algorithm1 deletes all the faults detected as the useless literals. The faults deleted are `red` and `elastic` which are irrelevant to `cup`. The outputs of this step are the following four operational definitions:

   (1) `cup(X) :- bottom(X,B),flat(B),has(X,Y),handle(Y).`

   (2) `cup(X) :- bottom(X,B),flat(B),made_from(X,Y),low_density(Y).`

   (3) `cup(X) :- bottom(X,B),concave(B),has(X,Y),handle(Y).`

   (4) `cup(X) :- bottom(X,B),concave(B),made_from(X,Y),low_density(Y).`

2. Next, FOIL specializes the operational definitions by adding `insulate_heat(X)` to (1) and (3), and adding `small(X)` and `insulate_heat(X)` to (2) and (4).

3. Then, more operational definitions are created since there are some positive examples unprovable by the existing definitions. The operational definitions are now

extended to the followings. The underlined literals are the literals that are added by FOIL.

(1) cup(X) :- bottom(X,B),flat(B),has(X,Y),handle(Y),insulate_heat(X).

(2) cup(X) :- bottom(X,B),flat(B),made_from(X,Y),low_density(Y), small(X),insulate_heat(X).

(3) cup(X) :- bottom(X,B),concave(B),has(X,Y),handle(Y),insulate_heat(X).

(4) cup(X) :- bottom(X,B),concave(B),made_from(X,Y),low_density(Y), small(X),insulate_heat(X).

(5) cup(X) :- has_support(X),liftable(X),insulate_heat(X).

(6) cup(X) :- paper_cup(X).

At this point, the theory with no useless literals in Figure 3c and the correct operational definitions are output from Algorithm1.

4. Next, Algorithm2 determines to which subconcept each literal added by FOIL belongs. This is done by abduction. First, the algorithm finds a set of individual corrections. In this case, the correction set C is the union of:

$$\begin{aligned} C_1 = \{&add(cup1, insulate\_heat(X)), add(stable1, insulate\_heat(X)), \\ &add(liftable1, insulate\_heat(X)), add(stable2, insulate\_heat(X))\} \\ &add(liftable2, small(X)), add(liftable2, insulate\_heat(X)), \} \end{aligned}$$

and

$$\begin{aligned} C_2 = \{&add(cup2, has\_support(X)), add(stable3, has\_support(X)), \\ &add(cup2, liftable(X)), add(cup2, insulate\_heat(X)), \\ &add(stable3, insulate\_heat(X)), add(cup3, paper\_cup(X))\}. \end{aligned}$$

Then, the algorithm attempts to remove the superfluous corrections from C in the order of preference value from low to high. Here, $add(cup1, insulate\_heat(X))$ is preferred to the other corrections in C_1 that add $insulate\_heat(X)$ to the existing rules, and $add(stable3, has\_support(X))$ is preferred to $add(cup2, has\_support(X))$. After the removal, C is reduced to

$$\begin{aligned} C \;=\; & \{add(cup1, insulate\_heat(X)), add(liftable2, small(X)), \\ & add(stable3, has\_support(X)), add(cup3, paper\_cup(X))\}. \end{aligned}$$

Finally, the theory in Figure 3c is refined according to the corrections in C, resulted in the theory in Figure 3a.

5 Experimental Evaluation

Currently, $\mathcal{RLA}$ is implemented in IF-Prolog on a SUN SPARC Station I. To evaluate the system, we made a refining experiment by using a theory of buffer solutions which is a theory in chemistry. The purpose of the theory is to classify whether a given solution consisting of two solutes (or dissolved substances) is a buffer solution. A buffer solution is a solution that maintains nearly constant pH despite the addition of a small amount of acid or base. The theory is constructed by using FOIL to learn the concept from examples selected by an expert. The expert then reorganizes the theory into the predefined concept and subconcepts, some of which have recursive definitions.

The theory is then modified to include some faults by randomly applying the following four operators: the rule-deletion, rule-addition, literal-deletion and literal-addition operators. The literal-addition operator randomly adds a literal from a set of all literals and binds at least one variable in the literal with existing variables in the clause. The rule-addition operator constructs a rule by iteratively using the literal-addition operator. Each operator is applied to the theory with equal probability, α. The probability ranges

Modification probability (α)	Concept error(%)		Subconcept error(%)		Total error(%)	
	Initial	Refined	Initial	Refined	Initial	Refined
0.05	20.0	0.0	21.4	0.0	20.9	0.0
0.10	45.7	5.7	35.7	21.4	39.0	16.1
0.15	62.8	11.4	40.0	30.0	47.6	23:8
0.20	45.7	11.4	48.5	32.8	47.6	25.7

Table 1: Comparison of the error rates in the initial and refined theories

from 0.05 to 0.20. The modified theory may contain a combination of faults.

Inputs to $\mathcal{RCA}$ are the modified theory and four selected positive examples of the top-most concept. The negative examples are automatically generated by using the closed-world assumption. The total number of the examples used is 288. The classified examples of the subconcepts are not given. To see how many errors the system can decrease, we compare the input theory with the refined theory by using 7 test examples of the concept and 14 examples of the subconcepts. Note that the examples of the subconcepts are also needed to test whether the subconcepts are correctly refined. Otherwise, it is impossible to distinguish two theories that have the same operational definitions but have different structures. Table 1 shows the experiment results averaged on five trials for each value of α.

The experiment shows that $\mathcal{RCA}$ can effectively refine a theory that has a high error rate; for example, it can reduce the error rate as high as 21% to 0.0%. The experiment also shows that the classification errors of the refined theory gradually increase as the errors in the initial theory increase. Most of the errors are due to the errors in the subconcepts; our system cannot decrease the subconcept errors as effectively as decrease the concept errors. An investigation reveals that the subconcept refinement is rather ineffective when the initial theory is severely modified and all the rules of a subconcept are deleted. When all rules of a subconcept are deleted, it is impossible to construct them

without constructive induction. Adding this ability to our system is a topic for the future research.

It is important to note that when the error rate of the initial theory is almost 50%, $\mathcal{RLA}$ can still learn the definitions of the top-most concept with less errors than when no theory is given[2]. The less error rates can be accounted by the fact that the hypothesis space of the learning is changed when the theory is given. Take the case of a theory that has a subconcept with a recursive definition for example. In this case, if the theory with that subconcept is not given, it is impossible to learn a finite number of rules of the top-most concept that classify all the test examples correctly.

6 Comparison with Related Work

In this section, we compare $\mathcal{RLA}$ with the theory refinement systems that are based on related approaches.

6.1 R_X (Refinement by eXample)

R_X [7], the predecessor of $\mathcal{RLA}$, is a system that can refine a relational theory with multiple faults. Both $\mathcal{RLA}$ and R_X similarly refine a theory by learning the operational definitions before inferring the correct theory. However, $\mathcal{RLA}$ improves over R_X in many aspects. The main improvement is that, when constructive induction is not needed, $\mathcal{RLA}$ can create the subconcept rules that are missing from the original theory while R_X cannot. Thus, the theory refined by $\mathcal{RLA}$ is more similar to the original theory than the one refined by R_X. The experiment on the same theory and examples also shows that the theory refined by $\mathcal{RLA}$ is considerably more accurate than the one refined by R_X. Furthermore, since $\mathcal{RLA}$ is based on the idea of abduction which has been extensively studied, its behaviors are better understood than that of R_X, which is based on a specific algorithm.

[2]With the same training examples but without an input theory, the concept definitions are learned with an error rate 14.2% while the subconcepts cannot be learned.

6.2 Ginsberg's system

Ginsberg's theory refinement system[2] also refines a theory by first learning the operational definitions. The main difference between his system and $\mathcal{RLA}$ is that his system can refine only the theory represented in the propositional logic, the language with lower expressive power than the first-order logic used in our system. Like R_x, his system is based on an adhoc approach which makes its behaviors difficult to be predicted.

7 Conclusion

We discussed the problem of automatic theory refinement when multiple faults exist in the theory. We then presented the $\mathcal{RLA}$ refinement system that is based on the ideas of learning and abduction. Our system can overcome many difficulties in the existing systems. Particularly, it can correct a combination of faults in both the concept and subconcepts when only the examples of the concept are given. The experiment shows that $\mathcal{RLA}$ can effectively refine a theory with multiple faults.

An important topic for further research is to incorporate constructive induction into our system. This will address its inability to learn a subconcept that is missing from the given theory. Another research topic is to extend $\mathcal{RLA}$ so that it can refine a theory when the training examples are noisy.

Acknowledgement

We would like to thank Prof. Quinlan for permitting us to experiment with FOIL. We are also indebted to Boonserm Kijisirikul for his fruitful discussions. We also acknowledge the helps of Surapan Meknavin and Kenjirou Tsuji. Finally, we thank Dr. Somchai Akaratiwa, a chemist, for his help with the buffer solution theory.

References

[1] N. Flann and T. Dietterich. A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226, 1989.

[2] A. Ginsberg. Theory reduction, theory revision, and retranslation. In *AAAI 90*, pages 777–782, California, 1990. Morgan Kaufman.

[3] H. Hirsh. Combining empirical and anytical learning with version space. In *Proc. the Sixth International Workshop on Machine Learning*, pages 29–33, California, 1989. Morgan Kaufman.

[4] M. Pazzani, C. Brunk, and G. Silverstein. A knowledge-intensive approach to learning relational concepts. In *Proc. the Eighth International Workshop on Machine Learning*, pages 432–436, California, 1991. Morgan Kaufman.

[5] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[6] E. Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, MA, 1982.

[7] S. Tangkitvanich and M. Shimura. Refining theory with multiple faults. *IEICE transaction on Information and Systems*, 1992. To be appeared.

Learning Optimal KRK Strategies

Michael Bain
The Turing Institute Limited
36 North Hanover Street
Glasgow G1 2AD
UK

e-mail: michael@turing.ac.uk

Abstract

Omniscient databases for chess endgames can be constructed by full-width backward chaining from won positions. Complete tabulations are available for certain endgames, including King and Rook against King (KRK), which contain optimal depth-to-win information. Previous work has applied decision-tree learning to construct rules for the classification of positions from such databases as won or not won. The current work takes an Inductive Logic Programming approach, using methods of generalisation in first-order logic and specialisation by predicate invention. Results are given on learning rules from example black-to-move KRK positions which are won-for-white in a fixed number of moves.

1 Motivation and First Results

Is it possible for a machine to learn to play chess optimally given only example positions and some simple facts about the geometry of the board ? Below is part of such a program, in Prolog, learned under the stated conditions.

```
krk(0,Kfile,3,WRfile,1,Kfile,1) :- not(wrfile_threat(Kfile,WRfile)).
krk(0,c,WKrank,a,WRrank,a,BKrank :- wrrank_safe(WKrank,WRrank,BKrank).

wrfile_threat(BKfile,WRfile) :- diff(BKfile,WRfile,d1).

wrrank_safe(2,WRrank,1) :- lt(2,WRrank).
wrrank_safe(Krank,WRrank,Krank) :- diff(Krank,WRrank,d2).
wrrank_safe(Krank,WRrank,Krank) :- diff(Krank,WRrank,d3).
wrrank_safe(Krank,WRrank,Krank) :- diff(Krank,WRrank,d4).
wrrank_safe(Krank,WRrank,Krank) :- diff(Krank,WRrank,d5).
wrrank_safe(Krank,WRrank,Krank) :- diff(Krank,WRrank,d6).
wrrank_safe(1,8,1).
```

This program is complete and correct for the canonical set of legal Black-to-move (BTM) positions in the King and Rook against King (KRK) endgame which are won-for-White (wfw) at a depth of 0 moves (i.e. checkmate). The top-level predicate is **krk/7**, the first argument of which gives the depth of win in moves for a minimax-optimal strategy. The other six arguments specify positions by the file and rank (x and y) chessboard coordinates of, respectively, the White King, White Rook and Black King. The two clauses for the top-level predicate are complete and correct in the sense that they cover all and only those positions in an exhaustive database (described in Section 2.1) which are won at depth 0. This solution, which is discussed in more detail in Section 4, calls the primitive predicates **lt/2** and **diff/3** which define the relations $<$ and symmetric difference for files and ranks. Apart from these background predicates, two machine-invented predicates which are here labelled **wrfile_threat/2** and **wrrank_safe/3** complete the definition.

In the KRK endgame the maximum depth of win for BTM positions is 16 moves. Currently, we have Prolog programs learned by the Inductive Logic Programming system GCWS (described in Section 3) which are complete in the above sense for all depths of win, i.e. from 0 to 16. To our knowledge this is the first time an optimal strategy for a complete endgame has been learned automatically in this way from example positions and low-level background knowledge predicates. In this paper we present results for complete and correct definitions in Prolog at depths of win of 0 and 1 moves. We are continuing to work on correction by specialisation of definitions at the remaining depth levels, and expect to incorporate these results in a subsequent version of this paper. For the present however we will describe only the results from learning the definitions of won at depths 0 and 1. The outline of the paper is as follows. Section 2 contains descriptions of the KRK endgame database used. In Section 3 the learning approach is discussed. The learned definitions are covered in Section 4 and their relation to other work in Section 5.

2 Materials

In previous work chess endgame databases have been used as sources of training and testing examples for machine learning experiments [Qui83, Sha87, Mug87, MBHMM89, Bai91]. The current work employs an exhaustive database which is a complete tabulation for the KRK endgame. The entries of this database contain optimal depth-to-win values for all positions. These entries constititute examples of position classes at each depth of an optimal strategy. Our training and testing examples were randomly sampled from this database.

2.1 Retrograde analysis of endgame databases

The retrograde analysis method for generating chess endgame databases as described in [Tho86] employs reduction of the space of positions by removing from consideration those positions equivalent to a canonical set by symmetry. Consequently, any

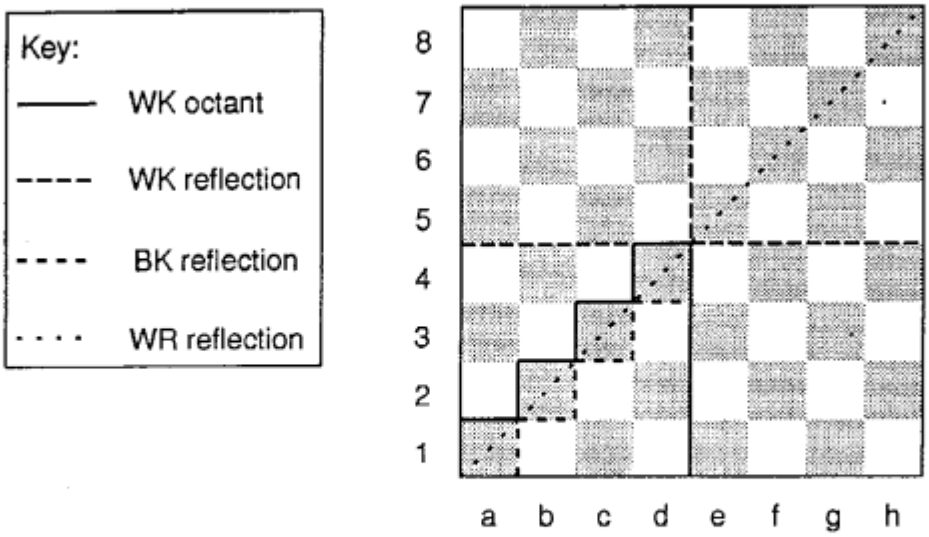


Figure 1: Canonical positions diagram
The axes of reflection and notes in the text indicate how the pieces of any non-canonical position may be translated to give its canonical equivalent.

legal position which could be encountered for example in over-the-board play must be translated to its canonical equivalent before its database value may be retrieved. The exact symmetries which may be exploited vary according to the pieces in the endgame. In the KRK database used to provide examples for the current work three types of symmetrical translation were applied. These are diagrammed in Figure 1 and their meaning is as follows:

WK octant these ten squares are the canonical locations for the White King (WK);

WK reflection reflection about the axes indicated place the WK in a canonical location;

BK reflection with the WK on squares a1, b2, c3 or d4 and the Black King (BK) above the diagonal a1 to h8, the BK can be reflected about this axis to place it below the diagonal;

WR reflection with both WK and BK on the diagonal and the White Rook (WR) above the diagonal, the WR can be reflected about this axis to place it below the diagonal.

The database contains information only on black-to-move (BTM) positions. However this is sufficient to allow optimal play using only a legal move generator which is operated with 2-ply (1 move) lookahead. Since every won position is tagged in the database with its minimax-optimal depth-of-win value, and the learned

Depth	Number	Depth	Number
0	27	9	1712
1	78	10	1985
2	246	11	2854
3	81	12	3597
4	198	13	4194
5	471	14	4553
6	592	15	2166
7	683	16	390
8	1433	Total	25260

Figure 2: Depth of win, optimal play
Depth is depth-of-win for white in moves with black-to-move; Number is number of positions. There are a total of 25260 wins. Together with 2796 draws this gives a total of 28056 positions used for positive and negative example sets.

definitions contain the same values, this method holds also to allow the output of the learning-from-examples method to play optimally.

2.2 BTM WFW positions in the KRK database

Following the removal of redundancy due to symmetries the total space of legal canonical positions in the KRK endgame reduces from a potential 262144 to 28056. In the BTM database used in our experiments the number of positions won-for-white was 25260. Each of these positions is tagged with its depth-of-win value. The number of positions in each depth-of-win class is tabulated in Figure 2.

3 Method

In the present work the goal was the induction of complete, correct and concise theories in the KRK domain. Example positions for Black-to-move (BTM) and win at depth D, where D is a number of moves, were extracted from an exhaustive database computed by the standard retrograde analysis method as discussed above. In this method, each entry in the database contains a position labelled by its minimax-optimal game-theoretic value, which in this case is its depth D. Such a database is taken to be a complete and correct definition of the endgame (although this has not yet been proved), containing as it does all legal positions for the given pieces together with their optimal depth-of-win labels. In logical terminology such a database may be thought of as an extensional definition of the endgame. Therefore

Let d be depth of win
Let P_d be positive examples
Let N_d be negative examples
Let B be the standard background
Let N_{d_k} be rand_subset($|P_d|$, N_d)
$i = 0$
Let T_i be GOLEM(P_d, N_{d_k}, B)

REPEAT
  Let $E_i = \{e : e \in N_d, T_i \vdash e\}$
  IF $E_i = \emptyset$ **BREAK**
  $j = i + 1$
  Let T_j be GCWS(T_i, E_i)
  $i = j$

Figure 3: GCWS algorithm schema.

the database is a complete and correct theory of KRK. Alternatively, the database is a relation $< D, P >$ where D is depth-of-win and P is a canonical position. However, despite reductions due to the removal of positions equivalent by symmetry, the database is too large to be called a concise theory. The goal was therefore the induction of a program which:

1. on input of a legal BTM position in KRK outputs the minimal depth-of-win for white;

2. on some measure was more concise than the database representation.

In this paper the experimental tasks were restricted to positions with depth 0 or 1. Also, we did not apply any measure of relative conciseness. A suitable candidate measure could be HP-compression as described in [MSB92].

We adopted an Inductive Logic Programming approach. Generalisation steps were carried out using the Rlgg algorithm as implemented in Muggleton and Feng's GOLEM system. Correction of over-general clauses with respect to a target model was carried out using the Closed-World Specialisation technique [BM91] in a new Prolog implementation by the author. The combined system is called GCWS and is as described in [Bai91] with a new extension which enables the automatic invention and introduction of non-negated exception predicates during the specialisation process. The method can be viewed as implementing a form of automatic hierarchical problem decomposition.

An informal complexity constraint on the number of clauses used in any predicate definition was applied for learning the depth 0 and 1 definitions, based on the hypothesised limit on human short term memory capacity of 7 ± 2 chunks. (A Prolog predicate is defined using one or more Horn clauses, where each clause contains a single positive literal with the same predicate symbol and arity).

To avoid bias of the method by supplying the learning algorithm with a large number of predefined domain-specific background predicates, the background knowledge was restricted to contain only one specifically chess-oriented geometrical predicate, namely the absolute symmetric difference between files and between ranks. The other background predicate available was "strictly-less-than" ($<$) over files and over ranks.

Details of the experimental method are given in Figure 3 in the form of an algorithm schema. At each depth of win, the positive examples are all positions in the database which are won for white at that depth, and the negative examples are selected randomly from the remaining positions in the database.

Each depth of win was treated as a separate learning problem. The results for depth 0 are in Section 4.1 and for depth 1 in Section 4.2.

4 Results

To recap on the Prolog representation used, the target predicate for the concept was **krk/7**. The first argument of this predicate indicates depth of win, in the range 0 to 16. The remaining six arguments give the file and rank coordinates for, respectively, the White King, the White Rook and the Black King. File arguments are in the range a to h while rank arguments are in the range 1 to 8. Note that these values are those used in the standard algebraic chess notation. For instance, the unit clause "krk(0, c, 1, a, 3, a, 1)" is read as "the black-to-move position WK:c1 WR:a3 BK:a1 is won-for-white at depth 0 (i.e. in 0 moves)". Recall that only legal positions are considered.

Background knowledge was restricted to the predicates **diff/3**, symmetric difference, and **lt/2**, strictly less than. Two argument types were used for file and rank arguments, as for the target predicate. A third type was used for absolute difference arguments.

Throughout this section we illustrate each top-level clause in the induced definition in a figure containing diagrams of partial chessboards. The diagrams indicate position classes covered by the clause indicated. Dotted lines and arrows are used to show variations in placing of the attacking White Rook.

4.1 BTM WFW depth 0

The induced Prolog definition for BTM WFW depth 0 is shown in Figure 4. Clause 1 of this definition is illustrated in Figure 5. As in clause 1, when the new predicate

```
krk(0,A,3,B,1,A,1) :- not(nonkrk1(A,B)).      % Clause 1
nonkrk1(A,B) :- diff(A,B,d1).

krk(0,c,A,a,B,a,C) :- krk2(A,B,C).            % Clause 2
krk2(2,A,1) :- lt(2,A).
krk2(A,B,A) :- diff(A,B,d2).
krk2(A,B,A) :- diff(A,B,d3).
krk2(A,B,A) :- diff(A,B,d4).
krk2(A,B,A) :- diff(A,B,d5).
krk2(A,B,A) :- diff(A,B,d6).
krk2(1,8,1).

%  Key (labels for machine-invented predicates) :
%    nonkrk1="wrfile_threat"
%    krk2="wrrank_safe"
```

Figure 4: BTM WFW depth 0

has only a single clause in its definition, it may be replaced with the clause body. In incremental learning this could however be undesirable – the invented predicate if retained might have further clauses added to its definition. For the time being we can interpret clause 1 as follows : "any BTM KRK position with the kings on the same file, the white king on rank 3 and both the rook and black king on rank 1 is WFW depth 0 when the symmetric difference between king and rook files is not 1". The machine-invented predicate **nonkrk1/2** clearly relates to the idea of rook safety when the black king is in check. It might be labelled **wrfile_threat/2**. This concept is apparent from the diagrams in Figure 5. Diagram (i) shows the placings of the White Rook on rank 1, essentially on file a and files e to h. Diagram (ii) illustrates the same pattern shifted one file to the right, with the White Rook on rank 1, files a, b and f to h.

The second clause is in a sense the dual of the first, this time for rank values, with the machine-invented predicate **krk2/3** being labelled **wrrank_safe/3**. This is shown in Figure 6. Diagram (i) illustrates the first clause of **krk2/3**, with the Kings not in opposition. Diagram (ii) covers the remaining clauses of **krk2/3** where the Kings are in opposition. In all cases the White Rook attacks the Black King down file a from the safety of rank 3 or above.

These two clauses cover all 27 positions won at depth 0. Although this is a small number of examples, the complexity of describing the concepts involved is clear from the diagrams of Figures 5 and 6.

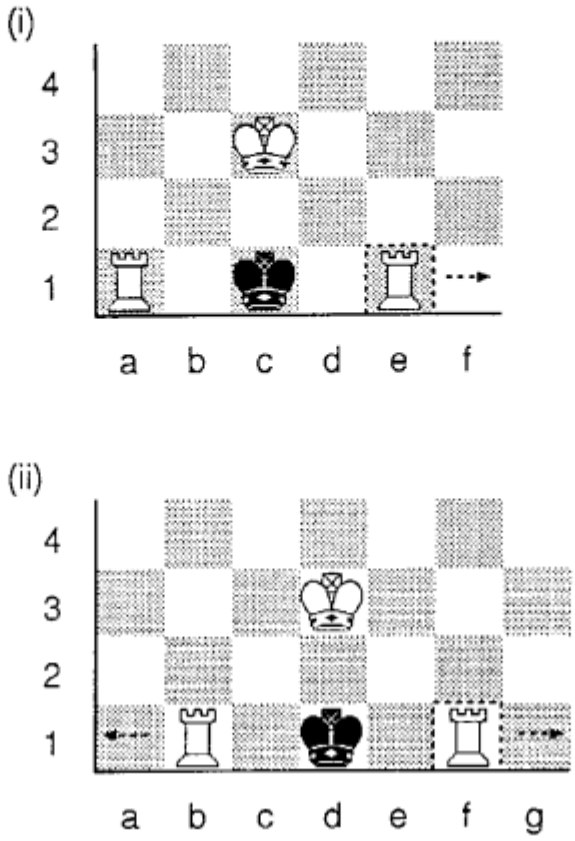


Figure 5: BTM WFW depth 0, clause 1

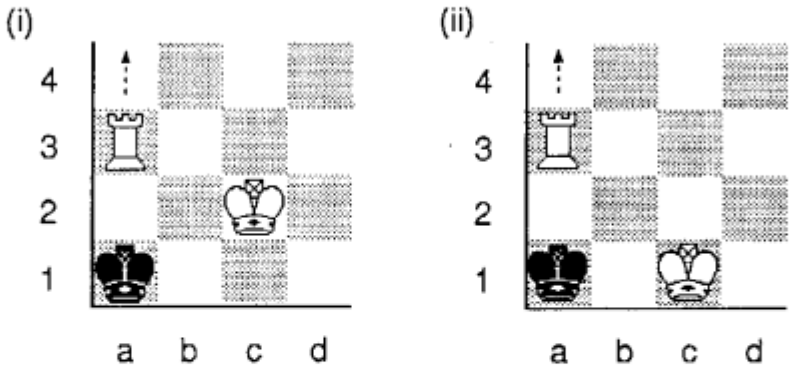


Figure 6: BTM WFW depth 0, clause 2

```
krk(1,A,3,B,C,D,1) :- krk1(A,B,C,D), diff(A,B,d2), diff(A,D,d1).      % Clause 1

krk1(c,a,A,b) :- not(nonkrk12(A)).
krk1(c,e,A,d) :- not(nonkrk12(A)).
krk1(d,b,A,c) :- not(nonkrk12(A)).
krk1(d,f,A,e) :- not(nonkrk12(A)).

nonkrk12(1).
nonkrk12(2).

krk(1,c,2,A,B,a,C) :- krk2(A,B,C).      % Clause 2

krk2(A,4,3) :- not(nonkrk21(A)).
krk2(A,3,2) :- not(nonkrk22(A)).
krk2(A,4,1) :- not(nonkrk22(A)).
krk2(A,5,1) :- not(nonkrk22(A)).
krk2(A,6,1) :- not(nonkrk22(A)).
krk2(A,7,1) :- not(nonkrk22(A)).
krk2(A,8,1) :- not(nonkrk22(A)).

nonkrk21(a).
nonkrk21(b).

nonkrk22(a).

krk(1,c,1,A,3,a,2) :- not(nonkrk3(A)).      % Clause 3

nonkrk3(a).
nonkrk3(b).

%   Key (labels for machine-invented predicates) :
%     krk1="wrrank_safe"
%     krk2="wrfile_safe"
%     nonkrk12="ranklte2"
%     nonkrk21="filelteb"
%     nonkrk22="edgefile"
%     nonkrk3="filelteb"
```

Figure 7: BTM WFW depth 1

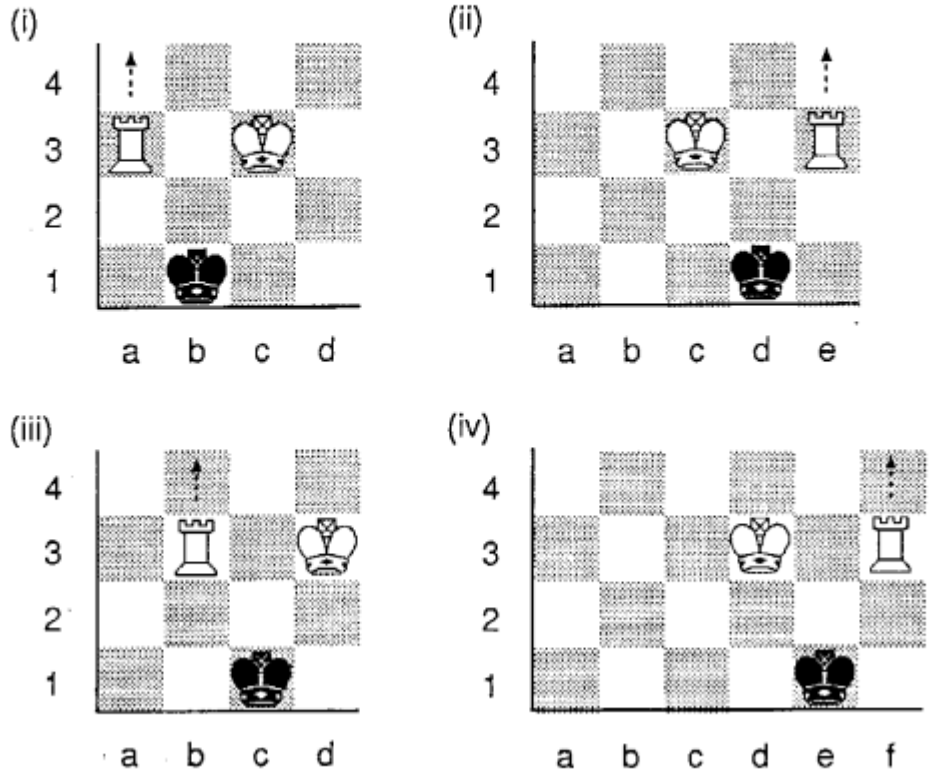


Figure 8: BTM WFW depth 1, clause 1

4.2 BTM WFW depth 1

The induced Prolog definition for BTM WFW depth 1 is show in Figure 7. The top-level predicate **krk/7** has three clauses in this definition. Clause 1 of this definition is illustrated in Figure 8. There are four diagrams in this figure, each of which show essentially the same pattern of attacking relationships between the pieces. Diagrams (i) to (iv) depict the patterns for the four clauses of the definition of the the invented predicate **krk1/4**. This predicate might be labelled **wrrank_safe**, since it is principally a condition on the White Rook's rank safety. This is because the key attack relationships are defined by the top-level of clause 1. To see that the patterns do indeed cover checkmate at depth 1, refer to diagram (i) in Figure 8. The Black King is forced to move to c1, which is followed by the White Rook moving to a1. Black is in check with no further moves. Recall that this checkmate position is covered by the pattern of diagram (i) in Figure 5.

The second clause of the BTM WFW depth 1 definition has only the machine-invented predicate **krk2/3** in the clause body. The clause has the White King fixed on square c2, and a condition is required on the relation of the White Rook

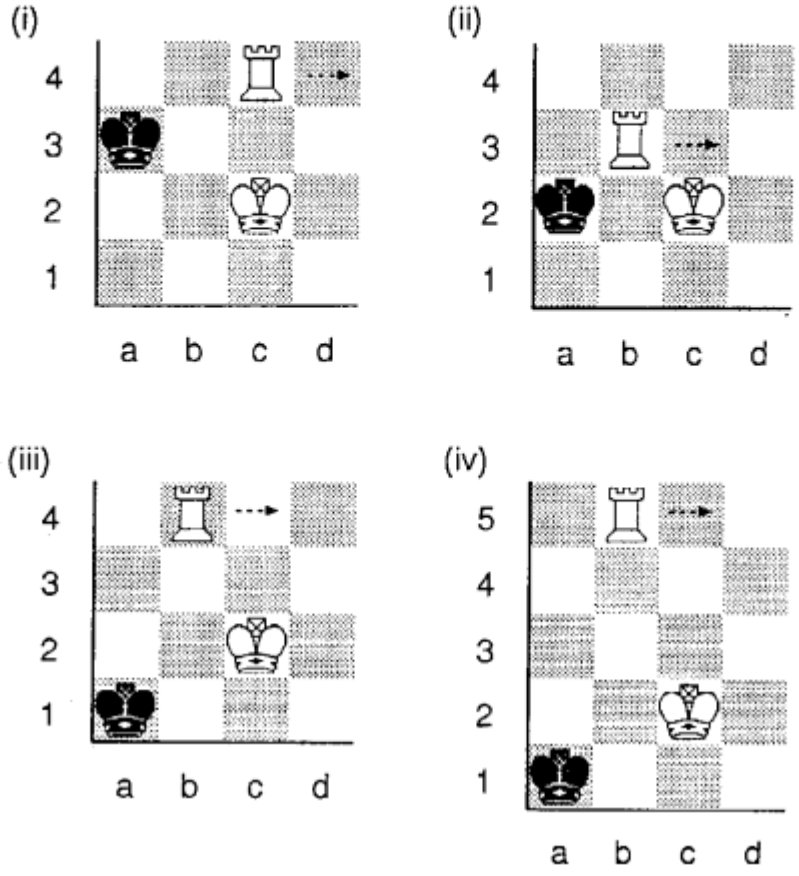


Figure 9: BTM WFW depth 1, clause 2

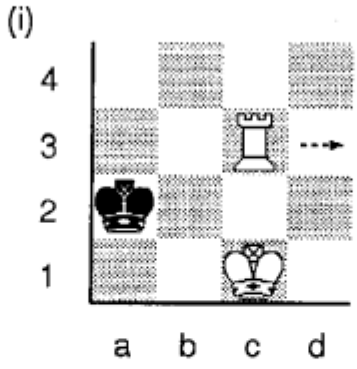


Figure 10: BTM WFW depth 1, clause 3

to the Black King. We might label this condition **wrfile_safe**, since the White Rook is restricting the movement of the Black King without directly attacking it or being attacked by it. The key patterns are shown in Figure 9. In diagram (i) the locations of the Kings are fixed with the White Rook free to occupy rank 4 on any file apart from a or b. This pattern corresponds to the first clause in the definition of **krk2/3**. Diagram (ii) presents an interesting configuration with the Kings in opposition. This pattern corresponds to the second clause of **krk2/3**, which covers one position where the White Rook is attacked by the Black King but defended by the White King. The remaining clauses in the definition cover positions falling into pattern classes closely related to that of diagram (i). For instance diagrams (iii) and (iv) corresponds to clauses 3 and 4, with the Black King forced to move into opposition on file a followed by the the White Rook moving to give checkmate on this file.

The third clause in the definition of the predicate **krk/7** covers positions fitting the pattern of Figure 10. Here the positions of both Kings are fixed, and the Black King is forced to move into square a1. From its safe postion on rank 3 at file c or greater, the White Rook moves to a3 giving checkmate. The reader may care to verify that the patterns shown account for all 78 of the canonical BTM positions which are WFW at depth 1.

5 Discussion

Work on the inductive synthesis of knowledge employing the "easy inverse trick" [Mic86] pushed the nascent technology of decision-tree induction to its limits in the late 70s and early 80s [Qui83]. These initial results in chess endgame domains led to a variety of successful applications. The landmark KARDIO system used a deep model of the heart to synthesise shallow rules for ECG interpretation [BML89]. Among the rules which came out of this study were some previously undiscovered in over 200 years of cardiology. This route was also taken in an application to satellite fault diagnosis [Pea88]. Most recently, an ILP approach in the same domain allowed the learning of significant temporal relations not expressed in the earlier solution [Fen91].

The rôle of Machine Learning in previous work has focused on database compression. Even with decision tree induction this compression can be significant, as in the KARDIO work. Typically, however, in the chess endgame applications to date the bottleneck for decision-tree induction has been selection of an adequate set of attributes. For example, in experiments on the KPa7KR domain [SM86] most of the effort was expended on hand-crafting the attributes which capture the necessary relational features of the won/not won predicate. The novelty of the present approach lies in the application of relational learning using Rlgg as implemented in GOLEM coupled with specialisation techniques based on predicate invention. This follows from earlier results with a similar approach in the simpler KRK illegality domain [Bai91].

The predicate invention methods by which induced clauses are specialised by introducing literals into the clause body are a special case of predicate invention within Muggleton's "refinement lattice" framework.

6 Concluding remarks

We have presented a results from learning optimal strategies in the KRK endgame. Optimality is achieved through the use of examples extracted from an exhaustive database for the endgame. A complete theory for black-to-move, won-for-white positions at all depths (0 to 16) has been learned automatically from ground instances of positions and only low-level background knowledge. The definitions for win at depths 0 and 1 have been fully specialised and were presented as a complete and correct definition of the target concept at the levels. They have been tested fully on the exhaustive example set.

Acknowledgements. This work was supported by IED project 4/1/1320 on Temporal Databases and Planning. I thank Dr Stephen Muggleton for proposing the KRK project, and Prof Donald Michie, Prof Ivan Bratko and the members of the Turing Institute ILP Group for their comments and suggestions regarding this work.

References

[Bai91] M. E. Bain. Experiments in non-monotonic learning. In L. Birnbaum and G. Collins, editors, *Proceedings of the Eighth International Workshop on Machine Learning*, pages 380–384, San Mateo, CA, 1991. Morgan Kaufmann.

[BM91] M. Bain and S. H. Muggleton. Non-monotonic learning. In J. E. Hayes, D. Michie, and E. Tyugu, editors, *Machine Intelligence 12*, pages 105–119. Oxford University Press, Oxford, 1991.

[BML89] I. Bratko, I. Mozetic, and N. Lavrac. *KARDIO: a study in deep and qualitative knowledge for expert systems.* MIT Press, Cambridge, 1989.

[Fen91] C. Feng. Inducing temporal fault diagnostic rules from a qualitative model. Machine Learning Research Seminar Paper 47, Turing Institute, Glasgow, 1991.

[MBHMM89] S. H. Muggleton, M. E. Bain, J. Hayes-Michie, and D. Michie. An experimental comparison of human and machine learning formalisms. In A. Segre, editor, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 113–118, Los Altos, CA, 1989. Kaufmann.

[Mic86] D. Michie. Towards a knowledge accelerator. In D. F. Beal, editor, *Advances in Computer Chess*, volume 4, pages 1–8. Pergamon Press, Oxford, 1986.

[MSB92] S. Muggleton, A. Srinivasan, and M. Bain. Compression, Significance and Accuracy. *submitted to Machine Learning*, 1992.

[Mug87] S. H. Muggleton. An oracle-based approach to constructive induction. In *IJCAI-87*, pages 287–292, Los Altos, CA, 1987. Kaufmann.

[Pea88] D. Pearce. The induction of fault diagnosis systems from qualitative models. In *AAAI-88: Proceedings of the 7th National Conference on Artificial Intelligence*, pages 353–357, San Mateo, CA, 1988. Morgan Kaufmann.

[Qui83] J. R. Quinlan. Learning efficient classification procedures and their application to chess end games. In R. Michalski, J. Carbonnel, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 464–482. Tioga, Palo Alto, CA, 1983.

[Sha87] A. D. Shapiro. *Structured Induction in Expert Systems*. Turing Institute Press with Addison Wesley, Wokingham, UK, 1987.

[SM86] A. D. Shapiro and D. Michie. A self-commenting facility for inductively synthesised endgame expertise. In D.F. Beal, editor, *Advances in Computer Chess*, volume 4, pages 147–165. Pergammon, Oxford, 1986.

[Tho86] K. Thompson. Retrograde Analysis of Certain Endgames. *ICCA Journal*, 8(3):131–139, 1986.

Drug design by machine learning : The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase

Ross D King(1), Stephen Muggleton(2),
Richard A Lewis(3) * & Michael J E Sternberg(3)@

(1) Department of Statistics
Strathclyde University
Glasgow , G1 1XH, U.K.

(2) Turing Institute
George House, 36 North Hanover Street
Glasgow, G1 2AD, UK

(3) Biomolecular Modelling Laboratory
Imperial Cancer Research Fund
44 Lincoln's Inn Fields, London WC2A 3PX, UK
(Tel +44 71-269-3565
FAX +44 71-430-2666)

(*) Present address: Rhone-Poulenc Rorer, Dagenham Research Centre
Rainham Road South, Dagenham Essex RU10 7XS, UK

@ to whom correspondence should be addressed

Key words: artificial intelligence, enzyme activity, protein modelling, active sites

Submitted to PNAS 12 Dec 1991 & revised form 15 April 1992

Abstract

The machine learning program Golem from the field of inductive logic programming was applied to the drug design problem of modelling structure - activity relationships. The training data for the program was 44 trimethoprim analogues and their observed inhibition of E. coli dihydrofolate reductase. A further 11 compounds were used as unseen test data. Golem obtained rules that were statistically more accurate on the training data and also better on the test data than a Hansch linear regression model. Importantly machine learning yields understandable rules that characterised the chemistry of favoured inhibitors in terms of polarity, flexibility and hydrogen-bonding character. These rules are in agreement with the stereochemistry of the interaction of trimethoprim with dihydrofolate reductase observed crystallographically. We conclude that machine learning offers a new approach which complements other methods and could provide vital insights to direct the time-consuming process of the design of potent pharmaceutical agents from a lead compound.

1 - Introduction

The design of a potent pharmaceutical agent from a lead compound is often based on an understanding of the quantitative structure-activity relationship (QSAR) in a related series of ligands (e.g.(1, 2, 3)). Generally, the structure of the receptor is unknown and systematic variation of the chemical properties of the ligands is used to infer the requirements imposed by the receptor binding site. Any method that can model accurately QSAR from a small sample of the total number of ligands will aid greatly in the rapid discovery of a compound with optimal activity and will be of clinical importance.

A standard method for modelling a QSAR was proposed by Hansch (4, 5, 6), in which the physicochemical properties of a series of similar compounds are linked by an empirical equation to their biological activity. However the equation gives little insight into the structure of the binding site. Recently, neural network models have been applied to QSAR with some success (7) but the design of the network is highly subjective and the numerical results are difficult to interpret.

Pattern recognition methods have been used in QSAR. Principal component analysis (8) is widely used to identify the chemical properties that contribute most to the activity of a compound but does not give insight into the factors controlling the binding of a ligand to its receptor. An alternative pattern recognition method with potential advantages for QSAR is machine learning. Machine learning methods are non-parametric and non-linear, and work best when using human understandable symbols to represent a problem. Thus in drug design the concepts used, such as size, polarity and flexibility, relate naturally to stereochemistry. The use of such symbols has two potential advantages over the numerical representation First, the problem can be set up and changed more easily because the designer can work in comprehensible terms. Background knowledge, in

particular the stereochemistry of the compounds, can be directly added, while in a statistical method it would typically be represented by some form of prior probabilities, or in a neural net by connection weights and topology. Second and more importantly, the production of humanly comprehensible rules from a machine learning system allows the rules to be checked for consistency with existing knowledge, and opens the possibility that the rules may provide fresh insight into the problem.

In this paper we apply the machine learning program Golem (9) from the newly developed field of ILP (Inductive Logic Programming) (10) to QSAR. In the development of methodologies, it is advantageous to consider systems for which atomic structural information of the drug-receptor complex is available. An ideal system is the complex formed between analogues of the drug trimethoprim and the enzyme dihydrofolate reductase (DHFR) from E. coli which has been studied crystallographically (11, 12). Thus one can compare the predicted QSAR models with the X-ray stereochemistry of interaction. These compounds have been studied by Hansch et al. (6) and so provide an ideal system to compare the performance of machine learning with the Hansch method.

2 - Methods

Data

The study was performed with a training set of 44 trimethoprim analogues (Table 1a) from Hansch et al.(6) and a testing set of 11 further cogeners (Table 1b) from Roth and coworkers (13, 14) (Table 1). Biological activities is measured as $\log(1/K_i)$ where K_i is the equilibrium constant for the association of the drug to DHFR. There are 25 different substituent groups in the set and three alternate substitution positions (3, 4 and 5) on the phenyl ring (Figure 1a). This yields a set

of 8125 possible substituents when the symmetry of the 3 and 5 positions are included.

Golem

Golem (9) is a program for machine learning by Inductive Logic Programming (ILP). The ILP methodology (10) was chosen because it is designed specifically to learn relationships between objects (e.g. molecular structures). In ILP, each language is a subset of first-order predicate calculus which is expressive enough to describe most mathematical concepts and, having a strong link with natural language, leads to ease of comprehension. Golem is written in the programming language C but implements predicate logic in the language Prolog.

Golem takes as input: positive examples, negative examples, and background knowledge described as Prolog facts. It produces as output: Prolog rules which are generalisations of the examples. In Golem, the generative phase resembles the standard scientific method. Observations are collected from the outside world (the activities of trimethoprim analogues). These are then combined by an ILP program with background knowledge (the stereochemistry of the compounds) to form inductive hypothesis (rules relating the structure of an analogue with its activity). These rules are then experimentally tested on additional data. If experimentation leads to high confidence in the validity of the hypotheses, the rules are added to the background knowledge. The Prolog rules are constructed so that together with the background knowledge, they explain all the positive input examples and none of the negative examples. The method of generalisation is based on the logical idea of Relative Least General Generalisation.

The basic algorithm used in Golem is as follows. Firstly it takes a random sample of pairs of positive examples. In this application, this will be a set of pairs of

compared drugs chosen randomly from the set of all examples represented (see below). For each of these pairs Golem computes the set of all properties which are common to both examples. These properties are then made into a rule which is true of both the examples in the pair under consideration. Having built such a rule for all chosen pairs of examples, Golem takes each rule and computes the number of examples which that rules could be used to predict. Clearly these rules might predict some examples which are false. Golem therefore chooses the rule which predicts the most true examples while predicting less than a predefined threshold of false examples. Having found the rule for the best pair, Golem then takes a further sample of as yet unpredicted examples and forms rules which express the common properties of this pair together with each of the individual residues in the sample. Again the rule which gives best predictions on the training set is chosen. The process of sampling and rule building is continued until no improvement in prediction is produced.

A method in Golem is also needed to avoid over-fitting the data and to deal with noise. These problems are dealt with by using the Minimal Description Length as implemented in the compression model of Muggleton (15) Each hypothesis produced by Golem in the experimentation stage was checked by compression for significance. Thus rules which have too many variables for the the number of examples they cover are rejected.

Application of Golem to the QSAR of the trimethoprim series

To apply Golem to the QSAR of the trimethoprim series, the data has to be converted into a form suitable for Golem. QSAR is generally a regression problem, in which a real number is predicted from the description of a compound. However,

Golem is designed to carry-out classification (discrimination) tasks in which a small number of discrete classes are predicted. This difficulty is by-passed by considering pairs of drugs and comparing their activities (pairs where the activity are equal or within the margin of experimental error are discarded). The output is a set of rules which decides which of a pair of drugs has higher activity. Paired comparisons are then converted to a ranking by the method of David (16).

The input to Golem are three types of fact: positive, negative and background. The positive examples are the paired examples of greater activity, e.g.
great(d20, d15).
which states that drug no. 20 has higher activity than drug no. 15. The negative examples are the paired examples of lower activity.

The background facts are the chemical structure of the drugs and the properties of the substituents. Chemical structure is represented in the form:
struc(d35, NO_2, $NHCOCH_3$, H).
which states that drug no. 35 has: NO_2 substituted at position 3 , $NHCOCH_3$ substituted at position 4, and no substitution at position 5. In addition, if either position 3 or 5 has no substitution, as in drug no 35, the position with no substitution is assumed to be position 5 (this is used in ref (6)).

Chemical properties were then assigned heuristically to substituents (Table 2). The properties, chosen to make the approach generally applicable to drug-design problems, are: polarity, size, flexibility, hydrogen-bond donor, hydrogen-bond acceptor, π donor, π acceptor, polarisability and σ effect. Each of the 24 non-hydrogen substituents was given an integer value for each of these properties. This was represented using different predicates for each property and value, e.g:
polar(Br, polar3).
states that Br has polarity of value 3.

Information was also given about the relative values of these properties for the substituent groups e.g:

polar(OCF_3, polar4).

polar(CH_2OH, polar2).

great_polar(polar4, polar2).

together state that OCF_3 is more polar than CH_2OH (the "great_polar" fact is background knowledge about arithmetic ordering).

Finally information was given about the relative values of the properties compared to fixed values, e.g.

great0_polar(polar1).

states that a polarity of 1 is greater than a polarity of 0.

The input to Golem was 871 positive facts, 871 negative facts, 2976 facts in the background information. The run time was about 30 cpu minutes on a SUN SparcStation 1 per rule.

3 - Results

Golem derived nine rules that predict the relative activities of two drugs. Table 3 lists the machine learned rules in Prolog syntax together with an English interpretation. The coverage indicates the number of pairs of relationship that are correctly and incorrectly represented. Each rules relates the relative activities of two drugs (A and B) and identifies the chemical properties of substituents that yield a drug of higher activity.

Table 1 gives the performance on training data of 44 compounds for both machine learning and application of the Hansch equation expressed as a rank value. The prediction from machine learning gave a rank correlation with the observed order of 0.916 (using the Spearman method (17)) (Figure 2a). As a benchmark, the

Hansch equation had a rank correlation of 0.794 (Figure 2b). The significance of the difference in these ranking was evaluated by Fisher's z transformation (17). The value $z = 2.18$ is significant at the 5% level and almost significant at the 1% level ($p=0.985$).

A better test of a prediction method is its performance on data not used in developing the algorithm. The structure and activity of eleven new trimethoprim analogues not used in the original paper (6) on which the Hansch equation was derived was used as a test set for the two approaches. A ranking of the new 11 drugs relative to all 55 drugs was obtained by: (1) forming all paired comparisons involving the new 11 drugs, (2) these are added to the predicted results of the paired comparisons of the original 44 drugs, (3) a ranking is then produced from all the paired comparisons. From this ranking of 55 drugs, a rank order for the 11 new drugs was extracted (Table 1b) and this was compared by a rank correlation coefficient to the observed order. The rank correlation for the 11 new drugs by machine learning was 0.457 compared to 0.415 for the Hansch method (Figure 2). The Fisher z-value is 0.10 which is not significant ($p= 0.540$) and reflects the similar rank correlations obtained on a small test set. Thus on the test set, the machine learning is as accurate as the regression approach of Hansch. Both methods predict well that the tests drugs have high activity (Figure 2).

A further test of the Golem approach was a cross-validation study in which 44 of the 55 drugs were chosen at random as a training set with the remaining 11 as the test data. The resultant Spearman rank correlation coefficients (Table 4) for the training sets are similar to those for the main trial. However in the cross-validation work, the tesing drugs were chosen over the entire range of activities and the correlation coefficients were higher than that obtained for the 11 new drugs taken in the main trial which were at the top end of activity.

Discussion and Conclusion

The X-ray crystallographic structures of the trimethoprim / E. coli DHFR complex (11) and of the ternary complex (12) with NADPH have been solved (see Figure 1b). In the ternary complex, the phenyl ring of trimethoprim is sandwiched in a hydrophobic cleft between Phe 31, Leu 28 and Met 20 on one side and Leu 54, Ile 54, Ser 49 and with the NADPH cofactor on the other. The aromatic ring is thus effectively buried whilst the environments of the 3, 4 and 5 substituents vary. The 4 (i.e. para) position is the most exposed to solvent whilst the meta positions (i.e. the 3 and 5 substituents) are restricted in size by the surrounding protein and cofactor atoms.

A main aim in using machine learning was to obtain rules that could provide insight into stereochemistry of drug / DHFR interactions. We examined the features that favour the better drugs (i.e. the properties of drug A in the rules). For 3 and/or 5 positions a favourable substituent D is defined as:

h_donor(D,h_don0),

pi_donor(D,pi_don1),

flex(D,G), less4_flex(G),

size(D,size2),

polar(D,V), great0_polar(V),

polarisable(D,polar1).

The properties are therefore: not a hydrogen-bond donor; a π-donor of 1; flexibility < 4; a size of 2; a polarity greater than zero; and a polarisability of 1. Only the methoxy (OCH_3) substituent satisfies these conditions. These principles are in keeping with the location of both meta sites (i.e. both 3 and 5 positions) in the crystal structures. Both meta sites are partially buried in a hydrophobic environment and hence have a restriction on size and flexibility. The absence of solvent at these sites might explain the requirement that the group should not be a

hydrogen bond donor. Substituents that are π-donors will force this group to lie in the plane of the aromatic ring and this has been suggested as a requirement for a favourable drug (14). Finally, because both meta positions have similar chemical locations in DHFR, one cannot decide whether the rules in Table 3 for a 3 position on the chemical compound (Figure 1a) relate exclusively to the upper meta position or exclusively to the lower meta position or to both locations in the three-dimensional location (Figure 1b).

The only positive feature for the 4-position is that it should have a polarity of 2. This property is consistent with a site that is exposed to solvent and should be polar. Matthews et al (11) proposed that the oxygen of the methoxy group might form a hydrogen bond with a neighbouring water molecule. In addition the rules suggest that each of the 3, 4 and 5 positions should not be hydrogen. This is in keeping with the suggestion (13) that an important role of the 4 position is to force the meta substituents away from the 4 position towards the 2 and 6 positions.

Golem could have used representations other than chemical properties (Table 2). The original numerical attributes used in Hansch et al (6) were tried as the background knowledge to Golem and this also produced rank correlations in the training data and test data better than the Hansch equation. However, the rules produced did not provide the insight we were seeking. An alternative approach would be to represent explicitly chemical bonds and their possible movements but this method is likely to be computationally expensive.

For drug design, we have shown that machine learning can yield rules that model a QSAR of a series of DHFR inhibitors better than one of the standard methods widely used. In addition one automatically derives a stereochemical description of the drug / receptor interaction For DHFR we know that these description are in broad agreement with crystallographic results. But for many systems of interest

there will be no structure for the receptor and these rules will thus provide valuable insight that can guide subsequent drug design. The machine learning approach will have to be tried on other structure / activity series. If successful we suggest that machine learning will be a useful tool in addition to the other approaches in QSAR to speed the process of drug discovery.

More generally, ILP techniques are particularly appropriate for biological problems due to their ability to describe complex relational and structural features presented by this domain. In another recent study, Golem (18) has produced predictions of the secondary structure of α/α proteins of 80% accuracy. Together with the results of this paper, we consider that this demonstrates the wide-ranging potential of ILP in the domain of modelling biological information.

Acknowledgements

RDK is supported by the Esprit "Statlog" project; RAL by a fellowship from the Royal Commission of 1851; SM by for an SERC research fellowship; and MJES by the Imperial Cancer Research Fund. We thank Drs B Roth and C Beddell for helpful comments.

References

1. Goodford, P. J. (1984) *J. Med. Chem.* **27**, 557-564 .

2. Dean, P. M., (1987) *Molecular foundations of drug-receptor interactions* (Cambridge University Press, Cambridge).

3. Marshall, G. R. & Cramer, R. D. (1988) *TIPS* **9**, 285-289 .

4. Hansch, C., Maloney, P. P., Fujita, T. & Muir, R. M. (1962) *Nature* **194**, 178-180 .

5. Hansch, C. (1969) *Acc. Chem. Res.* **2**, 232-239 .

6. Hansch, C., Li, R.-I., Blaney, J. M. & Langridge, R. (1982) *J. Med. Chem.* **25**, 777-784 .

7. Andrea, T. A. & Kalayeh, H. (1991) *J. Med. Chem.* **34**, 2824-2836 .

8. Wold, S., Dunn, W. J. & Hellberg, S., (1984) in *Drug Design: Fact or Fantasy?* (G. Jolles, K. R. H. Wooldridge, Eds) . (Academic Press, London), pp. p95-115.

9. Muggleton, S. & Feng, C., (1990) in *Proceedings of the first conference on algorithmic learning theory* (S. Arikawa, S. Goto, S. Ohsuga, T. Yokomori, Eds). (Japanese Society for Artificial Intelligence, Tokyo), pp. 368-381.

10. Muggleton, S. (1991) *New Generation Computing* **8**, 295-318 .

.11. Matthews, D. A., Bolin, J. T., Burridge, J. M., Filman, D. J., Volz, K. W., Kaufman, B. T., Beddell, C. R., Champness, J. N., Stammers, D. K. & Kraut, J. (1985) *J. Biol. Chem.* **260**, 381-391 .

12. Champness, J. N., Stammers, D. K. & Beddell, C. R. (1986) *FEBS Letters* **199**, 61-67 .

13. Roth, B., Aig, E., Rauckman, B. S., Strelitz, J. Z., Phillips, A. P., Ferone, R., Bushby, S. R. M. & Sigel, C. W. (1981) *J. Med. Chem.* **24**, 933-941 .

14. Roth, B., Rauckman, B. S., Ferone, R., Baccanari, D. P., Champness, J. N. & Hyde, R. M. (1987) *J. Med. Chem.* **30**, 348-356 .

15. Muggleton, S., Srinivasan, A. & Bain, M. (1992) in *Proceedings of 9th International Conference on Machine Learning* pub Morgan-Kaufman, San Diego

16. David, H. A. (1987) *Biometrika* **74**, 432-436 .

17. Kendall, M. & Stuart, A., (1977) *The Advanced Theory of Statistics* (Griffen and Company, London,).

18. Muggleton, S., King, R. D. & Sternberg, M. J. E. (1992) *Protein Engineering, submitted.*

Table 1 Predicted and observed activity of trimethoprim analogues

X gives the substituents. The observed value of the affinity is expressed as log $1/K_{i\ app}$. The first 44 drugs (Table 1a) were used in the training set and the observed rank ranges from 1 to 44. The final 11 drugs (Table 1b) are the testing set and the first number is the rank in the 55 drugs and the second the rank for the 11. The rank corresponding rank value obtained by machine learning (Golem) and by the application of the Hansch equation are in the subsequent columns. These two algorithms calculate a paired comparisons which is then converted to a ranking by the algorithm of David (16). A win is when a drug is predicted to have higher activity in a pair and a loss is lower activity in a pair. The David number of drug X is calculated to be the total number of (a) wins of drugs defeated by X minus losses of drugs to whom X lost, plus (b) X's wins minus X's loses. The David number is used to rank the drugs; low numbers have low activity, high numbers have high activity. If the drugs have the same David number they have tied rank.

Table 2 Chemical properties of substituents

PL - polarity which indicates the amount of residual charge on the α- and β- atoms of the substituent. SZ- size is a measure of the extended volume of the group. FLX- flexibility is assigned from the number of rotatable bonds. H-D and H-A indicate the presence and strength of hydrogen-bonding acceptors and donors. P-D and P-A indicate the presence and strength of π– acceptors and π-donors. POL indicates polarisability of the molecular orbitals and SIG is its σ-property. These properties were assigned so that they are internally consistent without any relative weight between them.

Table 3 Rules for QSAR derived by machine learning

The rules are first given as Prolog clauses in which ":-" is a definition and a "," is logical "and". Then an exact interpretation is given. The rules have been classified into those primarily relating to substituent 3 (rule 3.1 to 3.6); to substituent 4 (4.1 and 4.2) and one that relates to both positions (3&4.1).

Table 4: Results of cross-validation of machine learning.

The Spearman rank correlation coefficient betweeen the order obtain by Golem and the true rank is given for the training set of 44 drugs and the testing set of 11 drugs. The drugs in the testing set are defined by the order they are given in Tables 1a (1-44) and 1b(45-55).

Figure 1

(a) the structure of trimethoprim analogues

(b) a cartoon of the interaction of trimethoprim with DHFR from X-ray structures (11, 12). Faint stippling indicates that the residue lies below the plane of the phenyl ring, darker stippling that the atoms are above.

Figure 2

Scattergram of the observed rank versus that predicted by Golem (2a) and by Hansch (2b).

Table 1a

X	log $1/K_{i\ app}$	Observed rank	rank by Golem	rank by Hansch
3,5-$(OH)_2$	3.04	1	17	2
4-$O(CH2)_6CH_3$	5.60	2	4.5	4
4-$O(CH_2)_5CH_3$	6.07	3	4.5	10
H	6.18	4	1	6.5
4-NO_2	6.20	5	7.5	20.5
3-F	6.23	6	6	6.5
3-$O(CH_2)_7CH_3$	6.25	7	15	6.5
3-CH_2OH	6.28	8	2	16
4-NH_2	6.30	9	7.5	3
3,5-$(CH_2OH)_2$	6.31	10	3	23
4-F	6.35	11	9.5	6.5
3-$O(CH_2)_6CH_3$	6.39	12	16	11
4-$HCH_2CH_2OCH_3$	6.40	13	20	20.5
4-Cl	6.45	14	12	17.5
3,4-$(OH)_2$	6.46	15	18	1
3-OH	6.47	16	13	9
4-CH_3	6.48	17	9.5	17.5
3-$OCH_2CH_2OCH_3$	6.53	18	21	34
3-$CH_2O(CH_2)_3CH_3$	6.55	19	24	35.5
3-OCH_2CONH_2	6.57	20.5	14	13
4-OCF_3	6.57	20.5	19	22
3-CH_2OCH_3	6.59	22	28	29.5
3-Cl	6.65	23	30.5	29.5
3-CH_3	6.70	24	30.5	27.5
4-$N(CH_3)_2$	6.78	25	22.5	27.5
4-Br	6.82	27	11	24
4-OCH_3	6.82	27	22.5	26
3-$O(CH_2)_3CH_3$	6.82	27	29	32
3-$O(CH_2)_5CH_3$	6.86	29	26	14
4-$O(CH_2)_3CH_3$	6.89	30.5	27	15
4-$NHCOCH_3$	6.89	30.5	25	12
3-OSO_2CH_3	6.92	32	33	25
3-OCH_3	6.93	33	36	38
3-Br	6.96	34	37	35.5
3-NO_2, 4-$NHCOCH_3$	6.97	35	34	37
3-$OCH_2C_6H_5$	6.99	36	35	31
3-CF_3	7.02	37	32	19
3,4-$(OCH_2CH_2OCH_3)_2$	7.22	38	39	40
3-I	7.23	39	38	33
3-CF_3, 4-OCH_3	7.69	40	41.5	39
3,4-$(OCH_3)_2$	7.72	41	41.5	41
3,5-$(OCH_3)_2$, 4-$O(CH_2)_2OCH_3$	8.35	42	43	43
3,5-$(OCH_3)_2$	8.38	43	40	42
3,4,5-$(OCH_3)_3$	8.87	44	44	44

Table1b

X	log $1/K_{i\ app}$	Observed rank		rank by Golem		rank by Hansch	
3,5-$(CH_3)_2$, 4-OCH_3	7.56	40	1	52.5	9	45.5	4.5
3-Cl, 4-NH_2, 5-CH_3	7.74	43	2	40	2	44	3
3,5-$(CH_3)_2$, 4-OH	7.87	44.5	3.5	40	2	41	1
3,5-Cl_2, 4-NH_2	7.87	44.5	3.5	40	2	45.5	4.5
3,5-Br_2, 4-NH_2	8.42	48	5	44	4	53	10
3,5-$(OCH_3)_2$, 4-$OCH_2C_6H_5$	8.57	49	6	52.5	9	52	9
3,5-$(OCH_3)_2$, 4-CH_3	8.82	50.5	7.5	47.5	5.5	54.5	11
3,5-$(OCH_3)_2$, 4-$O(CH_2)_7CH_3$	8.82	50.5	7.5	52.5	9	43	2
3,5-$(OCH_3)_2$, 4-$O(CH_2)_5CH_3$	8.85	52	9	52.5	9	49	7
3,5-I_2, 4-OCH_3	8.87	54.5	10.5	52.5	9	50	8
3,5-I_2,4-OH	8.87	54.5	10.5	47.5	5.5	47	6

Table 2

Group	PL	SZ	FLX	H-D	H-A	P-D	P-A	POL	SIG
H	-	-	-	-	-	-	-	-	-
OH	3	1	0	2	2	2	0	1	2
$O(CH_2)_6CH_3$	2	5	7	0	1	1	0	1	1
$O(CH_2)_5CH_3$	2	5	6	0	1	1	0	1	1
NO_2	5	2	0	0	0	0	2	0	3
F	5	1	0	0	1	0	0	0	5
$O(CH_2)_7CH_3$	2	5	8	0	1	1	0	1	1
CH2OH	2	2	2	2	2	0	0	1	0
NH_2	3	1	0	2	0	2	0	0	1
$OCH_2CH_2OCH_3$	2	3	4	0	1	1	0	1	1
Cl	3	1	0	0	0	0	0	1	3
CH_3	0	1	0	0	0	0	0	1	0
$CH_2O(CH_2)_3CH_3$	0	4	6	0	0	0	0	1	0
OCH_2CONH_2	3	3	2	1	1	1	0	0	1
OCF_3	4	3	1	0	0	0	2	0	3
CH_2OCH_3	0	2	3	0	1	0	0	1	0
$N(CH_3)_2$	1	2	0	0	1	2	0	1	1
Br	3	1	0	0	0	1	0	2	3
$O(CH_2)_3CH_3$	2	3	4	0	1	1	0	1	1
$NHCOCH_3$	3	2	0	1	1	1	0	1	1
OSO_2CH_3	4	2	1	0	0	0	1	2	3
$OCH_2C_6H_5$	2	4	2	0	1	1	0	1	1
CF_3	3	1	0	0	0	0	0	0	3
I	3	1	0	0	0	1	0	3	3
OCH_3	2	2	1	0	1	1	0	1	1

Table 3

Rule 3.1 - (coverage 119/0 train : 105/0 test)

```
great(A, B) :- struc(A, D, E, F), struc(B, h, C, h), h_donor(D, h_don0),
               pi_donor(D, pi_don1), flex(D, G), less4_flex(G).
```

Drug A is better than drug B if
drug B has no substitutions at positions 3 and 5 and
drug A at position 3 has hydrogen donor = 0 and
drug A at position 3 has π-donor = 1 and
drug A at position 3 has flexibility < 4.

Rule 3.2 - (coverage 244/71 train : 248/4 test)

```
great(A, B) :- struc(A, C, D, E), struc(B, F, h, G), not except3.2(A, B).
except3.2(A, B) :- struc(A, C, D, h), struc(B, E, h, F), h_donor(E, h_don0).
```

Drug A is better than drug B if
drug B has no substitution at position 4 unless
drug A has no substitution at position 5 and
drug B at position 3 has hydrogen donor = 0.

Rule 3.3 - (coverage 102/13 train : 33/0 test)

```
great(A, B) :- struc(A, G, H, I), struc(B, C, h, D), pi_donor(C, pi_don0),
               polar(C, E), great0_polar(E), h_acceptor(C, F), great0_h_acc(F).
```

Drug A is better than drug B if
drug B has no substitutions at position 4 and
drug B at position 3 has π-donor = 0 and
drug B at position 3 has polarity > 0 and
drug B at position 3 has hydrogen acceptor > 0.

Rule 3.4 - (coverage 129/2 train: 126/0 test)

```
great(A, B) :- struc(A, C, D, E), struc(B, G, h, h), h_donor(C, h_don0),
               pi_donor(C, pi_don1), flex(C, F), less4_flex(F),
               polarisable(G, H), less3_polari(H).
```

Drug A is better than drug B if
drug B has no substitutions at position 4 and 5 and
drug B at position 3 has polarisability < 3 and
drug A at position 3 has hydrogen donor = 0 and
drug A at position 3 has π-donor = 1 and
drug A at position 3 has flexibility < 4.

Rule 3.5 - (coverage 84/0 train: 52/0 test)

```
great(A, B) :- struc(A, C, D, E), struc(B, F, h, h), size(C, size2),
               h_donor(C, h_don0), polarisable(F, polari1), polar(C, G),
               great0_polar(G).
```

Drug A is better than drug B if
drug B has no substitutions at position 4 and 5 and
drug B at position 3 has polarisability =1 and
drug A at position 3 has size = 2 and
drug A at position 3 has hydrogen donor = 0 and
drug A at position 3 has polarity > 0.

Rule 3.6- (coverage 29/0 train: 16/0 test)

```
great(A, B) :- struc(A, C, D, E), struc(B, F, h, h), h_donor(C,h_don0),
               polarisable(C, polari1), flex(F, G), flex(C, H), great_flex(G, H),
               great6_flex(G).
```

Drug A is better than drug B if
drug B has no substitutions at position 4 and 5, and
drug B at position 3 has flexibility > 6 and
drug A at position 3 has polarisability = 1 and
drug A at position 3 has hydrogen donor = 0 and
drug A at position 3 is less flexible than drug B at position 3.

Rule 4.1-(coverage 289/72 train: 99/0 test)

```
great(A, B) :- struc(A, D, E, F), struc(B, h, C, h), not except4.1(A,B).
except4.1(A,B) :- struc(B, h, C, h), size(C, size3).
except4.1(A,B) :- struc(B, h, C, h), size(C, size2), h_acceptor(C,h_acc1).
```

Drug A is better than drug B if
drug B has no substitutions at position 3 and 5 unless
drug B at position 4 has size = 3 or
drug B at position 4 has size = 2 and hydrogen acceptor = 1.

Rule 4.2 - (coverage 187/2 train: 193/2 test)

great(A, B) :- struc(A, E, F, G), struc(B, C, D, h), not_h(E), polar(F, polar2).

Drug A is better than drug B if
drug B has no substitution at position 5 and
drug A has a substitution at position 3 and
drug A at position 4 has polarity = 2.

Rule 3&4.1 - (coverage 85/0 train: 55/0 test)

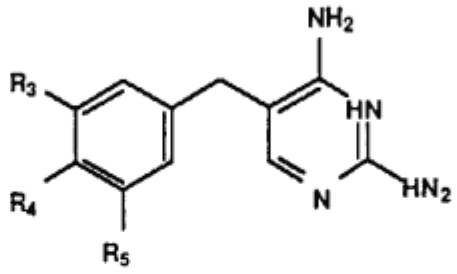
great(A,B) :- struc(A,C,D,E), struc(B,H,I,h), h_donor(C,h_don0), polar(C,F),
less5_polar(F), size(C,G), less5_size(G),
polarisable(I,J), less2_polari(J), sigma(I,K), great1_sigma(K).

Drug A is better than drug B if
drug B has no substitution at position 5 and
drug B at position 4 has polarisability < 2 and
drug B at position 4 has $\sigma > 1$ and
drug A at position 3 has hydrogen donor = 0 and
drug A at position 3 has polarity < 5 and
drug A at position 3 has size <5

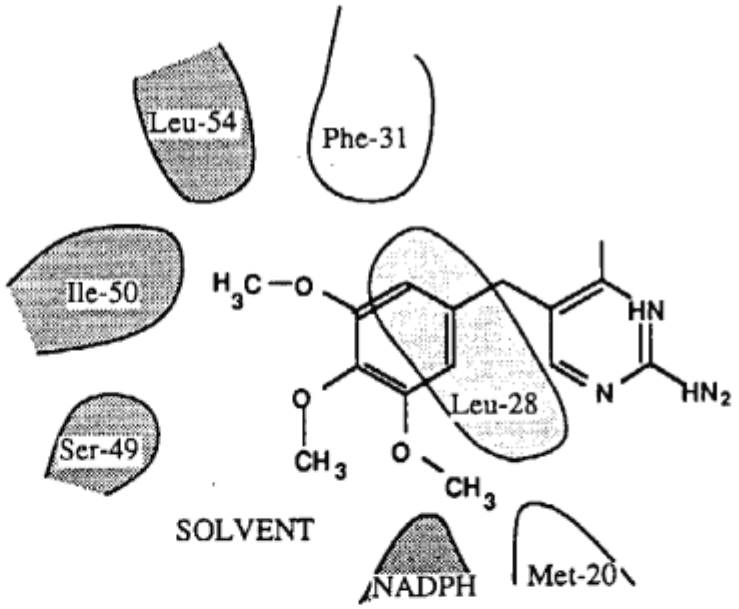
Table 4

Run	Rank correlation (Golem)		Drugs in testing
	training	testing	
main trial	0.92	0.46	45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55
validation 1	0.97	0.75	11, 31, 34, 42, 20, 24, 30, 23, 37, 8, 39
validation 2	0.93	0.57	16, 48, 43, 19, 22, 10, 41, 33, 35, 1, 9
validation 3	0.95	0.61	14, 3, 46, 6, 4, 53, 28, 25, 27, 51, 7
validation 4	0.94	0.86	54, 50, 49, 45, 12, 21, 44, 55, 38, 15, 18
validation 5	0.95	0.63	36, 47, 26, 32, 2, 17, 40, 13, 52, 29, 5

Ia


NH2
R3
HN
R4
N
HN2
R5

Ib


Leu-54
Phe-31
Ile-50
H3C—O
HN
N
HN2
Leu-28
Ser-49
O
CH3
O
CH3
SOLVENT
NADPH
Met-20

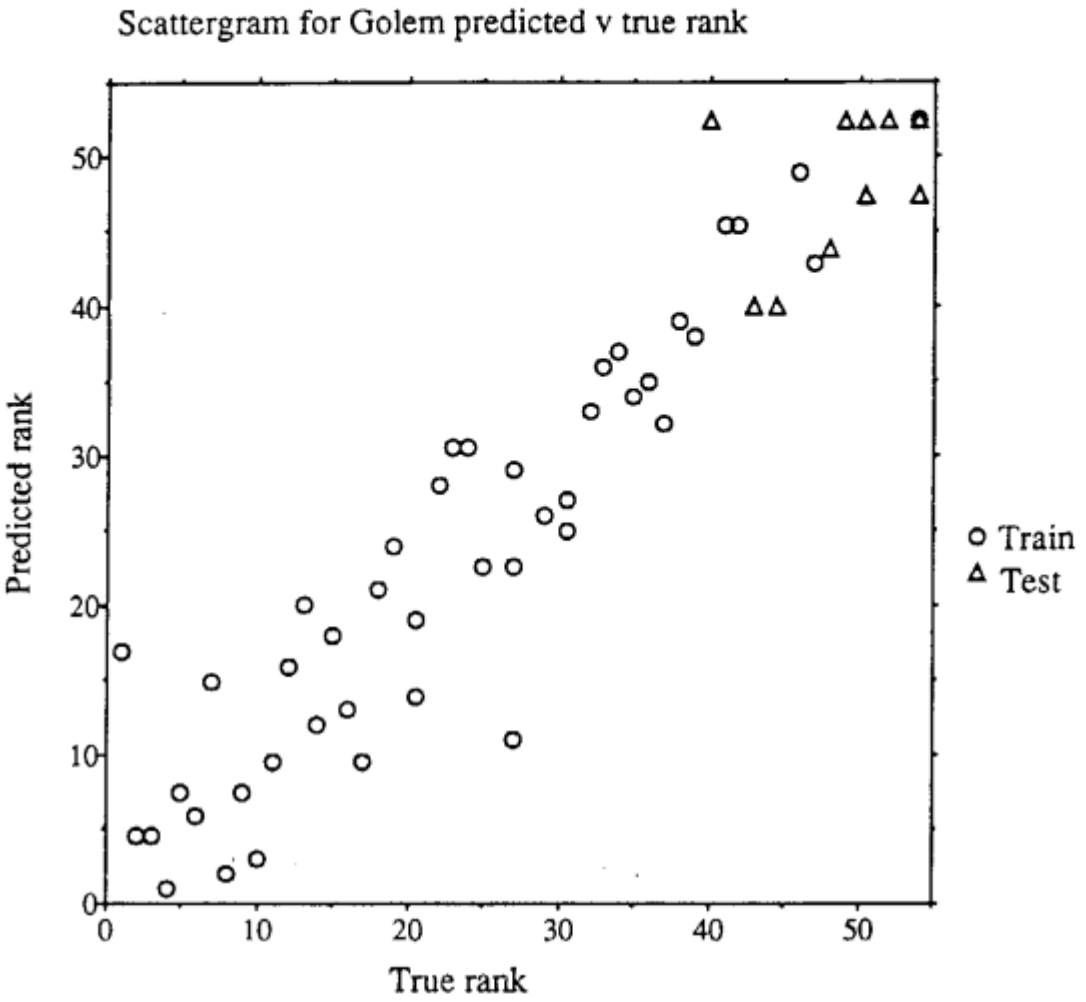
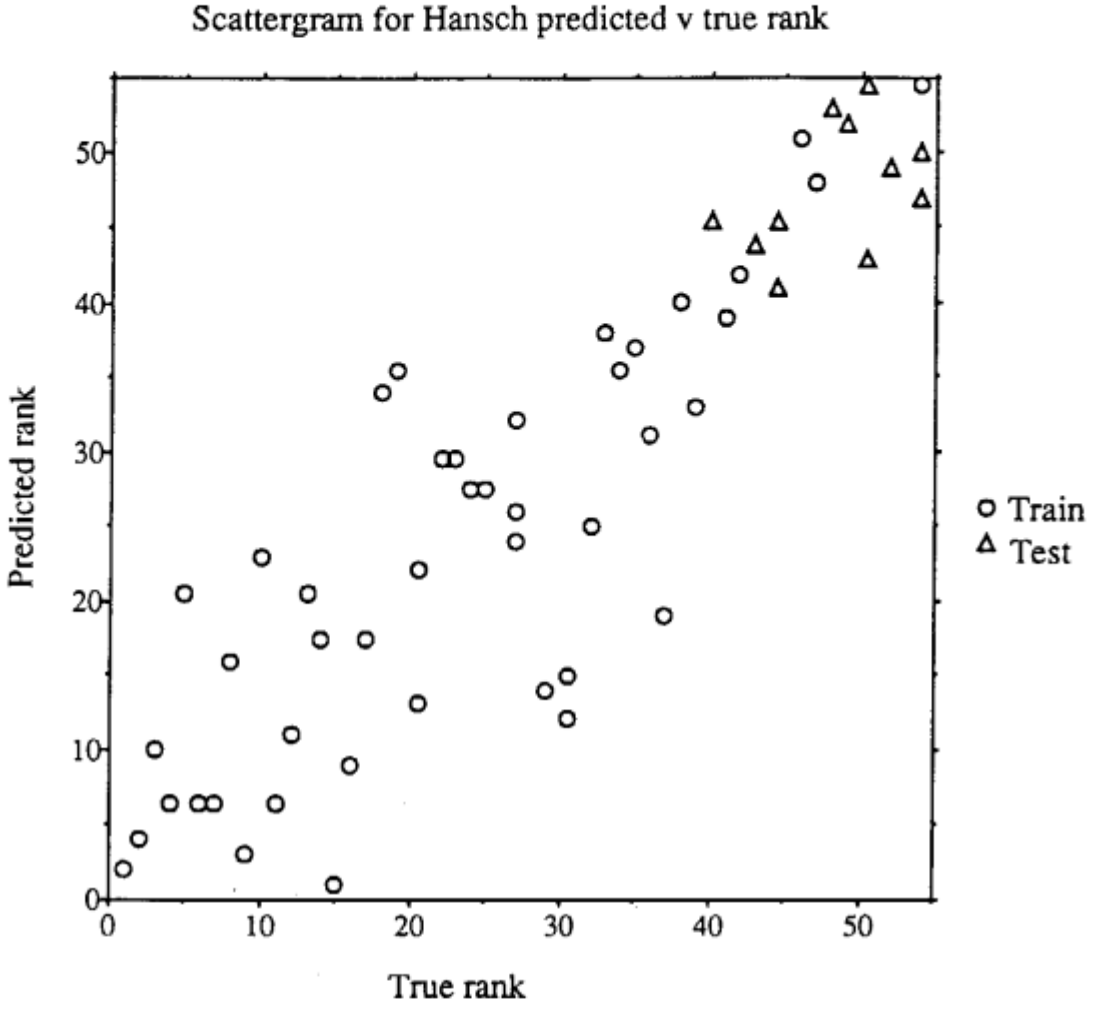

Scattergram for Golem predicted v true rank
Predicted rank
True rank
0
10
20
30
40
50
Train
Test

Fig 2a


Scattergram for Hansch predicted v true rank
Predicted rank
True rank
0
10
20
30
40
50
Train
Test

Protein secondary structure prediction using Logic

Stephen Muggleton[1,3], Ross D. King[1]
and Michael J.E. Sternberg[2]

[1]Turing Institute
George House
36 North Hanover St.
Glasgow G1 2AD
Scotland
UK
Tel. +44 41 552 6400
Fax. +44 41 552 2985

[2]Biomolecular Modelling Laboratory
Imperial Cancer Research Fund
P.O. Box 123, 44 Lincoln's Inn Fields
London WC2A 3PX
England
UK
Tel. +44 71 269 3565
Fax +44 71 430 2666

[3]To whom correspondence should be addressed.

Key Words: α-helix, secondary structure prediction, protein modelling, artificial intelligence, machine learning

Abstract

Many attempts have been made to solve the problem of predicting protein secondary structure from primary sequence but the best performance results are still disappointing. In this paper we show that the use of a machine learning algorithm which allows relational descriptions leads to improved performance. The ILP program Golem was applied to learning secondary structure prediction rules for alpha domain type proteins. The input to the Program consisted of 12 non-homologous proteins (1612 residues) of known structure, together with background knowledge describing the chemical and physical properties of the residues. Golem learned a small set of rules that predict which residues are part of α-helices - based on their positional relationships, chemical and physical properties. The rules were tested on 4 independent non-homologous proteins (416 residues) giving an accuracy of 81% (+/-2%). This is an improvement over the previously reported result of 73% by King & Sternberg on the same data using the machine learning program PROMIS, and the best previously reported result in the literature for the alpha domain of 76% achieved using a neural net approach. Machine learning also has the advantage over neural network and statistical methods in producing more understandable results.

1. Introduction

An active research area in the hierarchical approach to the protein folding problem is the prediction of secondary structure from primary structure[1,11]. Most of these approaches involve examining the Brookhaven database[12] of known protein structures to find general rules relating primary and secondary structure. However, this database is hard for humans to assimilate and understand as it consists of a large amount of abstract symbolic information although using molecular graphics provides some help. Today the best methods of secondary structure prediction achieve an accuracy of between 60-65%.[13] The generally accepted reason for this poor accuracy is that the predictions are being carried out using only local information, and long range interactions are not taken into account. Long range interactions are important because when a proteins folds up, regions of the sequence which are linearly far separated become close spatially. Established approaches to the problem of predicting secondary structure have involved: Bayesian statistical methods,[1] and hand-crafted rules by experts.[2] More recently a variety of machine learning methods have been applied: both neural networks[3,6] and symbolic induction.[7,11] An exact comparison between these methods is very difficult because different workers have used different types of proteins in their data sets.

One approach to get higher accuracy in the prediction of secondary structure is to decompose the problem into a number of sub-problems. This is done by splitting the data set of proteins into groups of the same type of domain structure, e.g. proteins with domains only having α-helices (alpha type domains), or β-strands (beta type domains), or alternate α-helices and β-strands (alpha/beta type domains). This allows the learning method to have a more homogeneous data set, allowing better prediction; but assumes a method of determining the domain type of a protein. The decomposition approach is adopted in this paper where we concentrate solely on proteins of domain type alpha. On these type of proteins, neural networks have achieved an accuracy of 76% on unseen proteins (using a slightly more homologous database than used in this paper).[13] These

proteins have also been studied using the symbolic induction program PROMIS, which achieved an accuracy of 73% on unseen proteins (using the same data as the present study).[8] Compared to the machine learning method used in this study, PROMIS has a limited representational power. This means that it was not capable of finding some of the important relationships between residues that the new method showed were involved in α-helical formation

In this paper, Inductive Logic Programming (ILP)[14] is applied to learning the secondary structure of alpha type domain proteins. ILP is a method for automatically discovering logical rules from examples and relevant domain knowledge. ILP is a new development within the field of symbolic induction, and marks an advance in that it is specifically designed to learn structural relationships between objects - a task particularly difficult for most machine learning or statistical methods. The ILP program used in this work is Golem.[15] Golem has had considerable previous success in other essentially relational application areas including Finite Element Mesh Design.[16] construction of Qualitative models.[17] and the construction of diagnostic rules for satellite technology.[18]

2. Methods

2.1. Golem

Golem is a program for Inductive Logic Programming (ILP). The general scheme for ILP methods is shown in Fig. 1. This scheme closely resembles that of standard scientific method. Observations are collected from the outside world (in the case of this paper the Brookhaven data bank). These are then combined by an ILP program with background knowledge to form inductive hypothesis (rules for deciding secondary structure). These rules are then experimentally tested on additional data. If experimentation leads to high confidence in the hypotheses' validity, they are added to the background knowledge.

Fig. 1.

In ILP systems, the descriptive languages used are subsets of first-order predicate calculus. Predicate logic is expressive enough to describe most mathematical concepts. It is also believed to have a strong link with natural language. This combination of expressiveness and ease of comprehension has made first-order predicate calculus a very popular language for artificial intelligence applications. The ability to learn predicate calculus descriptions is a recent advance within machine learning. The computer implementation of predicate logic used in Golem is the language Prolog. Prolog rules can easily express the learned relationships between objects such as molecular structures. Previous machine learning programs have lacked the ability to learn such relationships, and neural network and statistical learning techniques also have great difficulty. This gives ILP learning algorithms such as Golem a potential advantage in learning problems involving chemical structures.

Golem takes as input: positive examples, negative examples, and background knowledge described as Prolog facts. It produces as output: Prolog rules which are generalisations of the examples. The Prolog rules are the least general rules which, given the background knowledge, can produce the input examples and none of the negative examples. The method of generalisation is based on the logical idea of RLLG (Relative Least General Generalisation). The basic algorithm used in Golem is as follows. Firstly it takes a random sample of pairs of examples. In this application, this will be a set of pairs of residues chosen randomly from the set of all residues in all proteins represented. For each of these pairs Golem computes the set of all properties which are common to both residues. These properties are then made into a rule which is true of both the residues in the pair under consideration. For instance if the only common properties of the residues are that both residues are large and both are three residues distant from a more hydrophilic residue then Golem would construct the following explanation for their being part of an alpha-helix.

```
alpha(Protein,Position) :-
        residue(Protein,Position,R),
        large(R),
        P3 = Position+3,
        residue(Protein,Position,R3),
        more_hydrophilic(R3,R).      (see 2.3)
```

Having built such a rule for all chosen pair of residues, Golem takes each rule and computes the number of residues which that rules could be used to predict. Clearly these rules might predict some non-alpha-helix residues to be part of an alpha-helix. Golem therefore chooses the rule which predicts the most true alpha residues while predicting less than a predefined threshold of non-alpha-helix residues. Having found the rule for the best pair, Golem then takes a further sample of as yet unpredicted residues and forms rules which express the common properties of this pair together with each of the individual residues in the sample. Again the rule which gives best predictions on the training set is chosen. The process of sampling and rule building is continued until no improvement in prediction is produced. The best rule from this process is used to eliminate a set of predicted residues from the training set. The reduced training set is then used to build up further rules. When no further rules can be found the procedure terminates.

2.2. Database of proteins

Sixteen proteins were selected for the data set from the Brookhaven data bank.[1] The training proteins used were: 155C (cytochrome C550[19]), 1CC5 (cytochrome C5 oxidised[20]), 1CCR (cytochrome C[21]), 1CRN (crambin[22]), 1CTS (citrate synthase[23]), 1ECD (erythrocruorin reduced.deoxy[24]), 1HMQ (hemerythrin met[25]), 1MBS (myoglobin met[26]), 2B5C (cytochrome B5), 2C2C (cytochrome C2 oxidised), 2CDV (cytochrome C3), 3CPV (calcium-binding parvalbumin). The test proteins used were: 156B (cytochrome B562 oxidised[27]), 1BP2 (phospholipase A2[28]), 351C (cytochrome C551[29]), 8PAP (papain[30]) - in protein 8PAP only the first domain (residues 1 - 108) is used. the other domain is of type beta. These proteins have high resolution structure and alpha-type

domains (secondary structure dominated by α-helices and with little if any β-strands).[43] The proteins were also selected to be non-homologous (little structural or sequential similarity). This selection was done on the basis of knowledge of protein structure and biology; e.g. there is only one globin structure 1MBS (Myoglobin). The data set of proteins was randomly chosen from all the proteins to give a rough 70:30 split (Table I). Secondary structure was assigned using the Kabsch and Sander algorithm.[32]

Table I

2.3. Representation of the problem

Three types of file are input into Golem: foreground examples, background facts and negative examples. These are described in the following subsections.

2.3.1 Foreground and negative examples

Three types of file are input into Golem: facts that are true, facts that are false and background facts. The following is one of the foreground examples:

alpha(Protein name, Position).: e.g. alpha(155C, 105).

This says that residue 105 in protein 155C is an α-helix. The negative examples take the same form but state all residue positions in particular proteins in which the secondary structure is not in an α-helix secondary structure.

2.3.2 Background facts

The background information contains a large variety of information about protein structure. The most basic of this is the primary structure information. For instance the fact:

position(155C, 119, p).

says that the residue at position 105 in protein 155C is proline (the standard one 20 character coding for amino acids is used).

As Golem does not have arithmetic information built in, information has to be given

about the sequential relationships between the different positions (residues). These arithmetic-type relations allow indexing of the protein sequence relative to the residue being predicted. The first predicate describes 9 sequential positions. For instance the fact:

octf(19, 20, 21, 22, 23, 24, 25, 26, 27).

describing the sequence 19, .., 27 can be used to indexing the four flanking positions on either side of position 23. The second type gives sequences that are considered to be especially important in α–helices[3]. Thus for instance the background knowledge contains the facts:

alpha_triplet(5, 6, 9). alpha_pair(5, 8). alpha_pair4(5, 9).

The predicate alpha_triplet contain the numbers n, n+1, and n+4, in an alpha helix these residues will appear on the same face of the helix. Grouping these numbers together is a heuristic to allow preferential search for a common relationships between these residues. Similarly, the residues with positions in the alpha_pair predicate (n and n+3), and residues with positions in the predicate alpha_pair4 (n and n+4) are expected to occur on the same face of a helix.

Physical/chemical properties of individual residues are described by the unary predicates hydrophobic, very_hydrophobic, hydrophilic, positive, negative, neutral, large, small, tiny, polar, aliphatic, aromatic, hydro_b_don, hydro_b_acc, not_aromatic, small_or_polar, not_p, ar_or_al_or_m, not_k, aromatic_or_very_hydrophobic.[33] Each of these is expressed in terms of particular facts such as:

small(p).

meaning that proline is a small residue. The more complicated properties are given below:

hydro_b_don	-	hydrogen bond donator
hydro_b_acc	-	hydrogen bond acceptor
not_aromatic	-	the complement of the aromatic class
small_or_polar	-	either small or polar
not_p	-	everything but proline
not_k	-	everything but lysine
aromatic_or_very_hydrophobic	-	either aromatic or very hydrophobic
ar_or_al_or_m	-	either aromatic or aliphatic or methionine

The rather unusual looking logical combinations such as aromatic_or_very_hydrophobic has been previously found useful.[8] (For some runs, similar predicates to not_p were created for all twenty residue types).

Information was also given about the relative sizes and hydrophobicities of the residue was given. This was described using the binary predicates ltv and lth. Each of these is expressed in terms of particular facts such as:

ltv(X, Y)

meaning that X is smaller than Y (scale taken from Schulz & Schirmer[34]),

lth(X, Y)

meaning that X is less hydrophobic than (scale taken from Eisenberg[35]).

2.4. Experimental procedure

A run of Golem takes the form of asking Golem to find a good generalisation rules. These generalisation can then either be accepted or Golem can be asked to try and find another generalisation. A prediction rule was accepted if it had high accuracy and good coverage. If a rule was accepted, then the examples it covers are removed from the background observations (true and false facts), and the rule is added to the background information. Learning stops when no more generalisations can be found within set conditions

Golem was first run on the training data using the above background information. A certain amount of "noise" was allowed for in the data. and Golem was set to allow each rule to misclassify up to 10 negative instances. To be accepted rules had to have greater than 70% accuracy and coverage of at least 3%. If a rule had lower coverage than this it would not be statistically reliable. Learning was stopped when no more rules could be found meeting these conditions. Each rule found was typically very accurate (often in excess of 90% correct classification), overall around 60% of the instances where classified by the rules as a whole. The accuracy and coverage settings used to find the rules was based largely on subjective judgment and experience. Currently work is being carried out to replace the need for subjective judgment by objective measures from statistical and

algorithmic information theory.

To improve on the coverage found by these first rules the learning process was iterated. The predicted secondary structure positions found using the first rules (level 0 rules, appendix A) were added to the background information (Fig. 2), and then Golem was run again to produce new rules (level 1 rules). This forms a kind of bootstrapping learning process, with the output of a lower level of rules providing the input for the next level. This was needed because after the level 0 rules, the predictions made were quite speckled, i.e. only short sequences of α-helix predicted residues interspersed by predictions of coil secondary structure. The level 1 rules have the effect of filtering the speckled prediction and joining together the short sequences of α-helix predictions. The iterative learning process was repeated a second time, with the predicted secondary structure positions from the level 1 rules being added to the background information, and new rules found (level 2 rules). The level 2 rules had the effect of reducing the speckling even more, and clumping together sequences of α-helix. Some of the level 1 and 2 rules were finally generalised by hand with the formation of the symmetrical variants of the rules found by Golem.

Fig. 2.

3. Results

Applying Golem to the training set produced twenty one level 0 rules, five symmetrical level 1 rules, and two symmetrical level 2 rules (appendix A). These rules combined together to produce a Q_3 accuracy of 78% in the training set and 81% in the test set (Tables II, III). These are the most accurate predictions found by any method to date. The Q_3 accuracy is defined as:

$$((TP + TN) / T) * 100$$

where TP is the number of correct helical predictions, TN is the number of correct coil predictions, T is the total number of residues. The standard error in this prediction accuracies was estimated to be around 2%. Standard error is calculated:

$$\sqrt{(P (1 - P) / T)}, \text{ where } P = Q_3 / 100$$

The prediction rate of training and test set agrees within the range of the standard error; however the figure of 81% prediction on the test set at level 2 may be a slight overestimate, with a level of 79% being more consistent with the test set prediction. This accuracy still compares favourably with previous results on this problem. The residue by residue predictions are given in Fig. 3. These give an intuitive feel for how accurate the predictions are. Although Q_3 measure is the accepted measure for accuracy, it would be better to have a measure more closely related to the information given about conformation, e.g. are the correct number of α-helices predicted. It would also be useful to take into account that the boundary between α-helices and coil secondary structure can be ambiguous.

Table II

Table III

Fig. 3

Fig. 4

The rules generated by Golem can be considered to be hypotheses about the way α–helices form (Fig. 4). They define patterns of relationships which, if they exist in a sequence of residues, will tend to cause a specified residue to form part of an alpha-helix. For example considering rule 12, this rule specifies the properties of 8 sequential residues which if held will cause the middle residue in sequence (residue B) to form part of a helix. These rules are of particular interest because they were generated automatically and do not reflect any preconceived ideas about possible rule form (except those unavoidably built into the selection of background information). The rules raise many questions about α-helix formation. Considering rule 12 again, the residue p (proline) is disallowed in a number of positions, but allowed in others - yet proline is normally considered to disrupt proteins. It is therefore of interest to understand under exactly what circumstances proline can be fitted into an α–helix. One of the most interesting features to be brought out by the rules was the importance of relative size and hydrophobicity in helix formation, not just absolute values. It is an idea which warrants further investigation (N.B. relative values cannot easily be

used in statistical or neural network methods). One technique of making the level 0 rules more comprehensible to to display them on a helical wheel plan, this is a projection showing the α–helix from above and the different residue types sticking out from the sides at the correct angle, see rule 12 in Fig. 5. Rule 12 shows amphiphathicity, the tendency in α–helices of hydrophobic residues and hydrophilic residues to occur on opposite faces of the α-helix. This property is considered central in α-helical structure.[2,35] However, most of the level 0 rules when displayed on helical wheels do not display such marked amphiphathicity, and a detailed survey of the location of the positive and negative examples of the occurrences of the rules is required. This would involve a database analysis combined with the use of interactive graphics.

Fig. 5

One problem raised by protein structure experts with the rules, is that although they are much more easily understood than an array of numerical parameters or an Hinton diagram for a neural network, they appear somewhat complicated. This brings up the question about how complicated the rules for forming α-helices are,[9] and it may be that any successful prediction rules are complicated. The failure of Rooman and Wodak[36] to produce especially high accuracy using patterns with only 3 residues specifies lends support to this idea. One approach to make the rules easier to understand may be to find over-general rules, and then generalise the exception to these rules.[37] In such a procedure the over-general rules would tend to be easier to comprehend.

The meaning of the level 1 and 2 rules are much clearer to understand. In proteins secondary structures elements involve sequences of residues, e.g. an α-helix may occupy ten sequential residues, and then be followed by eight residues in a region of coil. However, the level 0 rules output predictions based only of individual residues. This makes it possible for the level 0 rules to predict a coil in the middle of a sequence of residues predicted to be of α-helix type; this is not possible in terms of structural chemistry.

This shortcoming is dealt with by the level 1 and 2 rules which group together isolated residue predictions to make predictions of sequences. For example: rule 23 predicts that a residue will be in an α-helix if both residues on either side have already been predicted to be α-helices; similarly rule 24 predicts that a residue will be in an α−helix if two residues on one side, and one residue on the other side have already been predicted to be α-helices.

4. Discussion

It is intended to extend the application of Golem to the protein folding problem by adding more background knowledge. One form of background knowledge that will be added is the division of each α-helix into three parts (beginning, middle and end). Analysis has shown that a specific pattern of residues can occur at these positions.[38,39] This is thought to occur because the physical/chemical environment experienced by the three different sections is very different: both the beginning and end have close contact with coil regions, while the middle does not; also a dipole effect cause the end of an α-helix to be more negative than the beginning. Evidence for the usefulness of this division idea is given by by rules 17 and 18. The structure of these rules suggests that they are biased towards the end of α-helices - the residues predicted by the rules occur at the end of the sequence of defined primary structure. Examination of the occurrences of these rules confirms this, showing that their predictions tend to occur at the end of α-helices (and often occur together). Protein secondary structure prediction methods normally only consider local interactions and this is the main reason for their poor success. On way of tackling this problem is that used in this paper of iterative predictions based on previous predictions. This may be extended by using well defined long range interaction such as super-secondary structure and domain structures,[40] or by using models of constraints.[41] It is if hoped that with the addition of such new types of knowledge, that the prediction accuracy of Golem will gradual improve, making it a more and more useful biological tool.

The advantages Golem enjoys in the protein prediction problem should apply

equally well to other problems in chemistry and molecular biology. This is because chemicals are structural objects and it is most natural to reason and learn about them using relational knowledge. One such application area is the human genome project which is producing a vast amount of sequential DNA data, and has associated with this data are a number of important learning problems, e.g. the recognition of promoter sequences, the recognition of translation initiation sequences, etc. Such problems have been investigated using neural network methods,[42] and it would be instructive to investigate how well Golem does in comparison. Golem could also be applied to problems in chemistry. Some important early machine learning work was done learning the rules for the breakup of molecules in mass spectroscopy (Meta-DENDRAL),[43] such a problem would be well suited for Golem.

Acknowledgments

Part of this research was supported by the Esprit Basic Research Action "ECOLES". Stephen Muggleton is supported by an SERC Research Fellowship. Ross D. King is supported by the Esprit project "Statlog". Michael Sternberg is employed by the Imperial Cancer Research Fund. This research was carried out at the Turing Institute.

References

1. Gibrat, J.E., Garnier, J., Robson, B. Further development of protein secondary structure prediction using information theory: New parameters and consideration of residue pairs. J. Mol. Biol. 198:425-443, 1987.
2. Lim, V.I. Structural principles of the globular organization of protein chains: A stereochemical theory of globular protein secondary structure. J. Mol. Biol. 80:857-872, 1974.
3. Qian, N., Sejnowski, T.J. Predicting the secondary structure of globular proteins with neural network models. J. Mol. Biol. 119:537-555, 1988.
4. Holley, L.H., Karplus, M. Protein secondary structure prediction with a neural network. Proc. Nat. Acad. Sci. U.S.A. 86:152-156, 1989.
5. McGregor, M.J., Flores, T.P., Sternberg, M.J.E. Prediction of β-turn in proteins using neural networks. Protein Eng. 2:521-526, 1989.
6. McGregor, M.J., Flores, T.P., Sternberg, M.J.E. Corrigendum: Prediction of β-turn in proteins using neural networks. Protein Eng. 3:459-460, 1990.
7. Seshu, R., Rendell, L., Tcheng, D. Managing constructive induction using subcomponent assessment and multiple-objective optimization. In: "Proceedings of the Sixth International Workshop in Change of Representation and Inductive Bias". NY:Cornell, 1988:293-305.
8. King, R.D., Sternberg, M.J.E. Machine learning approach for the prediction of protein secondary structure. J. Mol. Biol. 216:441-457, 1990.
9. Sallantin, J., Pingand, P. Artificial intelligence for genomic interpretation. In: " Symbols versus Neurons?" Stender, J., Addis,T. (eds.) Netherlands: IOS Press. 1990:99-117.
10. Cost, S., Salzberg, S. A Learning Algorithm for Symbolic Sequence Classification. In: "A.A.A.I. Workshop Proceedings: A.I. Approaches to Classification and Pattern Recognition in Molecular Biology" Noordewier, N. (ed.) (in press) 1991.
11. Zhang, X., Mesirov, J., Waltz, D. Determinant factors on the accuracy of protein secondary structure prediction. In: "A.A.A.I. Workshop Proceedings: A.I.

Approaches to Classification and Pattern Recognition in Molecular Biology" Noordewier, N. (ed.) (in press) 1991.

12. Bernstein, F.C., Koetzle, T.F., Williams, G.J.B., Meyer, E.F.Jr., Brice, M.D., Rodgers, J.R., Kennard, O., Shimanouchi, T., Tasumi, M. The protein data bank: A computer-based archival file for macromolecular structures. J. Mol. Biol. 112:535-542, 1977.
13. Kneller, D.G, Cohen, F.E., Langridge, R. Improvements in protein secondary structure prediction by an enhanced neural network. J. Mol. Biol. 214:171-182, 1990.
14. Muggleton, S. Inductive logic programming. New Generation Computing 8:295-318, 1991.
15. Muggleton, S., Feng, C. Efficient induction of logic programs. In: "Proceedings of the first conference on algorithmic learning theory" Arikawa, S., Goto, S., Ohsuga, S., Yokomori, T. (eds) Tokyo: Japanese Society for Artificial Intelligence, 1990:368-381.
16. Dolsak, B., Muggleton, S. In: "Proceedings of the International Workshop on Inductive Logic Programming" Muggleton, S. (ed.) (in press) 1991.
17. Bratko, I., Muggleton, S., Varsek, A. Learning qualitative models of dynamic systems. In: "Machine Learning: Proceedings of the Eighth International Workshop" Birnbaum, L.A., Collins, G.(eds.) San Mateo: Morgan Kaufmann, 1991:385-388.
18. Feng, C. Inducing temporal fault diagnostic rules from a qualitative model. In: "Machine Learning: Proceedings of the Eighth International Workshop" Birnbaum, L.A., Collins, G.(eds.) San Mateo: Morgan Kaufmann, 1991:403-406.
19. Timkovich, R., Dickerson, R.E. The structure of *Paracoccus denitrificans* cytochrome SC=550=. J. Biol. Chem. 251:4033-4046, 1976.
20. Carter, D.C., Melis, K.A., O'Donnel, S.E., Burgess, B.K., Furey, W.F., Wang, B.C., Stout, C.D. Crystal structure of Azotobacter cytochrome SC=5= at 2.5 Angstoms resolution. J. Mol. Biol. 184:279-295, 1985.
21. Ochi, H., Hata, Y., Tanaka, N., Kakudo, M., Sakurai, T., Aihara, S., Morita, Y. Structure of rice ferricytochrome SC at 2.0 Angstroms resolution. J. Mol. Biol.

166:407-418, 1983.

22. Hendrickson, W.A., Teeter, M.M. Structure of the hydrophobic protein crambin determined directly from the anomalous scattering of sulpher. Nature 290:103-107, 1981.

23. Remington, S., Wiegand, G., Huber, R. Two different forms of citrate synthase at 2.7 and 1.7 Angstroms resolution. J. Mol. Biol. 158:111-152, 1982.

24. Steigemann, W., Weber, E. Structure of erythrocruorin in different ligand states refined at 1.4 Angstroms resolution. J. Mol. Biol. 127:309-338, 1979.

25. Stekemp, R.E., Sieker, L.C. Jensen, L.H. Adjustment restraints in the refinement of methemythrin and azidomethemerythrin at 2.0 Angstroms resolution. Acta. Crystallogr., Sect B. 39:697, 1983.

26. Scouloudi, H., Baker, E.N. X-ray crystallographic studies of seal myoglobin. The molecule at 2.5 angstroms resolution. J. Mol. Biol. 126:637-660, 1978.

27. Lederer, F., Glatigny, A. Bethge, P.H., Bellamy, H.D., Mathews, F.S. Improvement of the 2.5 Angstroms resolution model of cytochrome $B=562= by redetermining the primary structure and using molecular graphics. J. Mol. Biol. 148:427-448, 1981.

28. Dijkstra, B.W., Kalk, K.H., Hol, W.G.J., Drenth, J. Structure of bovine pancreatic phospholipase A2 at 1.7 Angstroms resolution. J. Mol. Biol. 147:97-123, 1981.

29. Matsuura, Y., Takano, T., Dickerson, R.E. Structure of cytochrome SC=551= from *P. aeruginosa* refined at 1.6 Angstroms resolution and comparison of the two redox forms. J. Mol. Biol. 156:389-409, 1982.

30. Kamphuis, I.G., Kalk, K.H., Swarte, M.B.A., Drenth, J. Structure of papain refined at 1.65 Angstroms resolution. J. Mol. Biol. 179:233-256, 1984.

31. Sheridan, R.P., Dixon, S., Venkataghaven, R. Kuntz, I.D., Scott, K.P. Amino acid composition and hydrophobicity patterns of protein domains correlate with their structures. Biopolymers 24:1995-2023, 1985.

32. Kabsch, W., Sander, C. Dictionary of protein structure: Pattern recognition of hydrogen-bonded and geometrical features. Biopolymers 22:2577-2637, 1983.

33. Taylor, W.R. The classification of amino acid conservation. J. Theor. Biol.

119:205-221, 1986.

34. Schulz, G.E. & Schirmer, R.H. Principles of Protein Structure. Springer-Verlag, 1978.

35. Eisenberg, D. Three-dimensional structure of membrane and surface proteins. Ann. Rev. Biochem. 53:595-623, 1984.

36. Rooman, M.J. Wodak, S.J. Identification of predictive sequence motifs limited by protein structure data base size. Nature 335:45-49, 1988.

37. Bain, M. Experiments in non-monotonic learning. In: "Machine Learning: Proceedings of the Eighth International Workshop" Birnbaum, L.A., Collins, G.(eds.) San Mateo: Morgan Kaufmann, 1991:380-384.

38. Presta, L.G., Rose, G.D. Helix signals in proteins. Science 240:1632-1641, 1988.

39. Richardson, J.S., Richardson, D.C. Amino acid preferences for specific locations at the ends of α–helices. Science 240:1648-1652, 1988.

40. Schulze-Kremer, S., King, R.D. IPSA - Inductive Protein Structure Analysis. In: "A.A.A.I. Workshop Proceedings: A.I. Approaches to Classification and Pattern Recognition in Molecular Biology" Noordewier, N. (ed.) (in press) 1991.

41. Murzin, A.G., Finkelstein, A.V. General architecture of the α-helical globule. J. Mol. Biol. 204:749-769, 1988.

42. Stromo, G.D., Schneider, L.M., Gold, L.M., Ehrenfeucht, A. Use of the 'Perceptron' algorithm to distinguish translational initiation sites. Nucleic Acids Research 10:2997-3010, 1982.

43. Buchanan, B.G., Feigenbaum, E.A. Dendral and Meta-Dendral: Their application dimension. In: "Readings in Artificial Intelligence" Webster, B.L., N.J. Nilson N.J. (eds.) Palo Alto: Tioga publishing company, 1981:313-322.

Alpha		alpha	not alpha	beta	not beta	turn	not turn	Total
Train	No.	848	764	45	1567	719	893	1612
	ratio	0.526	0.474	0.028	0.972	0.446	0.554	
Test	No.	217	199	10	406	189	227	416
	ratio	0.522	0.478	0.024	0.0976	0.454	0.546	
All	No.	1065	963	55	1973	908	1120	2028
	ratio	0.525	0.475	0.027	0.973	0.448	0.552	

Table I Data statistics. Statistics of the random split of the data into training and test sets. The top row titles are the types of secondary structure, the left column titles are the splits of the data into training and test sets. No is the the number of residues of that secondary structure type, ratio is the ratio (secondary type. no / total no.).

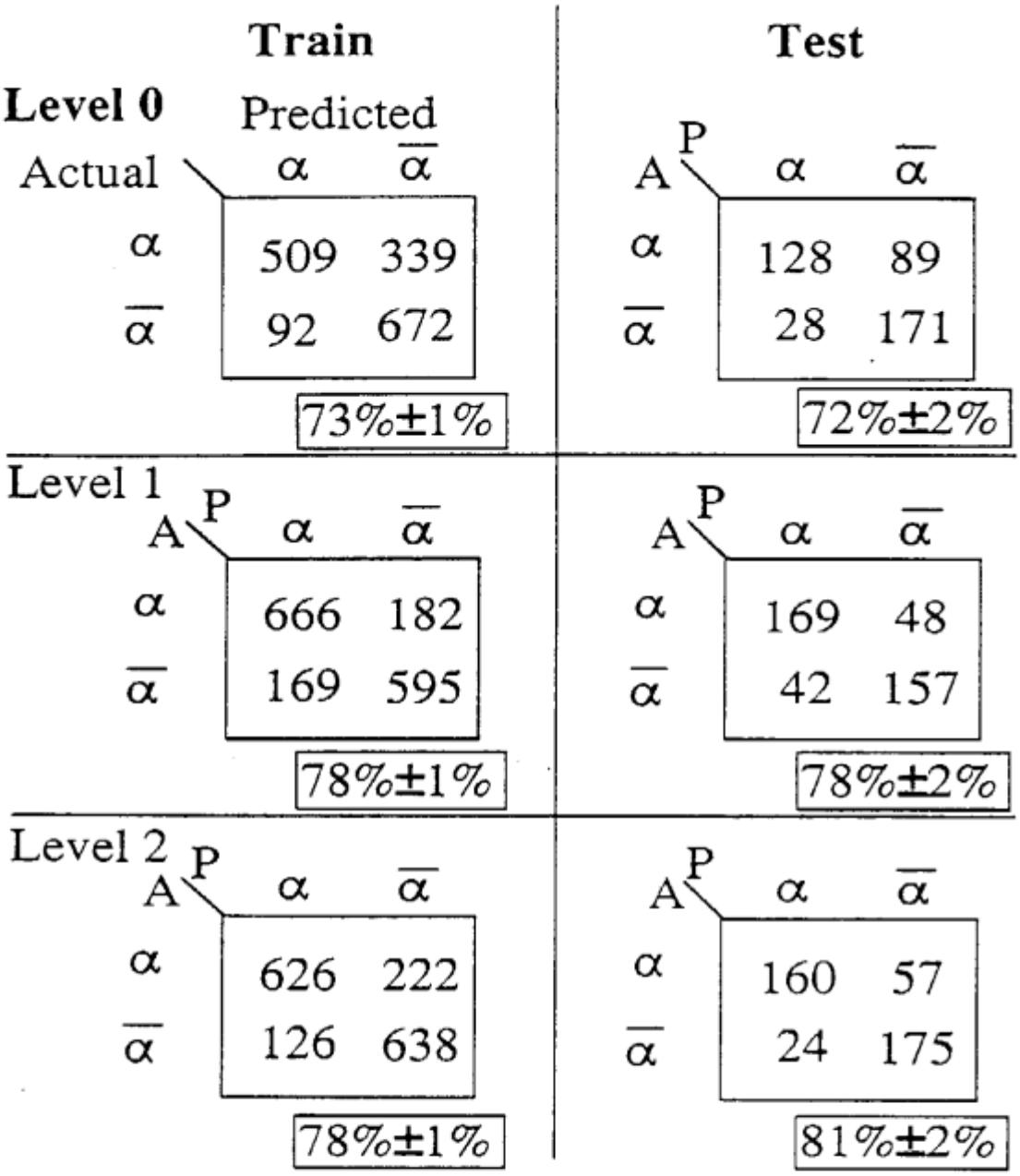


Table II Results summary. Confusion matrices and Q_3 percentage accuracies of rules found (P = predicted, A= actual). Each matrix has the following form A B

C D

The Q_3 percentage accuracies below each matrix are calculated as P * 100 where

P = (A+D) / (A+B+C+D)

Each percentage is followed by standard error (i.e. +/-2). Standard error is given as S * 100 where S = $\sqrt{}$(P(1-P) / (A+B+C+D))

No	Id.	Type	Q3
1	155C	Train	87.6
2	1CC5	Train	90.4
3	1CCR	Train	66.7
4	1CRN	Train	80.4
5	1CTS	Train	76.2
6	1ECD	Train	76.5
7	1HMQ	Train	68.1
8	1MBS	Train	76.5
9	2B5C	Train	82.4
10	2C2C	Train	88.4
11	2CDV	Train	82.2
12	3CPV	Train	77.8
13	156B	Test	76.7
14	1BP2	Test	81.3
15	351C	Test	78.0
16	8PAP	Test	85.2

Table III Protein results. The Q_3 accuracy of the predictions for each individual protein.

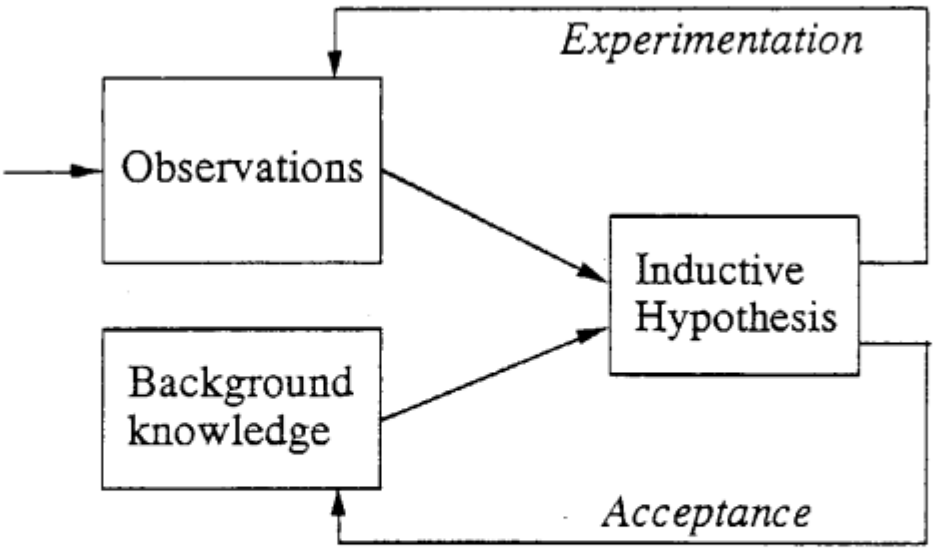


Fig 1 Inductive Logic Programming scheme

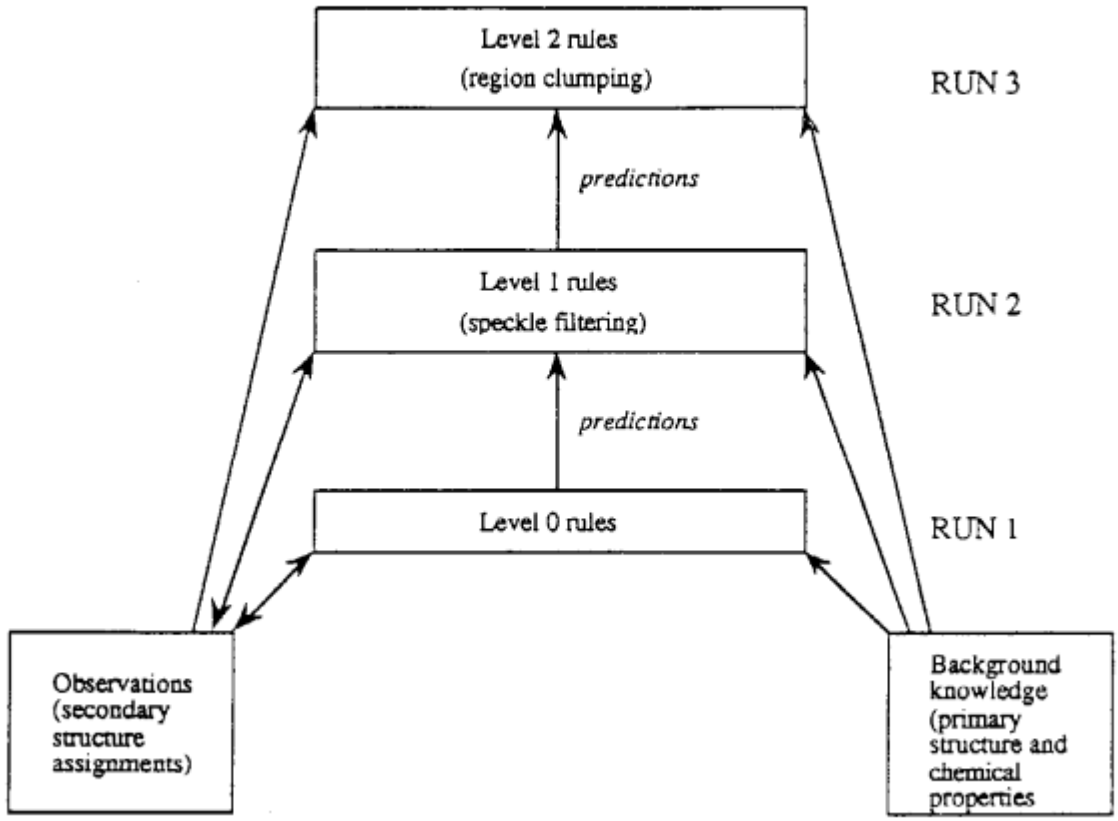


Fig. 2 Process used to generate the three levels of rules, showing the flow of information.

Training Proteins

Fig. 3

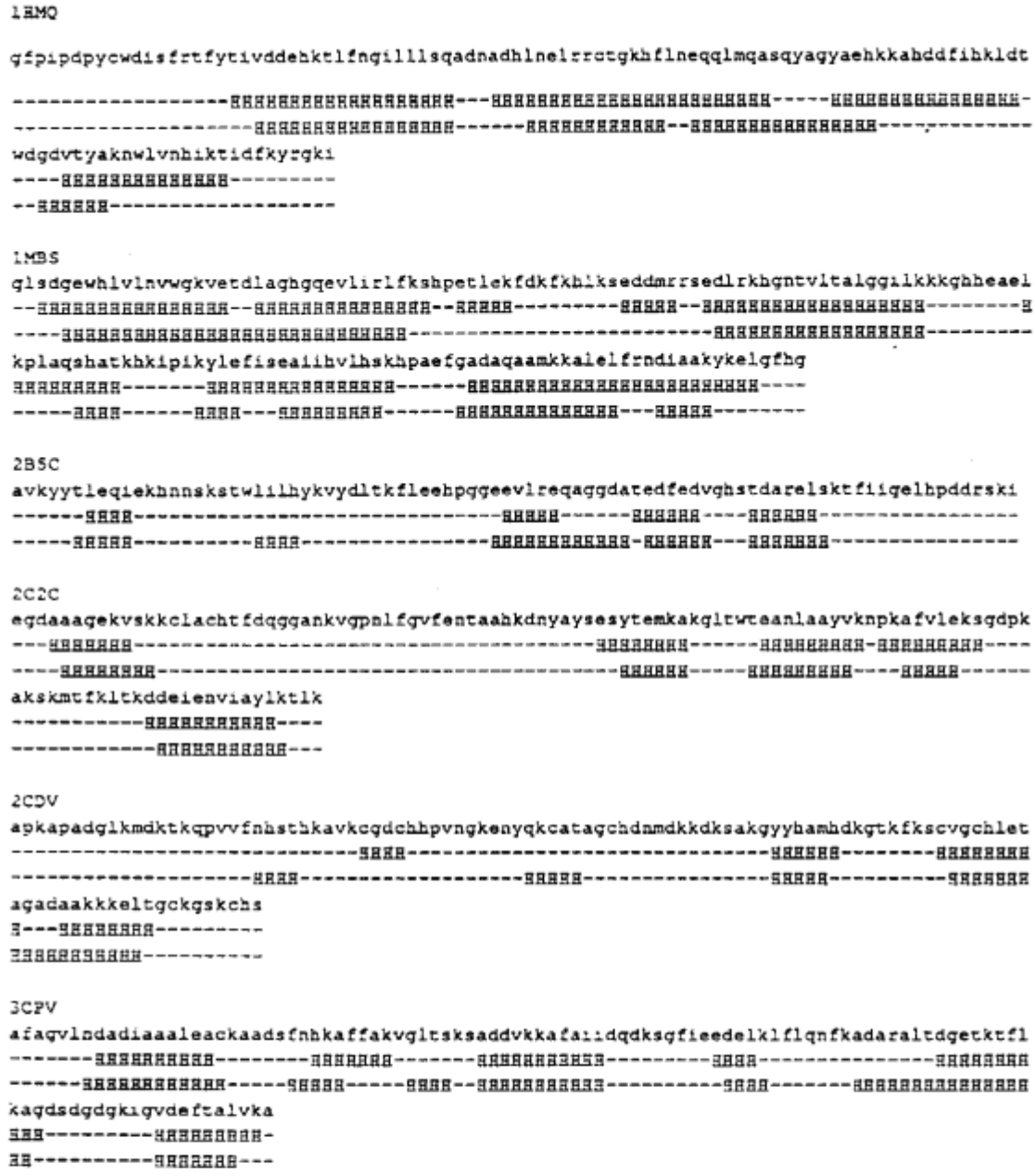


Fig. 3

Test Proteins

Fig. 3 Residue by residue predictions of secondary structute for the training and test data. The top line is the primary structure of the proteins. The middle line is the actual secondary structure. The bottom line is the predicted secondary structure. H signifies a residue with α-helix secondary structure. - signifies a residue with coil secondary structure.

Fig. 4
The list of the rules found by Golem at level 0, level 1, and level 2. The performance of each rule on the training and test data is given: as the number of correctly predicted residues and wrongly predicted residues (in rule 1 on the training data: 48 residues correct and 8 residues wrong), and the percentage accuracy and coverage. The rules are in Prolog format: Head :- Body. This means that if the conditions in the body are true then the head is true. Taking the example of rule 1:

There is an alpha helix residue in protein A at position B (the head) if:

at position D in protein A (position B - 4) the residue is not aromatic and not lysine,

and at position F in protein A (position B - 2) the residue is hydrophobic,

and at position D in protein A (position B - 1) the residue is not aromatic and not proline,

and at position B in protein A the residue is not aromatic and not proline.

and at position H in protein A (position B + 1) the residue is not proline and not lysine,

and at position I in protein A (position B + 2) the residue is hydrophobic and has a lower hydrophobicity than the residue at position D and a lower volume than the residue at position G,

and at position K in protein A (position B + 4) the residue is not aromatic and has a lower hydrophobicity than the residue at position F.

For the level 1 rules, a prediction of a helix by a level 0 rule is signified by: a1(Protein,Position). The positions of predictions made by the level 1 rules used by the level 2 rules are signified by: a2(Protein,Position).

```
% level 0 rules
% JOINT(1-21) TRAIN: 509/92 (85%acc,6%cov) TEST: 128/28 (82%acc,59%cov)

% RULE 1     TRAIN: 48/8 (84%acc,5%cov)     TEST: 12/0 (100%acc,5%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,N), not_aromatic(N), not_k(N),
   position(A,F,L), hydrophobic(L),
   position(A,G,O), not_aromatic(O), not_p(O),
   position(A,B,C), not_aromatic(C), not_p(C),
   position(A,H,Q), not_p(Q), not_k(Q),
   position(A,I,M), hydrophobic(M), lth(N,M), ltv(M,O),
   position(A,K,P), not_aromatic(P), lth(P,L).

% RULE 2     TRAIN: 30/1 (97%acc,4%cov)     TEST: 10/0 (100%acc,5%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,Q), not_p(Q), not_k(Q),
   position(A,E,O), not_aromatic(O), small_or_polar(O),
   position(A,F,R),
```

```
   position(A,G,P), not_aromatic(P),
   position(A,B,C), very_hydrophobic(C), not_aromatic(C),
   position(A,H,M), large(M), not_aromatic(M),
   position(A,I,L), hydrophobic(L),
   position(A,K,N), large(N), ltv(N,R).

% RULE 3     TRAIN: 34/3 (92%acc,4%cov)     TEST: 7/0 (100%acc,3%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,L), neutral(L), aromatic_or_very_hydrophobic(L),
   position(A,E,R), not_p(R), not_k(R),
   position(A,F,Q), small_or_polar(Q), not_k(Q),
   position(A,B,C), neutral(C),
   position(A,G,O), not_aromatic(O),
   position(A,H,P), not_aromatic(P), not_p(P), not_k(P),
   position(A,I,M), neutral(M), not_aromatic(M), not_p(M),
   position(A,J,N), neutral(N), aromatic_or_very_hydrophobic(N),
ltv(N,Q).

% RULE 4     TRAIN: 45/2 (96%,5%cov) TEST: 14/1 (93%acc,6%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,E,M), neutral(M),
   position(A,F,O), not_aromatic(O), not_p(O), not_k(O),
   position(A,G,P), not_aromatic(P),
   position(A,B,C), large(C), not_aromatic(C), not_k(C),
   position(A,H,N), neutral(N), aromatic_or_very_hydrophobic(N),
   position(A,I,R), not_p(R),
   position(A,J,Q), not_aromatic(Q), not_p(Q), not_k(Q),
   position(A,K,L), hydrophobic(L).

% RULE 5     TRAIN: 49/12 (80%,6%cov)       TEST: 10/1 (91%acc,5%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,M), not_aromatic(M), not_p(M), not_k(M),
   position(A,E,O), small_or_polar(O), not_p(O), not_k(O),
   position(A,F,N), not_aromatic(N), not_p(N),
   position(A,G,P), not_p(P),
   position(A,B,C), hydrophobic(C), lth(M,C),
   position(A,H,Q), not_p(Q), lth(Q,C),
   position(A,I,L), hydrophobic(L),
   position(A,J,S), ltv(S,N),
   position(A,K,R), not_k(R).

% RULE 6     TRAIN: 31/2 (94%acc,3%cov)     TEST: 7/1 (88%acc,3%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,N), not_aromatic(N), small_or_polar(N),
   position(A,E,O), not_aromatic(O), not_k(O),
   position(A,F,P), not_aromatic(P),
   position(A,G,M), hydrophilic(M), hydro_b_acc(M),
   position(A,B,C), hydrophobic(C),
   position(A,H,Q), not_aromatic(Q), not_k(Q),
   position(A,I,R), not_aromatic(R), small_or_polar(R),
   position(A,J,L), hydrophobic(L), not_aromatic(L), small_or_polar(L)

% RULE 7     TRAIN: 29/1 (97%acc,3%cov)     TEST: 7/1 (88%acc,3%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,E,M), polar(M), hydro_b_acc(M),
```

```
    position(A,G,L), large(L), not_k(L),
    position(A,B,C), large(C), not_k(C),
    position(A,H,O), not_p(O), not_k(O),
    position(A,J,P),
    position(A,K,N), not_aromatic(N), lth(N,P).

% RULE 8    TRAIN: 40/8 (83%acc,5%cov)    TEST: 6/1 (86%acc,3%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
    position(A,D,N), not_aromatic(N), small_or_polar(N),
    position(A,E,L), hydrophobic(L), large(L),
    position(A,F,Q), ltv(L,Q),
    position(A,B,C), not_aromatic(C), not_p(C),
    position(A,H,P), not_p(P), not_k(P),
    position(A,I,M), neutral(M),  large(M),
    position(A,K,O), not_aromatic(O), small_or_polar(O), not_p(O).

% RULE 9    TRAIN: 40/2 (95%,5%cov) TEST: 16/3 (84%acc,7%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
    position(A,D,M), not_aromatic(M), not_k(M),
    position(A,E,N), not_aromatic(N), small_or_polar(N), not_k(N),
    position(A,F,R), not_k(R),
    position(A,G,O), not_aromatic(O), not_p(O), not_k(O),
    position(A,B,C), not_aromatic(C),
    position(A,H,P), not_aromatic(P), not_p(P), not_k(P),
    position(A,J,L), hydro_b_don(L), lth(L,C), ltv(L,P),
    position(A,K,Q), not_aromatic(Q), not_k(Q).

% RULE 10   TRAIN: 33/3 (92%acc,4%cov)    TEST: 9/2 (82%acc,4%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
    position(A,D,N), not_aromatic(N), not_k(N),
    position(A,E,O), not_aromatic(O),
    position(A,F,P), not_aromatic(P), not_p(P),
    position(A,G,R), not_p(R), not_k(R),
    position(A,B,C), not_p(C), not_k(C),
    position(A,H,Q), not_aromatic(Q),
    position(A,I,L), hydrophobic(L), ltv(C,L), ltv(N,L), ltv(P,L),
    position(A,J,M), hydrophobic(M), not_aromatic(M),
    position(A,K,S), not_p(S), not_k(S).

% RULE 11   TRAIN: 40/3 (93%acc,5%cov)    TEST: 9/2 (82%acc,4%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
    position(A,D,O), not_aromatic(O), not_k(O),
    position(A,E,P), not_aromatic(P),
    position(A,F,Q), not_aromatic(Q), not_p(Q), not_k(Q),
    position(A,G,R), not_p(R), ltv(R,P),
    position(A,B,C), not_aromatic(C),
    position(A,H,N), large(N), ltv(N,L),
    position(A,I,L), hydrophobic(L),
    position(A,J,M), hydrophobic(M), not_aromatic(M),
    position(A,K,S), not_p(S).

% RULE 12   TRAIN: 58/3 (95%acc,7%cov)    TEST: 13/3 (81%acc,6%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
    position(A,F,Q), not_p(Q),
    position(A,G,N), not_aromatic(N), not_p(N),
```

```
   position(A,B,C), large(C), not_aromatic(C), not_k(C),
   position(A,H,L), hydrophobic(L), not_k(L),
   position(A,I,O), not_aromatic(O), not_p(O),
   position(A,J,P), not_aromatic(P), small_or_polar(N), not_p(P),
   position(A,K,M), hydrophobic(M), not_k(M).

% RULE 13   TRAIN: 29/1 (97%acc,3%cov)    TEST: 4/1 (80%acc,2%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,N), not_aromatic(N),
   position(A,E,O), not_aromatic(O), small_or_polar(O), not_p(O),
                    not_k(O),
   position(A,F,R), lth(R,N),
   position(A,B,C), not_p(C), not_k(C),
   position(A,H,P), not_aromatic(P), not_p(P), lth(P,Q),
   position(A,I,L), hydrophobic(L), not_aromatic(L), small_or_polar(L),
   position(A,J,Q), not_aromatic(Q), not_p(Q), not_k(Q),
   position(A,K,M), hydrophobic(M).

% RULE 14   TRAIN: 45/4 (92%acc,5%cov)    TEST: 14/3 (82%acc,6%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,E,O), not_aromatic(O), not_p(O),
   position(A,F,P), small_or_polar(P), not_aromatic(P), not_p(P),
                    not_k(P),
   position(A,G,Q), not_aromatic(Q), not_k(Q),
   position(A,B,C), hydrophobic(C), neutral(C),
   position(A,H,L), hydrophobic(L), neutral(L),
   position(A,I,M), hydrophobic(M),
   position(A,J,N), neutral(N), not_p(N),
   position(A,K,R), not_aromatic(R), small_or_polar(R).

% RULE 15   TRAIN: 29/5 (85%acc,3%cov)    TEST: 4/1 (80%acc,2%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,P), not_k(P),
   position(A,E,Q), not_k(Q),
   position(A,F,O), not_aromatic(O), not_p(O),
   position(A,G,L), hydrophobic(L), small_or_polar(L), not_aromatic(L),
   position(A,B,C), polar(C), lth(C,S),
   position(A,H,S),
   position(A,I,R), not_k(R),
   position(A,J,N), neutral(N), not_p(N),
   position(A,K,M), hydrophobic(M), not_aromatic(M).

% RULE 16   TRAIN: 25/1 (96%acc,3%cov)    TEST: 4/1 (80%acc,2%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), octf(K,L,M,N,C,D,E,F,B),
   position(A,C,S), not_aromatic(S), not_p(S),
   position(A,D,U), small_or_polar(U), not_k(U),
   position(A,E,T), not_aromatic(T), not_p(T), not_k(T),
   position(A,F,V), not_p(V),
   position(A,B,O), not_p(O), not_k(O), ltv(O,R),
   position(A,G,P), very_hydrophobic(P), not_aromatic(P),
   position(A,H,Q), large(Q),
   position(A,I,R), small(R),
   position(A,J,N), not_p(N).

% RULE 17   TRAIN: 34/5 (87%acc,4%cov)    TEST: 4/1 (80%acc,2%cov)
```

```
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), octf(K,L,M,N,C,D,E,F,B),
   position(A,C,R), not_aromatic(R), not_p(R), not_k(R),
   position(A,D,S), not_aromatic(S), small_or_polar(S),
   position(A,E,O), very_hydrophobic(O), large(O),
   position(A,F,Q), neutral(Q), not_p(Q),
   position(A,G,P), very_hydrophobic(P), not_aromatic(P),

% RULE 18   TRAIN: 26/3 (90%acc,3%cov)    TEST: 4/0 (100%acc,2%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), octf(K,L,M,N,C,D,E,F,B),
   position(A,C,T), not_p(T), lth(T,R),
   position(A,D,S), not_aromatic(S),
   position(A,E,R), large(R), ltv(R,T),
   position(A,F,Q), neutral(Q), not_p(Q),
   position(A,B,O), polar(O), lth(O,S),
   position(A,G,P), hydrophobic(P), not_aromatic(P),

% RULE 19   TRAIN: 22/4 (85%acc,3%cov)    TEST: 5/1 (83%acc,2%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K),
   position(A,D,P), not_k(P), not_p(P),
   position(A,E,N), small_or_polar(N),
   position(A,F,R), lth(R,M),
   position(A,G,L), hydrophobic(L), small_or_polar(L),
   position(A,B,C), not_p(C),
   position(A,H,O), small_or_polar(O), not_k(O), not_p(O),
   position(A,I,M), hydrophobic(M), not_aromatic(M),
   position(A,J,S), ltv(S,Q),
   position(A,K,Q), not_k(Q).

RULE 20     TRAIN: 84/20 (81%acc,10%cov)  TEST: 18/4 (82%acc,8%cov)
alpha(A,B) :- alpha_pair3(B,D), alpha_triplet(D,E,F),
alpha_triplet(B,H,E),
   position(A,B,C), not_e(C), not_f(C), not_g(C), not_i(C), not_k(C),
                    not_m(C), not_n(C), not_p(C), not_q(C), not_r(C),
                    not_w(C), not_y(C),
   position(A,F,G),
   position(A,H,I), not_c(I), not_e(I), not_f(I), not_g(I), not_h(I),
                    not_i(I), not_k(I), not_m(I), not_n(I), not_p(I),
                    not_q(I), not_w(I), not_y(I),
   position(A,D,J), not_c(J), not_d(J), not_e(J), not_f(J), not_g(J),
                    not_k(J), not_l(K), not_m(J), not_p(J), not_r(J),
                    not_w(J),
   position(A,E,K), not_d(K), not_e(K), not_f(K), not_h(K), not_r(K),
                    not_y(K),

RULE 21     TRAIN: 72/18 (72%acc,8%cov)   TEST: 18/4 (82%acc,8%cov)
alpha(A,B) :- alpha_triplet(B,D,E), alpha_pair3(B,G),
   position(A,B,C), not_c(C), not_e(C), not_g(C), not_k(C), not_m(C),
                    not_n(C), not_p(C), not_q(C), not_r(C), not_y(C),
   position(A,D,F), not_c(F), not_d(F), not_f(F), not_g(F), not_h(F),
                    not_i(F), not_k(F), not_m(F), not_n(F), not_p(F),
                    not_q(F), not_w(F), not_y(F),
   position(A,G,H), not_c(H), not_d(H), not_f(H), not_h(H), not_l(H),
                    not_n(H), not_p(H), not_r(H), not_w(H), not_y(H),
   position(A,E,I), not_c(I), not_d(I), not_e(I), not_h(I), not_i(I),
                    not_l(I), not_m(I), not_n(I), not_p(I), not_r(I),
```

```
                    not_v(I), not_w(I), not_y(I).
% level 1 rules
% JOINT: TRAIN: 666/169 (80%acc,78%cov)    TEST: 169/42 (80%acc,83%cov)

% RULE 22 TRAIN: 509/92 (85%acc,60%cov)    TEST: 128/28 (82%acc,59%cov)
alpha(A,B) :- a1(A,B).

% RULE 23a  TRAIN: 299/52 (85%acc,35%cov) TEST: 83/10  (89%acc,38%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K), a1(A,F), a1(A,G).
% RULE 23b  TRAIN: 303/44 (87%acc,36%cov) TEST: 85/7  (92%acc,40%cov)
alpha(A,B) :- octf(D,E,F,G,B,H,I,J,K), a1(A,H), a1(A,I).

% RULE 24a  TRAIN: 183/10 (95%acc,22%cov) TEST: 53/2  (96%acc,24%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a1(A,F), a1(A,G), a1(A,H).
% RULE 24b  TRAIN: 189/5 (97%acc,22%cov)  TEST: 54/2  (96%acc,25%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a1(A,E), a1(A,F), a1(A,G).

% RULE 25   TRAIN: 102/2 (98%acc,12%cov)  TEST: 35/1  (97%acc,16%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J),
   a1(A,E), a1(A,F), a1(A,H), a1(A,I).

% RULE 26a  TRAIN: 102/3 (98%acc,12%cov)  TEST: 36/0  (100%acc,17%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a1(A,D), a1(A,E), a1(A,G),
a1(A,H).
% RULE 26b  TRAIN: 86/6 (93%acc,10%cov)   TEST: 33/0  (100%acc,15%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a1(A,C), a1(A,D), a1(A,G),
a1(A,H).
% RULE 26c  TRAIN: 88/5 (95%acc,10%cov)   TEST: 32/1  (97%acc,15%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a1(A,D), a1(A,E), a1(A,H),
a1(A,I).
% RULE 26d  TRAIN: 87/5 (95%acc,10%cov)   TEST: 32/1  (97%acc,15%cov)
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a1(A,E), a1(A,F), a1(A,I),
a1(A,J).

% level 2 rules
% JOINT TRAIN: 626/126 (83%acc,74%cov)    TEST: 160/24 (87%acc,74%cov)
RULE 27
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a2(A,B), a2(A,G), a2(A,H).
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J), a2(A,B), a2(A,E), a2(A,F).

% RULE 28
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J),
   a2(A,B), a2(A,D), a2(A,E), a2(A,F), a2(A,G), a2(A,H).
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J),
   a2(A,B), a2(A,G), a2(A,H), a2(A,I).
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J),
   a2(A,B), a2(A,D), a2(A,E), a2(A,F).
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J),
   a2(A,B), a2(A,F), a2(A,G), a2(A,H).
alpha(A,B) :- octf(C,D,E,F,B,G,H,I,J),
   a2(A,B), a2(A,E), a2(A,F), a2(A,G).
```

Testing the Applicability of ILP Systems

Eduardo Morales
The Turing Institute, 36 North Hanover Street, Glasgow G1 2AD, U.K.
email: eduardo@turing.ac.uk

Abstract

Due to the large hypothesis space, most ILP systems have applied such constraints that only very simple concepts have been induced with a limited background knowledge. An applicability improvement has been recently achieved by representing the background knowledge as ground unit clauses, allowing a simplier and more effective selection of relevant background knowledge. This paper presents an overview of ILP strategies to constrain the search space, and illustrates some problems in the applicability of recent ILP systems by looking at their performance over a simple concept in chess.

1 Introduction

Inductive Logic Programming (ILP) is a fast growing research area which combines Logic Programming with Machine Learning to induce first-order logic programs from examples [11]. In ILP, the system's current knowledge consists of data and background knowledge expressed as a logic program. The inductive problem is to find a hypothesis, a set of clauses, consistent with the current background knowledge and capable of explaining the data in the sense that all the positive literals but no negative literals in the data are deducible from the hypothesis and the background knowledge. That is, given background knowledge $\mathcal{K}$ and some examples $\mathcal{E}^+$ and $\mathcal{E}^-$, the induction process tries to find a hypothesis $\mathcal{H}$ for which $\mathcal{K} \wedge \mathcal{H} \vdash \mathcal{E}^+$ and $\mathcal{K} \wedge \mathcal{H} \nvdash \mathcal{E}^-$. These conditions define a search space on hypotheses. For induction to take place efficiently it is often necessary to structure the hypothesis space. This is usually done with a model of generalisation to organise the search space into hierarchies, where all the clauses below (above) a particular clause are specialisations (generalisations) of that clause. Roughly, a clause C_1 is more general than clause C_2 if in any world C_1 can show the same results as C_2. Induction can then be achieved by searching through those clauses more general to a known specialisation of a clause, or through more specific clauses than a known generalisation. In general, there are infinite ascending and descending chains within the hierarchy, constructed by conjoining and disjoining formulae, and additional constraints are required to limit the search space.

Section 2 looks at the different search strategies used over the hypothesis space, the most common constraints used to limit this space, the different example presentations, and the effect of the characteristics of the background knowledge to the applicability of ILP systems. Section 3 illustrates some of the problems of current ILP systems with a simple example in chess.

2 A Review of ILP

2.1 The Search Strategy

ILP systems which search a generalisation hierarchy (or bottom-up) include, Marvin [19], Cigol [13], Relex [5], Golem [14], Itou [18], Clint [3, 2], and Pal [8, 9]. Bottom-up systems start with a very specialised clause and gradually generalise it, turning constants into variables, removing conditions (literals from the body of the clause), or transforming the body of the clause using background knowledge. In general, an infinite number of facts deduced from the theory can be used to construct "very specialised" definitions, and even with finite length hypotheses, there is a large number of ways in which to generalise them.

Systems which search a specialisation hierarchy (or top-down), include, MIS [20], Foil [17], Linus [6], Focl [15], etc. Top-down systems start with a very general clause and gradually specialise it, replacing variables with terms, adding literals to the body, or transforming the body using background knowledge. Similarly, there is a very large number of ways in which to specialised a clause.

Both approaches suffer from a combinatorial explosion in the search for hypotheses. Most systems trade efficiency for applicability by applying strong restrictions to limit the hypothesis space.

2.2 Constraints on the Hypothesis Space

Structuring the hypothesis space with a generalisation/specialisation hierarchy provides only a guideline to ILP systems and different constraints have been used to restrict the space to achieve practical results.

- *Functional Restriction*: The system is provideded with information which states what particular arguments can be fixed (output arguments) in a predicate, if the rest are known (input arguments). This information can be used to form directed graphs which link input/output arguments and guide the construction of hypotheses [14, 17, 18, 6]. The linkage between input and output arguments can be constrained as well by using typed variables, where only input/output links can be formed between arguments with the same type.

- *Variable Connection Restriction*: Consider only clauses in which all variables appear at least twice in the clause [14, 18, 3, 2, 8, 9] or introduce a new literal to the body only if at least an existing variable is used [17].

- *Rule Class Restriction*: Construct hypothesis only from a class of clauses. This can be defined through rule models (i.e., consider only hypotheses which "match" a particular rule model [10, 21, 23, 8, 9] or integrity constraints [2]) or with particular "refinement" operators as in MIS [20].

- *Information Content Restriction*: Construct hypotheses which produces a compression in information content [13] or guide the hypothesis search with a measure of information gain based on the discrimination between positive and negative examples [17, 6].

- *Initial Clause Restriction*: Provide an initial clause and add only literals to the body which can be deduced from the body of the clause and the background knowledge [19, 18].

The above constraints helps to reduce the search space and guide the learning process, however, the hypothesis space depends mainly on the examples and the background knowledge provided to the system. The following Sections look at the different example presentations, the background knowledge, and their effect on the induction process.

2.3 Example Presentation: User vs Batch vs Automatic

It is generally believed that a "careful" experiment selection is more effective for concept formation than a random experiment selection, and in general, the learning performance of an inference method changes with the example presentation. Examples can be provided by the environment, selected by an informed oracle, can be a subset of the example space, randomly generated, automatically generated by the system, etc. The most common example presentations are, either interactively, by the user or an example generator, or in 'batch', selected from an existing sample set (often generated by the user). In general, the termination criterion can be linked to the example presentation.

- Systems which rely for their success, on a careful presentation of examples by the user, include MIS [20], Cigol [13], Itou [18], Marvin [19], etc. The user provides the "right" examples and guides the learning process. The user is aware of the target concept and determines the criterion of success. Unfortunately, in some of such cases, the system will fail to produce the required concept if it is provided with another selection of examples or even with the same examples but in a different sequence. Choosing appropriate example selections is not always easy and calls into question the learning capabilities of systems which depends on them.

- Systems which accept a set of examples in batch, include Golem [14], Foil [17], Linus [6], etc. The termination criteria is based on completeness and correctness (with perhaps some exceptions) of the hypotheses. The user often performs an "artificial" generation of examples, as not all the domains have them readily available for the process. It is difficult to know in advance if the selection of examples is suitable for the task or if additional examples will be required to correct the hypotheses produced by the system. This can involve a process of selecting examples, inducing, and testing the hypotheses until the required results are obtained.

- Systems which generate their own examples, include Clint [3], Relex [5], Pal [8, 9], etc. The termination criteria is either given by the user or when the system is unable to generate an incorrectness or incompleteness with a new example. Experimentation (or active instance selection) has been employed in several machine learning systems [5, 1, 4, 7, 16, 8, 9] to reduce the dependency on the user and guide effectively the learning process. Most automatic example generators are either too domain specific or have problems on domains with a large example space.

2.4 Background Knowledge: Clauses vs Facts

In ILP, the background knowledge (generally represented as Horn clauses) is carefully selected by the user. Recent systems (i.e., Foil, Focl, Linus, and Golem), have represented their background knowledge with a set of ground unit clauses. Both approaches are analysed below.

- Most ILP systems which represent their background knowledge as non-ground clauses, suffer from search problems, need a careful selection of background knowledge definitions (and sometimes of examples), and have been applied to very restricted domains. Their main problem arises from the combinatorial explosion in the search for hypotheses, which is severely affected by the size and characteristics of the background knowledge. The selection of relevant background knowledge in the construction of hypotheses, often involves some form of derivation process, which in the presence of recursive definitions needs to be bounded... By carefully choosing a limited background knowledge, efficiency can obtained but only at the expense of applicability... Their main advantages rely on its concise representation, they can incorporate in principle any previously known concepts without requiring any transformation process, and the new learned concepts are immediately accessible to the system in the next inductive cycle.

- In trying to improve applicability, recent systems have replaced the representation of the background knowledge with a set of ground facts, where an efficient indexing mechanism can be incorporated and an easy selection of relevant background knowledge can be performed. This approach has been used in larger domains [11] as it reduces the combinatorial explosion in the search of hypotheses. Ground theory systems require a careful selection of a "representative" subset of ground facts in advance, which is often tailored to the nature of the examples over which the induction is made. A huge memory space is sometimes needed to store ground theories and none of the current systems is used in an incremental way.

There is a fundamental space-time tradeoff when choosing a particular background knowledge representation. Systems which use ground theories are limited by the number of facts that they can effectively process and usually require a careful selection of them. In general, the generation of appropriate background facts can be a time-consuming process, highly dependent on the examples. This memory-greedy and time-consuming generation process is compensated by an efficiency gain over systems which use non-ground theories, it constraints the search for hypothesis and facilitates the selection of background knowledge, which has allow it to be applied to larger domains. Regardless of the approach, problems can only be magnified when adding extra background knowledge and in general only a very limited number of background concepts have been used by ILP systems.

In the next Section, the applicability of recent ILP systems is analysed over a simple concept in chess where a large background knowledge can be used to induced concepts.

3 Chess as a Test Domain for ILP Systems

Chess has been an interesting and challenging domain for concept learning. Evidence suggests that high performance of concept learning systems for this domain, depends on the ability to represent relational concepts [12]. Most ILP systems have been used to infer very simple concepts, such as list definitions like *member*, *append*, etc., the concept of an *arch*, *family relations*, etc., with very restricted background knowledge (e.g. [20, 19, 13, 3, 18]). By contrast, concepts in chess can involve background definitions for threats, checks, legal moves, distances to places, etc. These concepts comprise a more realistic background knowledge with a larger hypothesis space from which relatively varied concepts can be induced.

Our interest in learning concepts chess has resulted in the design of Pal, an ILP system capable of learning concepts in chess [8, 9]. Pal was designed to learn concepts around the idea of patterns from which only a limited number of facts are derived from the background knowledge. A reasonable large number of background definitions can be used, which contrasts with most non-ground theory systems which get lost in the complexities of the search, when additional background knowledge is used.

Recently, systems have used a finite set of facts as background knowledge to improve efficiency and increase their applicability. Within this approach, probably the best known state-of-the-art systems are Golem [14] and Foil [17]. Both systems were tested over a simple concept which can be used in chess[1].

3.1 A Simple Example

In domains like chess, some concepts require a very large number of facts to be completely described[2] an a careful selection of them needs to be done to maintain efficiency. This can be a time-consuming process, and it is not always easy to know what to generate unless the nature of the examples are known in advance. Even with a limited set of background facts, Golem and Foil can have problems in domains which are non-deterministic by nature and where a large example space exists. Both systems have been used to learn the concept of illegal positions in a white to move, white king and rook against king endgame. In order simplify the test, a similar representation used in the concept of illegality was used with additional background knowledge. The target concept, called *diagonal*, represents those positions where three pieces are in a diagonal line. The basic idea behind this concept can be used to learn the concept of *pin*, *skewer*, *discovery checks*, etc. In addition to the concept of *less_than/2*, the definition of *line/4* was provided as background knowledge to represent any two positions in a diagonal, vertical or horizontal line. That is, *line(X1,Y1,X2,Y2)* means that position $\langle X1, Y1\rangle$ and position $\langle X2, Y2\rangle$ are in a straight line, while *less_than(N1,N2)* means that *N1* is less than *N2*. For ground-theory systems, this represents 1,456 facts for *line/4* and 28 facts for *less_than/2*. To simplify the concept, only positions where the first piece $\langle X1, Y1\rangle$ was in the lower left corner, the second piece $\langle X2, Y2\rangle$ in the middle, and the third piece $\langle X3, Y3\rangle$ in the upper right corner, were considered as positive. This makes a total of 196 possible positive examples out of 64^3 (262,144) examples in the example space. With this background knowledge, the target definition can be defined as follows (see Figure 1):

```
diagonal(X1,Y1,X2,Y2,X3,Y3) ←
    less_than(X1,X2), less_than(X1,X3), less_than(X2,X3),
    less_than(Y1,Y2), less_than(Y1,Y3), less_than(Y2,Y3),
    line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3), line(X2,Y2,X3,Y3).
```

- Pal was tried on the same concept. Pal requires the examples to be given as descriptions of chess positions and unlike other systems, the exact arguments of the goal predicate are not specified in advance. Pal also requires to know the domain of the arguments that are used to describe examples to automatically generate its own

[1] The latest public versions of Golem and Foil (Foil2, an improved version of Foil) were used in the test.

[2] A concept of *threat* between any two pieces in chess able to distinguish each piece, side and place, requires more than 20,000 facts to be completely described.

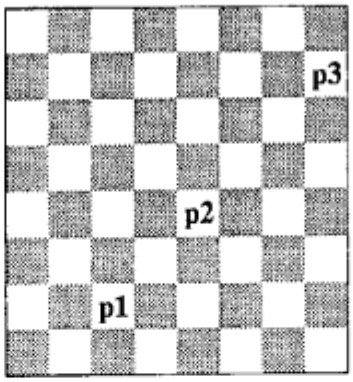


Figure 1: Three Pieces in Diagonal

examples. The example positions were described as lists of *piece(P,F,R)* predicates, where F and R are the file and rank of a piece, and P represents the particular piece. Since the piece and side are not important to this test, the domain for P was [*p1, p2, p3*] (the initial example given to Pal is shown in Figure 1). The domain for R and F is [1,2,...,8]. Pal was provided with the background definitions for *line/4* and *less_than/2*. The example generator only considered changes in the positions of the 3 pieces. Pal arrived to the following definition after generating 3 + and 135 − examples.

```
diagonal(p1,X1,Y1,p2,X2,Y2,p3,X3,Y3) ←
    piece(p1,X1,Y1), piece(p2,X2,Y2), piece(p3,X3,Y3),
    less_than(X1,X2), less_than(X1,X3), less_than(X2,X3),
    less_than(Y1,Y2), less_than(Y1,Y3), less_than(Y2,Y3),
    line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3), line(X2,Y2,X3,Y3).
```

- Foil was tested under the same conditions. The complete background facts for *line/4* and *less_than/2* were provided as background knowledge. Different positive and negative examples were given to Foil to try to learn this concept (the outputs produced by Foil are also included). Since no negative literals are expected in the final definition, and in order to simplify Foil's search, they were not considered in the test[3].

  - Test1: With the same examples used by Pal (i.e., 4 + and 135 −), Foil is not able to produce any definition. Presumably because of the relatively few number of positive examples in comparison with the number of negative examples.
  - Test2: Foil was tested with all the possible positive examples of the target concept (196 +) and the same negatives examples used by Pal (135 −). With them, Foil produced the following incorrect definition:

```
diagonal(X1,Y1,X2,Y2,X3,Y3) ←
    line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3), line(X2,Y2,X3,Y3),
    line(X1,X3,Y1,Y3), less_than(X1,X3).
diagonal(X1,Y1,X2,Y2,X3,Y3) ←
    =(X1,Y1), =(X2,Y2).
```

[3]Foil's tests were carried out with the following parameters: foil2 -n -g0 < infile > outfile (i.e., no negative literals and no determination).

This is a counter example: diagonal(3,5,4,4,5,3).

- Tests: Negative examples were incrementally added to Foil to try to learn the correct definition. Each time an incorrect hypothesis was produced, new negative examples were given to try to correct it. Foil changed several times its hypothesis (none of which was correct). For instance, the following definition was produced by Foil with all the positive examples and the same negative examples required by Golem to produce the correct definition (i.e., 195 + and 159 −, see below).

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3), line(X2,Y2,X3,Y3),
  less_than(X1,X3), less_than(Y1,Y3).

  A counter example: diagonal(1,1,3,1,2,2).

- Final Test: Foil eventually learned the target concept when all the positive examples were given and with 179 carefully selected negative examples. Foil's definition is as follows:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  less_than(X1,X2), less_than(X1,X3),
  less_than(Y1,Y2), less_than(Y2,Y3),
  line(X1,Y1,X2,Y2), line(X1,Y1,X3,Y3).

Although Foil is able to learn the above concept, it was not until a long process of several interacting sessions of analysing why Foil failed and providing new examples to contradict Foil's hypotheses. This example illustrates some of its problems previously mentioned in Section 2. Namely, problems of preparing the *right* data (background facts as well as examples), and problems with its information gain heuristic.

- Similar tests were run with Golem. However, the background definition of *line/4* is non-deterministic (i.e., the same inputs can produce different outputs). Golem cannot learn concepts which use non-deterministic background knowledge (i.e., it cannot learn *diagonal* with *line/4*). The way to go around this problem in Golem, is to design some deterministic background knowledge that could be used to define an "equivalent" definition. Instead of *line/4*, the complete background facts for *abs_diff/3* (absolute difference between two numbers) was provided as background knowledge. That is, *abs_diff(N1,N2,Diff)* means that the absolute difference between *N1* and *N2* is *Diff*. This represents a total of 64 background facts. With this new background knowledge, we expect Golem to arrive to the following equivalent definition:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
  abs_diff(X2,X3,D3), abs_diff(Y2,Y3,D3),
  less_than(X1,X2),less_than(X1,X3), less_than(X2,X3),
  less_than(Y1,Y2),less_than(Y1,Y3), less_than(Y2,Y3).

Similar to Foil, Golem was tested with a different number of examples.

- Test1: With the same examples required by Pal, Golem produces the following definition:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
  less_than(X1,X2), less_than(X1,Y3).

  This is a counter example: diagonal(1,3,2,2,3,1).

- Test2: Golem was then given all the possible positive examples (196 +) and the same negative examples used by Pal (135 −). Golem produces the following (still incorrect) definition:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
  less_than(X1,X2), less_than(X2,X3).

  Counter example: diagonal(2,3,3,2,4,1).

- Tests: Negative examples were incrementally given to try to obtain the solution. Some of its intermediate hypotheses include (with 196 +, 150 −):

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X2,Y3,D2), abs_diff(Y2,Y3,D2),
  abs_diff(X1,Y1,D3), abs_diff(X2,Y2,D3), abs_diff(X3,Y3,D3),
  less_than(X1,X2), less_than(X2,X3),

  Counter example: diagonal(4,3,5,6,6,5).

- Final Test: Golem was able to produce a correct definition with all the positive example (196 +) and with 159 negative examples. Golem's definition is as follows:

  diagonal(X1,Y1,X2,Y2,X3,Y3) ←
  abs_diff(X1,X2,D1), abs_diff(Y1,Y2,D1),
  abs_diff(X1,X3,D2), abs_diff(Y1,Y3,D2),
  abs_diff(X2,Y3,D3), abs_diff(Y2,Y3,D3),
  abs_diff(X1,Y1,D4), abs_diff(X2,Y2,D4), abs_diff(X3,Y3,D4),
  less_than(X1,X2), less_than(X2,X3).

  Golem is able to learn, at least an equivalent definition of the target concept. However, due to its inability to learn non-deterministic concepts, the background knowledge needs to be modified in such a way, that the resulting definition can become obscure to the user. Similar to Foil, Golem retains the burdensome of preparing the background facts.

In this Section, Foil and Golem are tried on a particular chess concept which involves a relatively simple background knowledge. "In principle", both systems could be used to induce other concepts in chess, however the example illustrates some of their main problems. Both systems suffer from the problem of preparing the background facts. In chess, some concept definitions can involve concepts of legal moves, threats, checks, etc. Defining background facts for such concepts is a time consuming and difficult process.

Specially since it is sometimes unworkable for the systems to include all the background facts, even if they are finite. In such cases, appropriate subsets need to be selected to maintain efficiency, which requires a prior knowledge of the example sample.

In addition to this, Foil's construction of hypotheses is heuristically guided by its information gain measure. This measure is affected by the number of positive and negative examples in the training set. As any greedy search algorithm, Foil is prone to make local optimal but global undesirable choices. A new implementation of Foil (Foil2, which was used in the test) incorporates *checkpoints*, that is, points where two or more alternatives appear to be roughly equal. If the greedy search fails, the system reverts to the most recent checkpoint and continues with the next alternative. This however, does not eliminate the need to carefully choose the training set to ensure that the desired literal is included in the definition. In domains where the example space can be very large, as with many chess concepts, trying to select an adequate subset for the required generalisation, is not an easy task and often requires an interactive process of analysing the system failures and adding new examples to try to correct them.

Golem, in addition to the preparation of the background facts, is limited to learn deterministic clauses. Most concepts in chess are, or at least involve some, non-deterministic concepts. Finding a deterministic counterpart, to intuitively non-deterministic concepts, is not always easy/possible to do, and sometimes can only be made in terms of such "opaque" concepts, that the found solution is not longer transparent to the user. Even after finding a way around learning a non-deterministic concept with deterministic background definitions, the resulting definition, being non-deterministic, cannot be used to learn in the future.

4 Conclusions

The combinatorial explosion in the search for hypotheses in first-order inductive systems have introduced several constraints to control this search, among others a very restricted background knowledge applicable to a limited number of concepts. More recent systems have used ground theories to improve applicability. However, they are still restricted by the "complexity" of the background knowledge. In a domain like chess, we would like to use legal moves, threats, distances between pieces, etc. as background knowledge to learn several chess concepts. However, this is not so easy to do with ground theory systems, as a subset of the possible ground facts, tailored to the examples, needs to be carefully selected. Using non-ground knowledge with appropriate constraints has been successfully used in chess to learn concepts like forks, threats, skewers, pins, discovery-attacks, etc., with several background knowledge definitions [8; 9]. The inadequacies of recent systems which use ground theories over a domain like chess, suggest that an effective use of larger background knowledge over large example spaces, common to more realistic domains, may well depend on the use of suitable constraints over non-ground theories.

References

[1] J. G. Carbonell and Y. Gil. Learning by experimentation. In Langley P, editor, *Proceedings of the fourth international workshop on machine learning. 22-25 June 1987. Irvine, California.*, pages 256 – 266, Los Altos, CA, 1987. Morgan Kaufmann Publishers.

[2] L. de Raedt and M. Bruynoghe. Indirect relevance and bias in inductive concept-learning. *Knowledge Acquisition*, 2(4, December):365 – 390, 1990.

[3] L. de Raedt and M. Bruynooghe. On interactive concept-learning and assimilation. In Sleeman D and Richmond J, editors, *EWSL 88: proceedings of the third European Working Session on Learning. The Turing Institute, Glasgow. 3-5 October, 1988.*, pages 167 – 176, London, 1988. Pitman.

[4] T. G. Dietterich and B. G. Buchanan. The role of experimentation in theory formation. In *ML-83: proceedings of the international machine learning workshop, June 22 - 24, 1983. Monticello, Illinois*, pages 147 – 155, Urbana, IL, 1983. University of Illinois, Department of Computer Science.

[5] C. Feng. *Learning by Experimentation.* PhD thesis, The Turing Institute - University of Strathclyde, 1990.

[6] N. Lavrac, S. Dzeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with linus. In Kodratoff Y, editor, *EWSL '91: machine learning: proceedings of the European working session on learning. Porto, Portugal. March 6-8, 1991. [ISBN: 3 54053816 X].*, pages 265 – 281, Berlin, 1991. Springer-Verlag.

[7] D. B. Lenat. AM: an artificial intelligence approach to discovery in mathematics as heuristic search. Technical Report (AIM-286 ; STAN-CS-76-570), Stanford University, Artificial Intelligence Laboratory, Stanford, CA, 1976.

[8] E. Morales. Learning features by experimentation in chess. In Kodratoff Y, editor, *Proceedings of the European working session on learning. EWSL-91*, pages 494 – 551, Porto, Portugal, 1984. Springer-Verlag.

[9] E. Morales. Learning Chess Patterns. In S. Muggleton, editor, *ILP-91: proceedings of the international workshop of inductive logic programming*, pages 291 – 307, Viana de Castelo, Portugal, 1991.

[10] K. Morik. Sloppy modeling. In Morik K, editor, *Knowledge representation and organization in machine learning*, pages 107 – 134. Springer-Verlag, Berlin, 1989.

[11] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8:295 – 318, 1991.

[12] S. Muggleton, M. Bain, J. Hayes-Michie, and D. Michie. An experimental comparison of human and machine learning formalisms. In Spatz B, editor, *Proceedings of the sixth international workshop on machine learning. Cornell University, Ithaca, New York. June 26-27, 1989. [ISBN: 1 55860036 1].*, pages 113 – 118, San Mateo, CA, 1989. Morgan Kaufmann.

[13] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Machine Learning-88. Ann Arbor, Michigan. June 1988.*, page 14, 1988.

[14] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In *ALT-90, Proceedings of the conference of algorithmic learning theory*, Tokyo, Japan, 1990. Ohmsha.

[15] M. Pazzani and D. Kibler. The Utility of Knowledge in Inductive Learning. Technical Report 90-18, University of California, Department of Information and Computing Science, Irvine, CA, 1990.

[16] B. W. Porter and D. F. Kibler. Experimental goal regression: A method for learning problem-solving heuristics. *Machine Learning*, (1):249 – 286, 1986.

[17] J. R. Quinlan. Learning logical definitions from relations. *Machine learning*, 5((3), August):239 – 266, 1990.

[18] C. Rouveurol. ITOU: Induction of first order theories. In S. Muggleton, editor, *ILP-91: proceedings of the international workshop of inductive logic programming*, pages 127 – 157, Viana de Castelo, Portugal, 1991.

[19] C. Sammut and R. B. Banerji. Learning concepts by asking questions. In Michalski R S, Carbonell J G, and Mitchell R M, editors, *Machine learning: an artificial intelligence approach [Volume 2]*, pages 167 – 191. Kaufmann, 1986.

[20] E. Y. Shapiro. Inductive inference of theories from facts. Technical Report (Research Report 192), Yale University, Department of Computer Science, New Haven, CT, 1981.

[21] S. Thieme. The acquisition of model-knowledge for a model-driven machine learning approach. In Morik K, editor, *Knowledge representation and organization in machine learning*, pages 177 – 191. Springer-Verlag, Berlin, 1989.

[22] P. H. Winston. Learning structural descriptions from examples. In Brachman R J Levesque H J, editor, *Readings in knowledge representation.*, pages 141–168. Kaufmann, Los Altos, Ca, 1985.

[23] S. Wrobel. Demand-driven concept formation. In Morik K, editor, *Knowledge representation and organization in machine learning*, pages 289 – 319. Springer-Verlag, Berlin, 1989.

Automatically Constructing Control Systems by Observing Human Behaviour

Claude Sammut

School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
claude@spectrum.cs.unsw.oz.au

Abstract

We describe experiments to devise machine learning methods for the construction of control systems by observing how humans perform control tasks. The present technique uses a propositional learning system to discover rules for flying an aircraft in a flight simulation program. We discuss the problems encountered and present them as a challenge for researchers in Inductive Logic Programming. Overcoming these problems will require ILP methods that go beyond our current knowledge, including induction over noisy numeric domains, dealing with time and causality and complex predicate invention.

1. Learning Control Rules

Almost all applications of inductive learning, so far, have been in classification tasks such as medical diagnosis. For example, medical records of patients symptoms and accompanying diagnoses made by physicians are entered into an induction program which constructs rules that will automatically diagnose new patients on the basis of the previous data. The output is a classification. We are interested in automatically building control rules that output an action. That is, when a state of a dynamic system arises that requires some corrective action, the rules should be able to recognise the state and output the appropriate action. Just as diagnostic rules can be learned by observing a physician at work, we should be able to learn how to control a system by watching a human operator at work. In this case, the data provided to the induction program are logs of the actions taken by the operator in response to changes in the system.

In a preliminary study (Sammut, Hurst, Kedzier and Michie, 1992), we have been able to synthesise rules for flying an aircraft in a flight simulator. The rules are able to make the plane take off, fly to a specified height and distance from the runway, turn around and land safely on the runway. While control systems have been the subject of much research in machine learning in recent years, we know of few attempts to learn control rules by observing human behaviour. Michie, Bain and Hayes-Michie (1990) used an induction program to learn rules for balancing a pole (in simulation) and earlier

work by Donaldson (1960), Widrow and Smith (1964) and Chambers and Michie (1969) demonstrated the feasibility of learning by imitation, also for pole-balancing. To our knowledge, the autopilot described here is the most complex control system constructed by machine learning methods. However, there are still many research issues to be investigated and they are the subject of this paper. The main problems we discuss are listed below.

> The difference between learning classifications and learning actions is that the learning algorithm must recognise that actions are performed in response to, and result in, changes in the system being controlled. Classification algorithms only deal with static data and do not have to cope with temporal and causal relations.
>
> In our preliminary study we were able to demonstrate the feasibility of learning a specific control task. The next challenge is to build a generalised method that can learn basic skills that can be used in a variety of tasks. These skills become building blocks that can be assembled into a complete new controller to meet the demands of a specified task.
>
> One of the limitations we have encountered with existing learning algorithms is that they can only use the primitive attributes supplied in the data. This results in control rules that cannot be understood by a human expert. Constructive induction (or predicate invention) may be necessary to build higher-level attributes that simplify the rules.

We believe it is important that machine learning research should be directed towards acquiring control knowledge since this will give us a way of describing human subcognitive skills and it will result in useful engineering tools.

One of the outstanding problems our research addresses is that subcognitive skills are inaccessible to introspection. For example, if you are asked by what method you ride a bicycle, you will not be able to provide an adequate answer because that skill has been learned and is executed at a subconscious level. By monitoring the performance of a subcognitive skill, we are able to construct a functional description of that skill in the form of symbolic rules. This not only reveals the nature of the skill but also may be used as an aid to training since the student can be explicitly shown what he or she is doing.

Learning control rules by induction provides a new way of building complex control systems quickly and easily. For example, the need in aerospace for pilots to control airplanes close to the margin of instability is putting increasing pressure on present techniques both of pilot training and of flight automation. We claim that it will be possible to build a pilots assistant using inductive methods. A control engineer is only able to supply automated modules, such as autolanders, provided that envisaged meteorological or other conditions are not too abnormal. There are specialised manoeuvres that the pilot would be relieved to see encapsulated into an automated sub-task, but which cannot, for reasons of complexity and unpredictability, be tackled with standard control-theoretic tools. Yet they can be tackled, often at the expense of

effectiveness or safety, by a trained pilots skills that have been acquired by practice but which the pilot cannot explain. Control engineers and programmers, much as they might wish to, at present have no way to capture these procedures so as to solve the flight automation problem. In this context, the industry requires a convenient, and not too expensive, means of automatically constructing models of individual piloting skills.

While our experiments have been primarily concerned with flight automation, inductive methods can be applied to a wide range of related problems. For example, an anaesthetist can be seen as controlling a patient in an operating theatre in much the same way as a pilot controls an aircraft. The anaesthetist monitors the patients condition just as a pilot monitors the aircrafts instruments. The anaesthetist changes dosages of drugs and gases to alter the state of a system (the patient) in the same way that a pilot alters thrust and attitude to control the state of a system (the aircraft). A flight plan can be divided into stages where different control strategies are required, eg. take-off, straight and level flight, landing, etc. So too, the administration of anaesthetics can be divided into stages: putting the patient to sleep, maintaining a steady state during the operation and revival after the procedure has been completed.

In the next section, we will describe our preliminary experiments using a decision tree induction program. While we were able to meet our initial goals, we believe that we are reaching the limits of the descriptive power of propositional learning algorithms and will have to a first-order system. Unfortunately, no existing Inductive Logic Programming algorithm is suitable for use in control applications. Section 3 describes some of the problems that we face and section 4 suggests a number of avenues of research for ILP.

2. Preliminary Study

This section provides a brief description of our preliminary study into constructing rules for an autopilot by logging the flights of human pilots. The reader is referred to (Sammut, Hurst, Kedzier and Michie, 1992) for more detail. The source code to a flight simulation program was made available to us by Silicon Graphics Incorporated (SGI). Our task was to log actions taken by pilots during a number of flights on the simulator. These logs were then used to construct, by induction, a set of rules that could fly the aircraft through the same flight plan that the pilots flew. The results presented below are derived from the logs of three subjects who each flew 30 times. We will refer to the performance of a control action as an event. During a flight, up to 1,000 events can be recorded. With three pilots and 30 flights each the complete data set consists of about 90,000 events. An autopilot has been constructed for each of the three subjects. Each pilot is treated separately because different pilots can fly the same flight plan in different ways.

The central control mechanism of the simulator is a loop that interrogates the aircraft controls and updates the state of the simulation according to a set of equations of motion. Before repeating the loop, the instruments in the display are updated. The display update has been modified so that when the pilot performs a control action by

moving the mouse or changing the thrust or flaps settings, the action and the state of the simulation are written to a log file. The data recorded are:

on_ground	boolean:	is the plane on the ground?
g_limit	boolean:	have we exceeded the planes g limit
wing_stall	boolean:	has the plane stalled?
twist	integer:	0 to 360° (in tenths of a degree, anti-clockwise)
elevation	integer:	0 to 360° (in tenths of a degree, anti-clockwise)
azimuth	integer:	0 to 360° (in tenths of a degree, anti-clockwise)
roll_speed	integer:	0 to 360° (in tenths of a degree per second)
elevation_speed	integer:	0 to 360° (in tenths of a degree per second)
azimuth_speed	integer:	0 to 360° (in tenths of a degree per second)
airspeed	integer:	(in knots)
climbspeed	integer:	(feet per second)
E/W distance	real:	E/W distance from centre of runway (in feet)
altitude	real:	(in feet)
N/S distance	real:	N/S distance from northern end of runway (in feet)
fuel	integer:	(in pounds)
rollers	real:	±4.3
elevator	real:	±3.0
rudder	real:	not used
thrust	integer:	0 to 100%
flaps	integer:	0°, 10° or 20°
spoilers	integer:	not relevant for a Cessna

Most of the attributes of an event are numeric, including real numbers, sub-ranges and circular measures. Since there can be an enormous amount of variation in the way pilots fly, the data are very noisy. Note also that the output value of induction is a control setting such as the position of the flaps, rollers or elevator. Thus, the output values are also required to be numeric.

At the start of a flight, the aircraft is pointing North, down the runway. The subject is required to fly a well-defined flight plan that consists of the following manoeuvres: take off and fly to an altitude of 2,000 feet; level out and fly to a distance of 32,000 feet from the starting point; turn right to a compass heading of approximately 330°; at a North/South distance of 42,000 feet; turn left to head back towards the runway; line up on the runway and descend; land on the runway.

The data from each flight were segmented into the stages listed above. For each stage we construct four separate decision trees for the elevator, rollers, thrust and flaps. The rudder is not used. A program filters the flight logs generating four input files for the induction program. The attributes of a training example are the flight parameters of the simulator, listed above. The dependent variable or class value is the attribute describing a control action. The reason for segmenting the data is that each stage requires a different manoeuvre. By combining all the data from all stages, we would be expecting the induction program to construct seven sets of rules for controlling the aircraft in each of the seven stages. This makes the programs task more difficult than is necessary since we have already defined the sub-tasks and have told the

human subjects what they are. It is reasonable that the learning program should have the same information as the pilots.

For the preliminary study, we used the decision tree induction program C4.5 (Quinlan, 1987). To test the induced rules, the original autopilot code in the simulator is replaced by the rules. A post-processor converts C4.5s decision trees into if-statements in C so that they can be incorporated into the flight simulator easily. Hand-crafted C code determines which stage the flight has reached and decides when to change stages. The appropriate rules for each stage are then selected in a switch statement. Each stage has four, independent if-statements, one for each action.

We demonstrate how these rules operate by describing the controllers for the first stage. The critical rule at take-off is the elevator rule:

```
elevation > 4 : level_pitch
elevation <= 4 :
|   airspeed <= 0 : level_pitch
|   airspeed > 0 : pitch_up_5
```

This states that as thrust is applied and the elevation is level, pull back on the stick until the elevation increases to 4°. Because the controls take some time to respond, the final elevation usually reaches 11°, which is close to the values obtained by the pilot. `pitch_up_5` indicates a large elevator action, whereas, `pitch_up_1` would indicate a gentle elevator action. The other significant control at this stage is flaps:

```
elevation <= 6 : full_flaps
elevation > 6 : no_flaps
```

Once the aircraft has reached an elevation angle of 6°, the flaps are raised.

The rules we have synthesised are successful in the sense that the plane follows the flight plan just as the human trainer would and lands safely on the runway. Because induction over a large set of data has an averaging effect, the autopilot actually flies more smoothly than the trainer. Figure 1 shows a profile of the trainers flight, plotting the E/W distance travelled as a function of the N/S distance away from the runway. Each point represents an action being taken by the pilot. This flight can be compared with the autopilots flight shown in Figure 2. A similar comparison can be made between the altitude profiles for the trainer and the autopilot, shown in figures 3 and 4, respectively.

Figure 2: Cross-range Profile for Autopilot

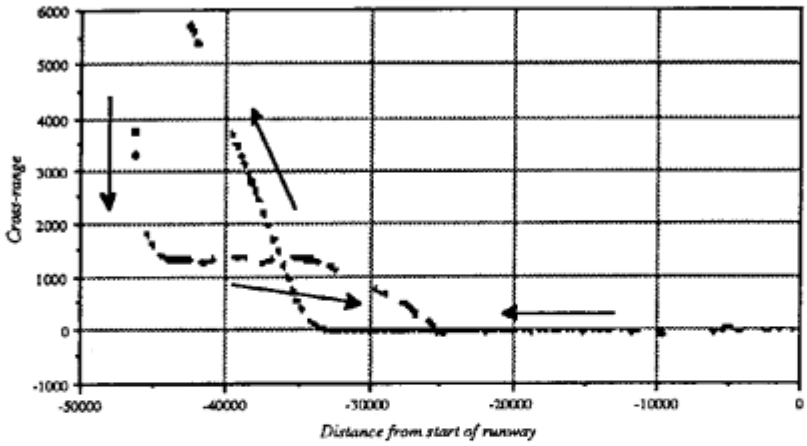


Figure 3: Altitude Profile for Trainer

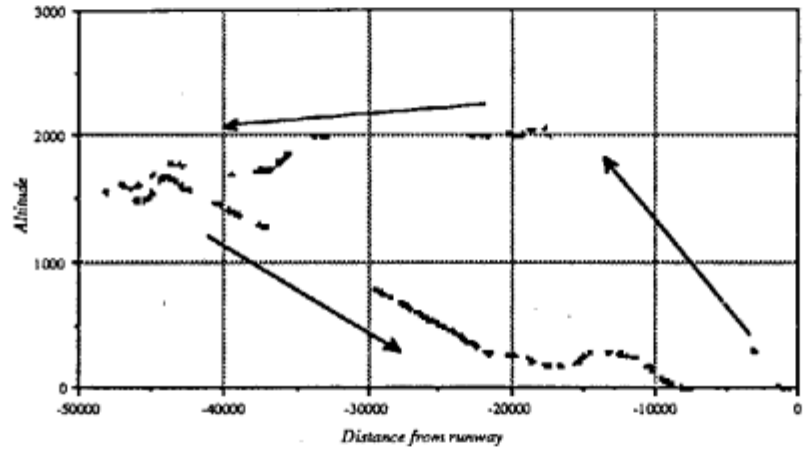
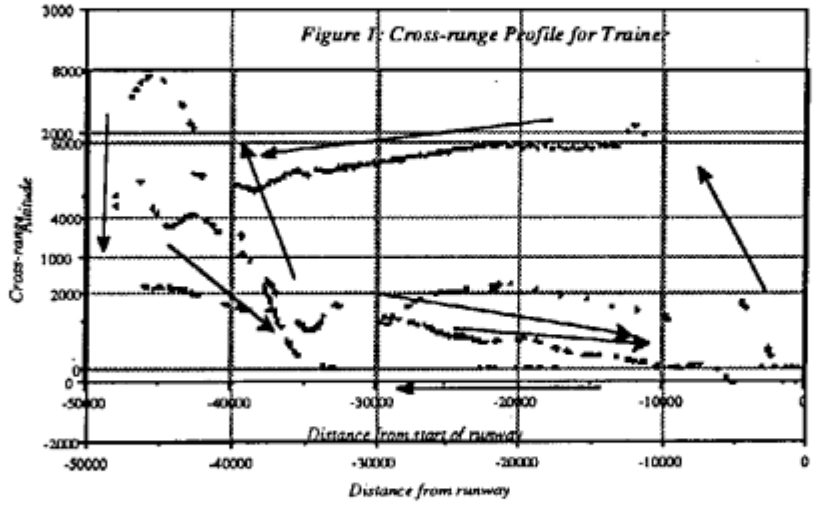


Figure 4: Altitude Profile for Autopilot



It is important to note that this study was not merely an exercise in collecting data and applying an existing program. The output of an induction program is only as good as its input. Before obtaining a successful outcome it was necessary to learn how best to collect and analyse the data and while the study has demonstrated the possibility of building control systems by induction, it has also revealed many problems still to be solved. We discuss these problems in the next section and our proposed solutions after that.

3. Problems

While our work as been concentrated on the domain of flight automation because it provides us with a complex, realistic task that ensures our methods remain practical, we are mindful of the importance of keeping our methods sufficiently general that they can be used in other domains. Thus, while the issues we discuss below are described in terms of the flight simulator, the reader should keep in mind that similar problems exist in almost all control tasks. The challenge posed in this paper is, Can Inductive Logic Programming methods be useful in solving these problems and how must they be extended in order to handle control tasks?

3.1 Causality

For our preliminary study, we used an induction program that was designed for learning classifications, not control actions. This program, and others of its kind, produced flight rules such as:

```
if speed > 130 knots
then thrust = 100%
```

This accurately summarises the data since high speed is usually due to high thrust. Unfortunately, it is not a useful control rule since it tells us how we got to a high speed but not what to do now that were there. The induction algorithm has no way of recognising a causal relationship between thrust and speed.

There has been some work in trying to extract causal relations from data (Bratko, Muggleton & Varsek, 1991) but this can be quite difficult when there are many variables and actions involved. However, the task becomes easier when the program is provided with background knowledge about causality.

3.2 What does the pilot see?

We do not record the same information that the pilot uses. For example, when landing, a pilot usually chooses an aiming point on the runway and stays on a steady glide path by maintaining a direction toward the aiming point. The aiming point and deviation from it are not recorded since these are attributes chosen by the pilot and are not part of the instrumentation of the aircraft. As a result, rules become more complicated than they should be because they must effectively perform a coordinate transformation between the recorded parameters and the pilots parameters. To avoid this problem, the

induction program would have to reconstruct the information that the pilot is using from the data and generate new attributes that better reflect the pilots goals.

3.3 Learning General Skills

The present method only learns how to fly a specific flight plan and so is very limited. We should be able to learn basic skills that can be used as building blocks for a complete flight. So instead of learning a flight plan, the system should learn rules for sub-tasks such as, turn left through X degrees or climb to altitude H. Once building blocks have been learned, it should be possible to use well-known planning algorithms to put them together into a complete flight plan. To make reusable rules requires an induction system that can generate concepts in a first-order language (ie. with variables and relations).

3.4 Brittleness

When strong disturbances are allowed in the simulation, the airplane can be sent a considerable distance off-course. If there were no training data for such a situation, the rules in the autopilot will be meaningless and will result in nonsensical behaviour. A human pilot would use a deep model to guess the appropriate action to take but our present system cannot even recognise the limits of its knowledge, let alone reason about is predicament.

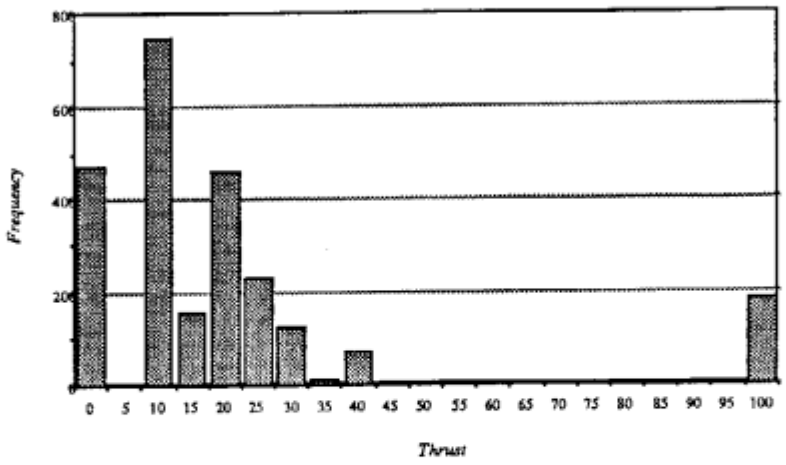
3.5 Determining class values

C4.5 expects class values to be discrete but the values for elevator, rollers, thrust and flaps are numeric. No ILP system currently allows numeric values. For these experiments, a preprocessor breaks up the action settings into sub-ranges that can be given discrete labels. Sub-ranges are chosen by analysing the frequency of occurrence of action values. This analysis must be done for each pilot to correctly reflect differing flying styles. There are two disadvantages to this method. One is that if the sub-ranges are poorly chosen, the rules generated will use controls that are too fine or too course. Secondly, C4.5 has no concept of ordered class values, so classes cannot be combined during the construction of the decision tree.

Figure 1 shows the frequency of thrust values in stage 6 of the data for one pilot. Since thrust is controlled by a keystroke, it is increased and decreased by a fixed amount, 10%. The values with very low frequencies are those that were passed through on the way to a desired setting. The graph reflects the facts that this pilot held the thrust at 100% until the approach to the runway began. The thrust was then brought down to 40% immediately and gradually decreased to 10% where it remained for most of the approach. Close to the runway, the thrust was cut to 0 and the plane glided down the rest of the way.

In this case, class values corresponding to 0, 10, 15, 10, 25, 30, 35, 40 and 100 were used. Anything above 40% was considered full-throttle. Anything below 10% was considered idle. Another reasonable clustering of values could be to group values from 15 to 35 together.

Figure 5. Frequency of thrust values in stage 6



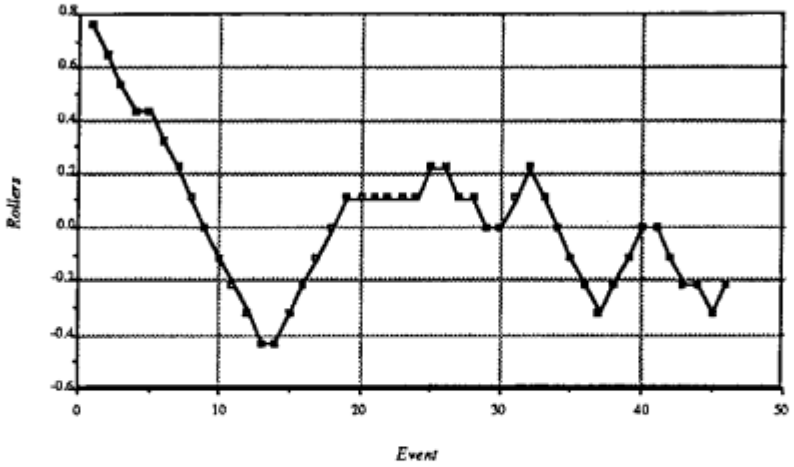
3.6 Absolute and Incremental Controls

An event is recorded when there is a change in one of the control settings. A change is determined by keeping the previous state of the simulation in a buffer. If any of the control settings are different in the current state, a change is recognised. For example, if the thrust is being reduced from 100% to 40%, all of the values in between are recorded. For thrust, these values are easily eliminated as noise during induction.

It is not so easy to eliminate spurious values from the elevator and rollers data. Both thrust and flaps can be set to a particular value and left. However, the effects of the elevator and rollers are cumulative. If we want to bank the aircraft to the left, the stick will be pushed left for a short time and then centred since keeping it left will cause the airplane to roll. Thus, the stick will be centred after most elevator or roller actions. This means that many low elevator and roller values will be recorded as the stick is pushed out and returned to the centre position.

To ensure that records of low elevator and roller values do not swamp the other data, another filter program removes all but the steady points and extreme points in stick movement. Figure 2 shows a small sample of roller settings during a flight. Each point on the graph represents one event. Clearly many of the points are recorded as part of a single movement. The filter program looks for points of inflection in the graph and only passes those on to the induction program. In this graph, only the points marked in black will get through the filter.

Figure 6. Change in rollers



3.7 Good Pilots are Bad

Another problem for data collection is the surprising fact that poor pilots often provide better training data than good pilots. Of the three pilots for whom we have data, one flew very erratically, the other two flew smoothly. The data from the erratic pilot have been the easiest to synthesise rules from. This is because there are many more examples of actions being performed to correct errors. A good pilots provides few of these, so the induction program does not generate rules for course corrections when the aircraft strays. We must devise a learning system that is robust enough to produce reliable control rules for any pilot who is able to fly competently. One way of doing this is to incorporate disturbances in the simulation and record the pilots corrective manoeuvres. This will also be necessary since we want to demonstrate that the autopilot can handle difficult conditions.

4. Solutions?

Three components are required in an algorithm for learning control rules: a representation language that includes variables and relations, the ability to invent new attributes and relations, and using background knowledge of causality.

These issues have, to some extent, been investigated by researchers in Inductive Logic Programming (ILP). Bratko, Muggleton and Varsek (1991) have shown that causal models can be learned by ILP when a theory of qualitative physics is supplied as background knowledge. This work has not yet been extended to learning control actions but it is very promising.

Unfortunately, current ILP systems cannot handle the noisy numeric data generated by a flight simulator and propositional learning programs (such as C4.5) that can cope with numeric data, cannot make use of background knowledge. Thus we are faced with the prospect of trying to extend one or the other method. As an alternative, hybrid systems as suggested by Lavrac, Dzeroski and Grobelnik (1991) are an interesting

possibility. Using this approach, we take advantage of well-developed decision tree induction algorithms to perform a first pass of the data and then map the results of the decision tree analysis into a first-order language for further processing. In the transformation to a first-order language, constants are turned into variables, allowing the rules to be reused in a different context, thus enabling us to create building blocks for new flight plans.

The problem of constructing new attributes and relations to simplify the output of a learning program has been investigated but only using discrete valued data or simple numeric data. Our problem is that we are dealing with noisy numeric data that have very complex relationships. It is not reasonable to expect that a program will be able to reconstruct flight equations from the data or perform complex transformations without some help. So, while humans are unable to explain their subcognitive skills, they are able to provide knowledge in the form of a theory to help guide induction. Thus, it is essential to provide a flexible way of describing background knowledge so that it can be used to preprocess that data before induction.

Causal relations illustrate the importance of data analysis prior to induction. A single training example for the induction program consists of an action, along with the state of the simulation when the action was taken. Normally, an induction program is predictive. For example, the thrust rule discussed earlier, predicts that when the speed is greater than 130 knots, the thrust will be at 100%. However, we know that the training examples are logged when an action is taken. For example, a thrust action is recorded when the thrust is changed from one setting to another. So we already have some clues about causality. We know that the thrust has changed. Now we want to know why. One way of doing this is to record the state of the simulation, not as it is at the time of the action, but as it was some time before. In addition, if we have a qualitative model that tells us that airspeed depends on thrust and elevation and that a certain airspeed must be maintained to avoid stalling then these values can be weighted to help the induction program choose the criteria for an action.

The presence of a causal model can also help remove brittleness from the control system. We envisage a complete control system consisting of a high-level planner that uses a qualitative model to establish goals for the system. Rules derived from human-data are invoked to achieve known goals. When the system wanders into regions for which there has been no training, the qualitative model may be used to guide control. This is necessarily slower and not desirable when rapid real-time response is required, but may be the only alternative in unknown regions.

5. Conclusion

We have presented a problem in learning control rules for real-time systems by observing human operators. This domain of applications has great significance for industry and is fruitful area of research. While existing induction algorithms go some way toward solving the problem, they are inadequate for further progress. This requires new techniques being pioneered in Inductive Logic Programming. However,

ILP has not yet addressed a number of issues that are essential if it is to move into the domain of control systems. These include processing noisy numeric data and the invention of predicates to describe complex numeric relationships. In addition the work that has already begun in combining qualitative modelling and ILP must be developed further.

Acknowledgments

Donald Michie suggested the problem and the method of using induction to learn reactive strategies. Scott Hurst and Dana Kedzier were the chief test pilots and data analysts. Mark Pendrith performed many modifications to the flight simulator. Jim Kehoe and Peter Horne conducted human reaction time studies and collected much valuable data. Silicon Graphics Incorporated made the source code of the flight simulator available. This research has been supported by the Australian Research Council and the University of New South Wales.

References

Bratko, I., Muggleton, S. and Varsek, A. (1991). Learning Qualitative Models of Dynamic Systems. In S. Muggleton (Ed.), *Proceedings of the International Workshop on Inductive Logic Programming*, (pp. 207-224). Viana do Castelo, Portugal.

Chambers, R. A. and Michie, D. (1969). Man-machine co-operation on a learning task. In R. Parslow,R. Prowse and R. Elliott-Green (Eds.), *Computer Graphics: Techniques and Applications* London: Plenum.

Donaldson, P. E. K. (1960). Error decorrelation: a technique for matching a class of functions. In *Proceedings of the Third International Conference on Medical Electronics*, (pp. 173-178).

Lavrac, N., Dzeroski, S. and Grobelnik, M. (1991). Learning Non-Recursive Definitions of Relations with LINUS. In Y. Kodratoff (Eds.), *Proceeding of the 1991 European Working Session on Learning* (pp. 265-281). Porto, Portugal: Springer-Verlag.

Michie, D., Bain, M., and Hayes-Michie, J.E. (1990). Cognitive Models from Subcognitive Skills. In M. Grimble, S. McGhee and P. Mowforth (Eds.) *Knowledge-base Systems in Industrial Control.*Peter Peregrinus.

Quinlan, J. R. (1987). Simplifying decision trees. *International Journal of Man-Machine Studies*, **27**, 221-234.

Sammut, C., Hurst, S., Kedzier, D. and Michie, D. (1992). Learning to Fly. In D. Sleeman (Ed.) *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen: Morgan-Kaufmann.

Widrow, B. and Smith, F. W. (1964). Pattern recognising control systems. In J. T. Tou and R. H. Wilcox (Eds.), *Computer and Information Sciences* Clever Hume Press.