EFFICIENT INDUCTION OF LOGIC PROGRAMS

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

Abstract

Recently there has been increasing interest in systems which induce first order logic programs from examples. However, many difficulties need to be overcome. Well-known algorithms fail to discover correct logical descriptions for large classes of interesting predicates, due either to the intractability of search or overly strong limitations applied to the hypothesis space. In contrast, search is avoided within Plotkin's framework of relative least general generalisation (rlgg). It is replaced by the process of constructing a unique clause which covers a set of examples relative to given background knowledge. However, such a clause can in the worst case contain infinitely many literals, or at best grow exponentially with the number of examples involved. In this paper we introduce the concept of h-easy rlgg clauses and show that they have finite length. We also prove that the length of a certain class of "determinate" rlgg is bounded by a polynomial function of certain features of the background knowledge. This function is independent of the number of examples used to construct them. An existing implementation called GOLEM is shown to be capable of inducing many interesting logic programs which have not been demonstrated to be learnable using other algorithms.

1 Introduction

Imagine that a learning program is to be taught the logic program [1] for "quick-sort". The following definitions are provided as background knowledge.

```
partition(X,[],[],[]).
partition(X,[Head|Tail],[Head|Sublist1],Sublist2) ←
        lte(Head,X),
        partition(X,Tail,Sublist1,Sublist2).
partition(X,[Head|Tail],Sublist1,[Head|Sublist2]) ←
        gt(Head,X),
        partition(X,Tail,Sublist1,Sublist2).
```

[1] For formal definitions of logic programs and other concepts used in logic programming see [5].

```
append([],List,List).
append([Head|Tail],List,[Head|Rest]) ←
        append(Tail,List,Rest).

lte(0,0).
lte(0,1).
...
gt(1,0).
gt(2,0).
gt(2,1).
...
```

The program is then provided with a set of positive ground examples of quick-sort, such as qsort([],[]), qsort([0],[0]), qsort([1,0],[0,1]), ... together with some negative examples such as qsort([1,0],[1,0]). In this case we might hope that the algorithm would, given a sufficient number of examples, suggest the following clauses for "quick-sort".

```
qsort([],[]).
qsort([Head|Tail],Sorted) ←
            partition(Head,Tail,List1,List2),
            qsort(List1,Sorted1),
            qsort(List2,Sorted2),
            append(Sorted1,[Head|Sorted2],Sorted).
```

The general setting for such a learning program can be described as follows. Given background knowledge $\mathcal{K}$ (in this case partition, append, etc.), a set of positive examples $\mathcal{E}^+$ (ground instances of quick-sort) and negative examples $\mathcal{E}^-$ for which $\mathcal{K} \not\vdash \mathcal{E}^+$, find a hypothesised set of clauses $\mathcal{H}$ such that $\mathcal{K} \wedge \mathcal{H} \vdash \mathcal{E}^+$ and $\mathcal{K} \wedge \mathcal{H} \not\vdash \mathcal{E}^-$. Many existing algorithms [7, 2, 12, 13, 11, 6] operate within this theoretical framework [8, 7]. However, not many of these have reached a sufficient state of maturity to be widely applicable in practice. We believe that first-order learning programs must have the following properties to achieve wide-scale application.

- **Effective**. The program can learn at least the class of efficient Prolog programs, such as quick-sort.
- **Efficient**. A hypothesis can be constructed from up to 10,000 examples in a reasonably short time (eg. 1 minute).
- **Mode**. Examples can be provided either incrementally or as a "batch".
- **Invention**. The program can "invent" some of its background knowledge by introducing auxiliary predicates. In the case of quick-sort it might have to invent partition.
- **Noise**. The program can deal with errors in the descriptions of examples or background knowledge.

In this paper we will concentrate on the problems of effectiveness and efficiency.

Existing learning programs divide into "top-down" or model-driven methods and "bottom-up" or data-driven methods. Top-down methods such as Shapiro's MIS [13] and Quinlan's FOIL [11] search the hypothesis space of clauses from the most general towards the most specific. MIS employs a breadth-first search through successive levels of a "clause refinement" lattice,

considering progressively more complex clauses. The combinatorics of this search are such that Shapiro failed to demonstrate that clauses as complex as the recursive clause of quick-sort were learnable within reasonable time limits. To achieve greater efficiency Quinlan's FOIL greedily searches the same space guided by an information measure similar to that used in ID3 [9]. This measure supports the addition of a literal in the body of a clause on the basis of its ability to discriminate between positive and negative examples. This gains efficiency at the expense of completeness. For instance the literal *partition(Head,Tail,List1,List2)* in the recursive quick-sort clause does not produce any discrimination between positive and negative examples. As a result quick-sort cannot be learned by FOIL. The problem recurs throughout a large class of predicates in which new variables in the body are functionally derived from terms in the head of the clause. Such predicates include arithmetic multiply, list reverse and many real-world domains.

Bottom-up algorithms based on inverting resolution [12, 6] also have problems related to search strategies. In the framework of inverse resolution clauses are constructed by progressively generalising examples with respect to given background knowledge. Each step in this process inverts a step of a resolution derivation of the examples from the clause and background knowledge. Search problems are incurred firstly since there may be many inverse resolvents at any stage, and secondly because several inverse resolution steps may be necessary to construct the required clause. Clauses such as the recursive quick-sort clause are difficult to learn without carefully chosen examples due to local minima in the search space.

Thus problems related to search hamper both top-down and bottom-up methods. In search-based methods efficiency is gained only at the expense of effectiveness.

2 Relative least general generalisations

Plotkin's [8, 7] notion of relative least general generalisation (rlgg) replaces search by the process of constructing a unique clause which covers a given set of examples. Firstly Plotkin defines θ-subsumption, a partial ordering of generality over clauses in the absence of background knowledge. Thus clause C θ-subsumes clause D, or $C \preceq D$, if and only if there exists a substitution θ such that $C\theta \subseteq D$. Next he defines the least general generalisation (lgg) C of two clauses D_1 and D_2 as being the greatest lower bound within the clause lattice induced by the relation $\preceq$. In the following discussion we restrict Plotkin's least general generalisation of two clauses with respect to given background knowledge (rlgg) to the case of the rlgg of a pair of ground atoms. Let $\mathcal{K}$ be a set of background clauses, e_1 and e_2 be two ground atomic examples ($e_1, e_2 \in \mathcal{E}^+$) for which $\mathcal{K} \not\vdash e_1$ and $\mathcal{K} \not\vdash e_2$. Clause C is said to be the least general generalisation of e_1 and e_2 relative to $\mathcal{K}$ whenever C is the least general clause within the θ-subsumption lattice for which $\mathcal{K} \wedge C \vdash e_1 \wedge e_2$, where C is used only once in the derivation of both e_1 and e_2. A method for constructing rlggs can be derived as follows.

$$
\begin{aligned}
\mathcal{K} \wedge C &\vdash e_1 \\
C &\vdash \mathcal{K} \rightarrow e_1 \\
&\vdash C \rightarrow (\mathcal{K} \rightarrow e_1) \\
&\vdash C \rightarrow (\overline{\mathcal{K}} \vee e_1) \\
&\vdash C \rightarrow ((\overline{a_1 \wedge a_2 \wedge \ldots}) \vee e_1) \qquad (1) \\
&\vdash C \rightarrow ((\overline{a_1} \vee \overline{a_2} \vee \ldots) \vee e_1)
\end{aligned}
$$

In step 1 $\mathcal{K}$ is replaced by a set of ground atoms $a_1 \wedge a_2 \wedge \ldots$ representing a model of $\mathcal{K}$. Clearly various models of $\mathcal{K}$ could be chosen here. Later in this section we will discuss the implications

of various choices for the model of $\mathcal{K}$. Similarly for e_2

$$\vdash \quad C \rightarrow ((\overline{a_1} \vee \overline{a_2} \vee \ldots) \vee e_2)$$

If we let C_1 and C_2 be $((\overline{a_1} \vee \overline{a_2} \vee \ldots) \vee e_1)$ and $((\overline{a_1} \vee \overline{a_2} \vee \ldots) \vee e_2)$ respectively it follows that

$$\vdash C \rightarrow lgg(C_1, C_2)$$

where $lgg(C_1, C_2)$ is the least general generalisation of the clauses C_1 and C_2. In fact by definition $C = lgg(C_1, C_2)$.

Plotkin [8, 7] describes the following method for constructing lggs. The lgg of the terms $f(s_1, .., s_n)$ and $f(t_1, .., t_n)$ is $f(lgg(s_1, t_1), .., lgg(s_n, t_n))$. The lgg of the terms $f(s_1, .., s_n)$ and $g(t_1, .., t_n)$ where $f \neq g$ is the variable v where v represents this pair of terms throughout. The lgg of two atoms $p(s_1, .., s_n)$ and $p(t_1, .., t_n)$ is $p(lgg(s_1, t_1), .., lgg(s_n, t_n))$, the lgg being undefined when the sign or predicate symbols are unequal. The lgg of two clauses C_1 and C_2 is $\{l : l_1 \in C_1$ and $l_2 \in C_2$ and $l = lgg(l_1, l_2)\}$. In the following the length of a clause is the number of literals contained within it.

Theorem 1 *$lgg(C_1, C_2)$ has length at most $|C_1| \cdot |C_2|$.*
Proof. *Every element of $lgg(C_1, C_2)$ is the lgg of an element of the set $C_1 \times C_2$. Thus $|lgg(C_1, C_2)| \leq |C_1 \times C_2| = |C_1| \cdot |C_2|$.* □

In the case of rlggs $|C_1| = |C_2|$ and thus the length of the rlgg clause is at most $|C_1|^2$. The lgg of n clauses $lgg(C_1, C_2, .., C_n)$ is $lgg(C_1, lgg(C_2, .., C_n))$.

Corollary 2 *The length of $lgg(C_1, C_2, .., C_n)$ is at most $|C_1| \cdot |C_2| \cdot .. \cdot |C_n|$.*
Proof. *Trivial extension of theorem 1.* □

Consider the computation of rlgg for quick-sort. Assume that $\mathcal{K}$ consists of the definitions of partition and append given at the beginning of this paper together with the unit clauses qsort([],[]), qsort([1,0],[0,1]) and qsort([4,3],[3,4]). Let e_1 = qsort([1],[1]) and e_2 = qsort([2,4,3,1,0],[0,1,2,3,4]). In this case

C_1 = qsort([1],[1]) ←
append([],[1],[1]),
append([0,1],[2,3,4],[0,1,2,3,4]), ...,
partition(1,[],[],[]),
partition(2,[4,3,1,0],[1,0],[4,3]), ...,
qsort([],[]),
qsort([1,0],[0,1]),
qsort([4,3],[3,4]), ...

C_2 = qsort([2,4,3,1,0],[0,1,2,3,4]) ←
append([],[1],[1]),
append([0,1],[2,3,4],[0,1,2,3,4]), ...,
partition(1,[],[],[]),
partition(2,[4,3,1,0],[1,0],[4,3]), ...,
qsort([],[]),
qsort([1,0],[0,1]),
qsort([4,3],[3,4]), ...

$C = lgg(C_1, C_2)$

= qsort([A|B],[C|D])←
append(E,[A|F],[C|D]),
append([],[1],[1]),
append(G,[H|I],[J|K]),
append([0,1],[2,3,4],[0,1,2,3,4]),
partition(A,B,L,M),
partition(1,[],[],[]),
partition(H,N,O,P),
partition(2,[4,3,1,0],[1,0],[4,3]),
qsort(L,E),
qsort([],[]),
qsort(O,G),
qsort([1,0],[0,1]),
qsort(M,F),
qsort(P,I),
qsort([4,3],[3,4]) ...

Note that C contains all the literals of the recursive quick-sort clause. However, it also contains many other literals. There are various logical and practical problems involved with the construction of rlgg clauses.

- **Ground model of $\mathcal{K}$.** In the derivation of rlgg, step 1 replaces $\mathcal{K}$ by a ground model of $\mathcal{K}$. Buntine [2] suggests the use of a finite subset of the least Herbrand model of $\mathcal{K}$. Clearly it is not possible to construct an infinite model of $\mathcal{K}$, though it would seem desirable to use a finite subset with some well-defined properties.

- **Intractably large rlggs**. Even when a finite subset $\mathcal{M}$ of a ground model of $\mathcal{K}$ is used the rlgg of n examples can be of length $|M|^n + 1$. Buntine [2] reports a clause of length 2,000 literals for the relatively simple problem of learning list membership. We have experience of clauses containing tens of thousands of literals when learning arithmetic multiplication. In the case of constructing a rlgg of 6 quick-sort examples, with 15 ground elements of the model (instances) of partition, 49 instances of append and 16 instances of quick-sort, the clause will have length $15^6 + 49^6 + 16^6 + 1 = 13,869,455,043$.

- **Infinite rlggs**. Plotkin [8, 7] shows that in some cases, a logically reduced rlgg (see below) can be infinite in length. Thus in some sense any use of finite models can restrict the ability to construct 'true' rlggs.

- **Logical clause reduction**. Plotkin [8, 7] suggests the use of logical clause reduction to eliminate redundant literals in rlgg clauses. A literal l is logically redundant in clause C with respect to $\mathcal{K}$ whenever $\mathcal{K} \wedge C \vdash (C - \{l\})$. A clause is logically reduced if and only if it contains no logically redundant literals. Clearly this approach is at worst semi-decidable and at best highly inefficient since it requires theorem proving. Moreover, even after logical reduction clauses can remain very large. Buntine [2] reports a reduced list-membership clause containing 43 literals. The following is typical of a logically irreducible clause for reversing lists.

reverse([A|B],[C|D]) ←
reverse(B,E),

append(E,[A],[C|D]),
reverse(D,F),
append(F,[C],[A|B]).

3 Restricted forms of background knowledge

3.1 The use of ground facts as background knowledge

In the case in which $\mathcal{K}$ contains only ground facts the problem of choosing a ground model $\mathcal{M}$ disappears. We simply take $\mathcal{M}$ as being equal to $\mathcal{K}$. According to corollary 2 the rlgg of n examples in this case is finite and has length at most $|\mathcal{K}|^n + 1$.

3.2 h-easy models

We have already noted that in the case that $\mathcal{K}$ consists of arbitrary clauses the model $\mathcal{M}$ of $\mathcal{K}$ can be infinite, leading to possibly infinite rlgg clauses. The notion of *h-easiness* suggested originally by Blum and Blum [1] and adapted by Shapiro [13] can be applied to this problem. Whereas h is a total recursive function within [1, 13], in our definition h is simply a natural number.

Definition 3 *Given a logic program $\mathcal{K}$ an atom a is h-easy with respect to $\mathcal{K}$ if and only if there exists a derivation of a from $\mathcal{K}$ involving at most h binary resolutions. The Herbrand h-easy model of $\mathcal{K}$, $M_h(\mathcal{K})$ is the set of all Herbrand instantiations of h-easy atoms of $\mathcal{K}$.*

Theorem 4 *For any finite h the number of h-easy atoms of $\mathcal{K}$ is finite.*
Proof. *This follows from the fact that there are only a finite number of such derivations.* □

However this does not imply that $M_h(\mathcal{K})$ is finite. For example, consider the case in which $\mathcal{K}$ = {member(X,[X|Y])}. Although there is only one h-easy atom derivable from $\mathcal{K}$, $M_h(\mathcal{K})$ is the infinite set {member([],[[]]),member([],[[[]]]),...}, using [] as the constant to construct the Herbrand base.

3.3 Generative clauses

In this section we describe a restricted form of Horn clause program for which $M_h(\mathcal{K})$ is finite. First we define the notion of derivation more formally.

Definition 5 *The n-atomic-derivation set of $\mathcal{K}$, $D^n(\mathcal{K})$ is defined recursively as follows. $D^0(\mathcal{K})$ is the set of unit clauses in $\mathcal{K}$. $D^n(\mathcal{K}) = D^{n-1}(\mathcal{K}) \cup \{A\theta_1..\theta_m : A \leftarrow B_1,..,B_m \in \mathcal{K}$ and for each B_i there exists $B_i' \in D^{n-1}(\mathcal{K})$ such that θ_i is the mgu of B_i and $B_i'\}$. The atomic-derivation-closure $D^*(\mathcal{K})$ is the set $D^0(\mathcal{K}) \cup D^1(\mathcal{K}) \cup ...$.*

Definition 6 *The logic program $\mathcal{K}$ is said to be semantically generative if and only if every element of $D^*(\mathcal{K})$ is ground.*

The following syntactic constraint on logic programs corresponds to this semantic definition.

Definition 7 *The clause $A \leftarrow B_1,..,B_m$ is syntactically generative whenever the variables in A are a subset of the variables in $B_1,..,B_m$. The logic program $\mathcal{K}$ is syntactically generative if and only if every clause in $\mathcal{K}$ is syntactically generative.*

Not all semantically generative logic programs are syntactically generative, as the following example shows. Let $\mathcal{K}$ contain only the clause append([Head|Tail],List,[Head|Rest]) $\leftarrow$ append(Tail,List,Rest). Clearly $\mathcal{K}$ is semantically generative since $D^*(\mathcal{K})$ is empty. However, $\mathcal{K}$ is not syntactically generative although $\mathcal{K}' = \{$append([Head|Tail], List, [Head|Rest]) $\leftarrow$ append(Tail,List,Rest), natural_number(Head)$\}$ is both semantically and syntactically generative. The following theorem establishes the correspondence between these two definitions.

Theorem 8 *Every syntactically generative logic program $\mathcal{K}$ is semantically generative.*
Proof. *We prove the result inductively for all n. For $n = 0$ $D^0(\mathcal{K})$ contains the unit clauses of $\mathcal{K}$, which must each be ground since they have empty bodies. Suppose every element of $D^{n-1}(\mathcal{K})$ is ground. Let $A \leftarrow B_1, .., B_m$ be a clause in $\mathcal{K}$ and $A\theta_1..\theta_m$ be an element of $D^n(\mathcal{K}) - D^{n-1}(\mathcal{K})$ such that for each B_i there exists $B_i' \in D^{n-1}(\mathcal{K})$ for which θ_i is the mgu of B_i and B_i'. By the inductive assumption each $B_i' \in D^{n-1}(\mathcal{K})$ is ground thus each $B_i\theta_i$ is ground. Since each θ_i is a ground substitution and every variable in A is found in the domain of $\theta_1..\theta_m$ it follows that $A\theta_1..\theta_m$ is ground, which completes the proof.* □

It is worth noting that the least Herbrand model, the success set and the atomic-derivation-closure of a semantically generative logic program are all equal. Throughout the rest of this paper we use the phrase 'generative' to mean 'syntactically generative'.

Theorem 9 *For every generative logic program $\mathcal{K}$ and every h $M_h(\mathcal{K})$ is finite.*
Proof. *Follows from theorem 4, theorem 8 and definitions 6 and 7.* □

Corollary 10 *For every generative logic program $\mathcal{K}$ and set of examples $e_1, .., e_n$ the rlgg of $e_1, .., e_n$ with respect to $M_h(\mathcal{K})$ is finite and has length at most $|M_h(\mathcal{K})|^n + 1$.*
Proof. *Follows from corollary 2 and theorem 9.* □

In an incremental learning system, learned clauses are added to the background knowledge. Thus it is reasonable to constrain hypothesised clauses to also being generative. However, this is generally not a very strong constraint since any clause which is not generative can be made so by the addition of literals stating type information about variables in the head of the clause.

4 Restrictions on the hypothesis language

The results of the previous sections show that although finite, the length of the rlgg of n examples can grow exponentially in n. In this section we show that by placing certain 'weak' restrictions on the hypothesis language we find that the number of literals in rlgg clauses is bounded by a polynomial function independent of n. In order to see the significance of this language restriction consider the following unreduced rlgg clause for list membership.

```
member(A,[B,C|D]) ←
        member(A,[C|D]),
        member(E,[F|G]),
        member(A,[A,C|D]),
        member(A,[E,F,C|D]),
        member(H,[I|J]), ...
```

The literals member(H,[I|J]) and member(A,[A,C|D]) could be logically reduced. However, almost all literals introduce variables not found in the head of the clause. In our experience such literals account for most of the exponential growth of rlgg clauses. The vast majority of these

literals can be avoided by considering only those which introduce 'determinate terms'. First we will define the notion of determinate terms. In this definition we use the phrase 'ordered Horn clause' to mean that there exists a total ordering over the literals in the body of the clause.

Definition 11 *Let $\mathcal{K}$ be a logic program and the examples $\mathcal{E}$ be a set of ground atoms. Let $A \leftarrow B_1, .., B_m, B_{m+1}.., B_n$ be an ordered Horn clause and t be a term found in B_{m+1}. t is a determinate term with respect to B_{m+1} if and only if for every substitution θ such that $A\theta \in \mathcal{E}$ and $\{B_1, .., B_m\}\theta \subseteq M(\mathcal{K})$ there is a unique atom $B_{m+1}\theta\delta$ in $M(\mathcal{K})$, i.e. for any such θ there is a unique valid ground substitution δ whose domain is the variables in t.*

Thus given appropriate background knowledge concerning decrement, plus and multiply, the variables D and E are determinate with respect to the literals decrement(B,D) and multiply(A,D,E) in the clause

```
multiply(A,B,C) ←
        decrement(B,D),
        multiply(A,D,E),
        plus(A,E,C).
```

Notice that the valid substitutions for a determinate term t in any literal B_{m+1} are a function of the substitutions applied to a certain number of other terms $s_1, .., s_d$ within B_{m+1}. In this case we say that literal B_{m+1} has degree d with respect to t. In the case of the literal decrement(B,D) the valid substitutions for D are dependent on the substitutions applied to only one term, B. Thus decrement(B,D) has degree 1 with respect to D. In the case of the literal multiply(A,D,E), the values for E are dependent on the substitutions to the two terms A and D, i.e. multiply(A,D,E) has degree 2 with respect to E. We can define a restricted hypothesis language by placing a limit j on the maximum degree of any literal with respect to any term.

Notice also that the dependencies between variables induce a partial ordering over the literals within the recursive multiply clause. Thus we might say that the literal decrement(B,D) is found at depth 1 within the multiply clause since its determinate variable D is a function of B, a variable found in the head of the clause. On the other hand the literal multiply(A,D,E) is found at depth 2 since its determinate variable E is a function of A and D, where D is determined by a literal at depth 1. However, the literal plus(A,E,C) has depth 1 since the variable E is determined by A and C, which are both found in the head of the clause. Again we can define a restricted hypothesis language by placing a limit i on the maximum depth of any literal in an hypothesised clause. Combining the parameters of depth and degree of literals we get the the following definition of ij-determination.

Definition 12 *Let $\mathcal{K}$ be a logic program and the examples $\mathcal{E}$ be a set of ground facts. Every unit clause is 0j-determinate. An ordered clause $A \leftarrow B_1, .., B_m, B_{m+1}, .., B_n$ is ij-determinate if and only if a) $A \leftarrow B_1, .., B_m$ is (i-1)j-determinate, b) every literal B_k in $B_{m+1}, .., B_n$ contains only determinate terms and has degree at most j.*

In the following we use the phrase 'ij-determinate rlgg of n examples' to mean a clause containing all and only those literals from the rlgg which form an ij-determinate clause.

The following clauses are typical of the various classes of determinate clause up to but not including 22-determination.

11-determinate:
```
class(A,mammal) ←
        has_milk(A,true).
```

12-determinate: double(A,B) ←
plus(A,A,B).

21-determinate: grandfather(A,B) ←
father(A,C),
father(C,B).

Despite the fact that the predicates member and append are thought of as being 'non-deterministic' predicates by logic programmers, they both conform to the definition of being ij-determinate. In fact most logic programs found in standard texts conform to this definition. For instance the standard definitions of quick-sort, partition, list-reverse, arithmetic addition and multiplication are all ij-determinate with respect to their ground instances. An example of a clause which is not ij-determinate is the transitive definition of 'less-than': less_than(A,B) ← less_than(A,C), less_than(C,B). The variable C is not determinate with respect to any of the literals in the body of the clause. However, even this clause can be restated as the ij-determinate clause less_than(A,B) ← successor(A,C), less_than(C,B).

Although this restriction on the hypothesis space is relatively weak, it has the effect of reducing the above example of a clause for 'member' to member(A,[B,C|D]) ← member(A,[C|D]), member(A,[A,C|D]). The general effect of this language constraint is described by the following theorem. This theorem uses the notion of 'place' within an atom which can be defined recursively as follows. The term at place $\langle\langle g, m, i\rangle\rangle$ within $g(t_1, .., t_m)$ is t_i. The term at place $\langle\langle g_1, m_1, i_1\rangle, .., \langle g_n, m_n, i_n\rangle\rangle$ within $g_1(t_1, .., t_{m_1})$ is the term at place $\langle\langle g_2, m_2, i_2\rangle, .., \langle g_n, m_n, i_n\rangle\rangle$ within t_{i_1}.

Theorem 13 *Let $M_h(\mathcal{K})$ consist of facts concerning the predicates $p_1, .., p_m$. Let the number of distinct places in the facts of each predicate be represented by the numbers $f_1, .., f_m$ where $f = max(f_1, .., f_m)$. Let $e_1, .., e_n$ be a set of facts and the atom $lgg(e_1, .., e_n)$ contain t terms. The number of literals in the body of the ij-determinate rlgg of $e_1, .., e_n$ with respect to $M_h(\mathcal{K})$ is at worst $O((tfm)^{j^i})$.*

Proof. *Consider first the case of 1j-determinate rlggs. Each literal l in the body of the rlgg is determined by the j terms $s_1, .., s_j$ from the head of the clause found at at places $a_1, .., a_j$ within l. Supposing l is a literal of predicate p_k, $a_1, .., a_j$ must be a set chosen from a maximum of f_k places.*

This can be done in a maximum of $t^j C_j^{f_k}$ ways. Summing for all predicates we get $t^j \sum_{k=1}^{m} C_j^{f_k}$

literals. This can introduce at most $t^j f \sum_{k=1}^{m} C_j^{f_k}$ terms not found in the head of the clause. Thus

a 2j-determinate rlgg can contain up to $(t^j f \sum_{k=1}^{m} C_j^{f_k})^j \sum_{k=1}^{m} C_j^{f_k}$ literals, or $f^j t^{j^2} (\sum_{k=1}^{m} C_j^{f_k})^{j+1}$.

In the general case this is $f^{j^{i-1}} t^{j^i} (\sum_{k=1}^{m} C_j^{f_k})^{j^{i-1}+1}$ literals, which is $O((tfm)^{j^i})$. □

Note that whereas corollary 10 states that the size of the unreduced rlgg of n examples grows exponentially in n, theorem 13 demonstrates that for fixed values of i and j, the size of the ij-determinate rlgg is independent of n. Clearly i and j must be kept small to limit this worst case. However, a setting of i=j=2 is sufficient for a class of predicates which includes quick-sort, partition, append, member, arithmetic plus, multiply, n-choose-m and list reverse. For this class of predicate the rlgg has a polynomial upper bound of $O((tfm)^4)$. Note also that when $i = 1$ the rlgg length does not grow exponentially in m.

n	2	3	4	5	6	7	8	9	10	11	12	13	$\|M(\mathcal{K})\|$
Reverse	16	23	25	21	8								42
Multiply	48	81	52	49	48	47	43						278
N-Choose-M	116	105	88	87	71	66							273
Quick-sort	470	650	734	632	650	598	593	535	534	524	36	20	84

Table 1: Clause lengths based on varying numbers of randomly chosen examples

The upper bound of theorem 13 is very conservative. Table 1 demonstrates the typical effect of n on the length of 22-determinate rlgg clauses constructed using our implementation. In the table $M(\mathcal{K})$ is the number of ground facts used as background knowledge. Successive examples were chosen randomly without replacement from a large set of positive examples. The sequence was terminated when the rlgg clause covered all positive examples. The length of the rlgg clause is given for each corresponding value of n. From the table it can be observed that rlgg clause size does not increase exponentially. In fact it seems to have a tendency to eventually decrease with increasing n.

Intuitively one might expect rlggs to decrease in length with increasing n. The following example demonstrates why this is not the case. Let $\mathcal{K}$ contain the facts s(1,2) and s(2,3). The rlgg of the examples p(1,1) and p(2,2) is the ij-determinate clause p(A,A) $\leftarrow$ s(A,B). However the rlgg of p(1,1), p(2,2) and p(1,2) is the larger clause p(A,B) $\leftarrow$ s(A,C), s(B,D), where p(A,B)=lgg(p(1,1), p(2,2), p(1,2)), s(A,C)=lgg(s(1,2), s(2,3), s(1,2)), and s(B,D)=lgg(s(1,2), s(2,3), s(2,3)). Notice that the clause length increases because the arguments of the third fact p(1,2) are unequal, which results in an additional variable in the head of the clause.

5 Clause reduction

ij-determination has been found to avoid the exponential size explosion of rlggs. However it does not completely avoid the need to reduce clauses following their construction. As we have shown earlier, logical clause reduction is too weak to produce computationally useful clauses. In this section we present various alternative approaches to clause reduction.

5.1 Functional reduction

Shapiro [13] introduced the idea of describing the input and output status of arguments of a predicate as an additional constraint for constructing Horn clauses. We use a similar approach to this in order to reduce clauses which represent functions. Mode declarations such as mode(append(in,in,out)) allow the construction of a partially ordered graph which defines the computation of output variables from input variables. Figure 1 shows this 'functional' graph for quick-sort. The graph can be constructed by searching an and/or graph based on the literals within the unreduced rlgg clause. The and/or graph can be represented as a propositional Horn clause program which can be constructed as follows. For each literal l in the body of the rlgg clause which computes variables $v_1,..,v_n$ from variables $u_1,..,u_m$ there is associated a set of propositional Horn clauses $(v_1 \leftarrow u_1,..,u_m),..,(v_n \leftarrow u_1,..,u_m)$. For each input variable i in the head of the rlgg clause there is an associated propositional fact i. A functional graph can be constructed if and only if for each output variable o in the head of the rlgg clause there is at least one refutation with respect to this propositional program. It is well-known that this

can be decided in time linear in the size of the propositional program. Multiple refutations of the output variables lead to multiple functional reductions of the rlgg clause. This can provide multiple algorithmic solutions which compute the target predicate (see quick-sort example in section 6).

5.2 Negative-based reduction

Many machine learning programs, including MIS [13] make use of both positive and negative examples. A common assumption is that the hypothesised clause should cover as many positive examples as possible without covering any negative examples. This approach can be used to reduce clauses. Given a clause $A \leftarrow B_1, .., B_n$ we need a method which avoids the requirement of having to test all subsets of $B_1, .., B_n$. Our approach is as follows. Let B_i be the first literal in the ordered set $B_1, .., B_n$ such that the clause $A \leftarrow B_1, .., B_i$ covers no negative examples. Now reduce the clause $A \leftarrow B_i, B_1, .., B_{i-1}$ in the same manner and iterate. The reduction process terminates when the clause length remains the same within a cycle. Clearly the algorithm will terminate after at most $O(n^2)$ cycles.

6 Implementation and results

RELEX [4] was an early Prolog implementation which used the rlgg framework. RELEX adopted an incremental control strategy similar to CIGOL [6]. RELEX was demonstrated on learning a definition of Fibonacci numbers and a greatest common denominator algorithm. RELEX did not employ the ij-determinate constraint described in section 4 and consequently had difficulties in constructing clauses as complex as the recursive quick-sort clause from a large set of randomly chosen examples (though it was able to do so with carefully chosen examples such as those in section 2). In the remainder of this section we will describe implementation details of GOLEM, a C-coded program which uses the ij-determinate constraint.

Given a set of positive and negative examples $\mathcal{E}^+$ and $\mathcal{E}^-$ and background knowledge $\mathcal{K}$ we are often not interested in constructing a single clause which is the rlgg of $\mathcal{E}^+$, but rather a set of hypothesised clauses which of $\mathcal{E}^+$, but rather a set of hypothesised clauses $\mathcal{H}$ which together cover all examples in $\mathcal{E}^+$ and do not cover any elements of $\mathcal{E}^-$. The following 'greedy' control strategy has been adopted.

Let $\mathcal{E}^+$ be a set of positive examples.
Let $\mathcal{E}^-$ be negative examples.
Let $M_h(\mathcal{K})$ be an h-easy model of background knowledge $\mathcal{K}$.
Let s be a given sample limit.
Let Pairs(s) be a random sample of pairs from $\mathcal{E}^+$.
Let Lggs = {C: $\{e, e'\} \in$ Pairs(s) and C=rlgg($\{e, e'\}$) wrt $M_h(\mathcal{K})$ and C consistent wrt $\mathcal{E}^-$}.

Figure 1: Functional graph for quick-sort([A|B],[C|D]).

Predicate	$\|\mathcal{E}^+\|$	$\|M_h(\mathcal{K})\|$	Single rlgg(sec)	Complete Search	Reduction
Propositional-animals	10	70	0.1	1.3	0.02
Member	14	20	0.2	2.4	0.02
Less-than	15	116	0.08	1.4	0.03
Reverse	12	42	0.1	2.6	0.03
Multiply	21	278	0.2	13.2	0.08
N-Choose-M	15	273	0.5	8.1	0.12
Quick-sort	15	84	6.8	11.6	0.07

Table 2: Times taken for constructing and reducing various clauses

Let S be the pair $\{e, e'\}$ whose rlgg has the greatest cover in Lggs.
DO
  Let $\mathcal{E}^+$(s) be a random sample of size s from $\mathcal{E}^+$.
  Let Lggs = {C: $e' \in \mathcal{E}^+(s)$ and C=rlgg(S $\cup$ $\{e'\}$) consistent wrt $\mathcal{E}^-$}.
  Find e' which produces the greatest cover in Lggs.
  Let S = S $\cup$ $\{e'\}$.
  Let $\mathcal{E}^+ = \mathcal{E}^+$ - cover(rlgg(S)).
WHILE increasing-cover

If the sample limit is set to be twice the expected number of clauses, where the cover of each clause is disjoint, it can easily be shown that a randomly chosen pair of examples will be covered by one of the clauses with high probability. The above algorithm constructs a clause which can be reduced using the clause reduction methods of the previous section. By iterating over this process removing covered examples from $\mathcal{E}^+$ at each stage a set of clauses is produced which covers all the positive examples.

Table 2 gives times and number of examples used for various Prolog clauses constructed using GOLEM. Times quoted are for runs carried out on Sun SPARCStation/330. The times are given in seconds for a) construction of a single rlgg clause from a pair of examples, b) complete search for a single clause using the greedy algorithm above and c) reducing the clause using a combination of functional and negative reduction. The reduced clauses constructed for the tabulated cases above are as follows.

Propositional-animals:
```
class(A,mammal) ←
    has_milk(A,true).
```

Member:
```
member(A,[B,C|D]) ←
    member(A,[C|D]).
```

Less-than:
```
lte(A,B) ←
    successor(A,C),
    lte(C,B).
```

Reverse:
```
reverse([A|B],[C|D]) ←
    reverse(B,E),
    append(E,[A],[C|D]).
```

Multiply:

```
multiply(A,B,C) ←
        decrement(B,D),
        multiply(A,D,E),
        plus(A,E,C).
```

N-Choose-M:

```
choose(A,B,C) ←
        decrement(B,D),
        decrement(A,E),
        choose(E,D,F),
        multiply(F,A,G),
        divide(G,B,C).
```

Quick-sort:

```
qsort([A|B],[C|D]) ←
        partition(A,B,E,F),
        qsort(F,G),
        qsort(E,H),
        append(H,[A|G],[C|D]).
```

In the case of quick-sort GOLEM gives multiple solutions as a result of the reduction method. GOLEM suggests the following clause as an alternative hypothesis to the above quick-sort clause.

```
qsort([A|B],[C|D]) ←
                qsort(B,E),
                partition(A,E,F,G),
                append(F,[A|G],[C|D]).
```

The clause represents a way of expressing insertion sort, an inefficient algorithm compared to quick-sort. Although not implemented, these hypotheses could be distinguished on the basis of choosing the clause which produces the minimal proofs of the examples, i.e. the one with lower time complexity. A similar case occurs with n-choose-m, in which the choice is between the clause above and a clause which recurses in the manner of Pascal's triangle. Without tabulation the Pascal's triangle method is NP in the size of the arguments.

7 Conclusion

In the introduction we introduced various properties that first-order learning programs should possess. In this paper we have shown that methods based on relative least general generalisation can be both efficient and effective according to our criteria. We believe that the main advantages of using rlgg as an underlying hypothesis generator lies in the ability to avoid search in the construction of clauses. In fact we have found that GOLEM produces robust and rapid behaviour when tested on problems involving up to 10,000 examples. At present we are applying GOLEM to the real-world problems of secondary structure prediction for proteins and qualitative model construction for fault diagnosis of satellites.

Although we have presented some theoretical analysis of rlgg within this paper, we believe that characterisation within Valiant's PAC model [14] of learnability is a challenging problem.

We see the main extensions of this research being in the areas of dealing with noise and predicate invention. A simple method of handling noise is to allow hypotheses to cover a certain number of negative examples based on information theoretic thresholds. This would be similar to the approach taken in C4 [10] and CN2 [3]. We hope to achieve predicate invention within

this framework by extending the methods used in CIGOL [6].

Acknowledgements.
We would like to thank Wray Buntine for his advice and help in this research. We would also like to thank members of the Turing Institute machine learning group for helpful and interesting discussions. This work was supported partly by the Esprit Ecoles project 3059 and an SERC Post-graduate fellowship held by Stephen Muggleton.

References

[1] L. Blum and M. Blum. Towards a mathematical theory of inductive inference. *Information and Control*, 28:125–155, 1975.

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

[3] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–283, 1989.

[4] Cao Feng. *Learning by Experimentation.* PhD thesis, The Turing Institute, University of Strathclyde, 1990.

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

[6] 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.

[7] G. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1971.

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

[9] J.R. Quinlan. Discovering rules from large collections of examples: a case study. In D. Michie, editor, *Expert Systems in the Micro-electronic Age*, pages 168–201. Edinburgh University Press, Edinburgh, 1979.

[10] J.R. Quinlan. Generating production rules from decision trees. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, pages 304–307, Los Altos, CA:, 1987. Kaufmann.

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

[12] C. Sammut and R.B Banerji. Learning concepts by asking questions. In R. Michalski, J. Carbonnel, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach. Vol. 2*, pages 167–192. Kaufmann, Los Altos, CA, 1986.

[13] E.Y. Shapiro. *Algorithmic program debugging.* MIT Press, 1983.

[14] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.