ProGolem: A System Based on Relative Minimal Generalisation

Stephen Muggleton, José Santos, and Alireza Tamaddoni-Nezhad

Department of Computing, Imperial College London
{shm,jcs06,atn}@doc.ic.ac.uk

Abstract. Over the last decade Inductive Logic Programming systems have been dominated by use of top-down refinement search techniques. In this paper we re-examine the use of bottom-up approaches to the construction of logic programs. In particular, we explore variants of Plotkin's Relative Least General Generalisation (RLGG) which are based on subsumption relative to a bottom clause. With Plotkin's RLGG, clause length grows exponentially in the number of examples. By contrast, in the Golem system, the length of ij-determinate RLGG clauses were shown to be polynomially bounded for given values of i and j. However, the determinacy restrictions made Golem inapplicable in many key application areas, including the learning of chemical properties from atom and bond descriptions. In this paper we show that with Asymmetric Relative Minimal Generalisations (or ARMGs) relative to a bottom clause, clause length is bounded by the length of the initial bottom clause. ARMGs, therefore do not need the determinacy restrictions used in Golem. An algorithm is described for constructing ARMGs and this has been implemented in an ILP system called ProGolem which combines bottom-clause construction in Progol with a Golem control strategy which uses ARMG in place of determinate RLGG. ProGolem has been evaluated on several well-known ILP datasets. It is shown that ProGolem has a similar or better predictive accuracy and learning time compared to Golem on two determinate real-world applications where Golem was originally tested. Moreover, ProGolem was also tested on several non-determinate real-world applications where Golem is inapplicable. In these applications, ProGolem and Aleph have comparable times and accuracies. The experimental results also suggest that ProGolem significantly outperforms Aleph in cases where clauses in the target theory are long and complex.

1 Introduction

There are two key tasks at the heart of ILP systems: 1) enumeration of clauses which explain one or more of the positive examples and 2) evaluation of the numbers of positive and negative examples covered by these clauses. Top-down refinement techniques such as those found in [25,22,23], use a generate-and-test approach to problems 1) and 2). A new clause is first generated by application of a refinement step and then tested for coverage of positive and negative examples.

L. De Raedt (Ed.): ILP 2009, LNAI 5989, pp. 131–148, 2010.

It has long been appreciated in AI [20] that generate-and-test procedures are less efficient than ones based on test-incorporation. The use of the bottom clause in Progol [15] represents a limited form of test-incorporation in which, by construction, all clauses in a refinement graph search are guaranteed to cover at least the example associated with the bottom clause. The use of Relative Least General Generalisation (RLGG) in Golem [13] provides an extended form of test-incorporation in which constructed clauses are guaranteed to cover a given set of positive examples. However, in order to guarantee polynomial-time construction the form of RLGG in Golem was constrained to ij-determinate clauses. Without this constraint Plotkin [21] showed that the length of RLGG clauses grows exponentially in the number of positive examples covered.

In this paper we explore variants of Plotkin's RLGG which are based on subsumption order relative to a bottom clause [28]. We give a definition for Asymmetric Relative Minimal Generalisation (ARMGs) and show that the length of ARMGs is bounded by the length of the initial bottom clause. Hence, unlike in Golem, we do not need the determinacy restrictions to guarantee polynomial-time construction. However, we show that the resulting ARMG is not unique and that the operation is asymmetric. ARMGs can easily be extended to the multiple example case by iteration. We describe an ILP system called ProGolem which combines bottom-clause construction in Progol with a Golem control strategy which uses ARMG in place of determinate RLGG. The use of top-down ILP algorithms such as Progol, tends to limit the maximum complexity of learned clauses, due to a search bias which favours simplicity. Long clauses generally require an overwhelming amount of search for systems like Progol and Aleph [27]. In this paper we also explore whether ProGolem will have any advantages in situations when the clauses in the target theory are long and complex.

ProGolem has been evaluated on several well-known ILP datasets. These include two determinate real-world applications where Golem was originally tested and several non-determinate real-world applications where Golem is inapplicable. ProGolem has also been evaluated on a set of artificially generated learning problems with large concept sizes.

The paper is arranged as follows. In Section 2 we review some of basic concepts from the ILP systems Golem and Progol which are used in the definitions and theorems in this paper. In Section 3 we discuss subsumption relative to a bottom clause. ARMG is introduced in Section 4 and some of its properties are demonstrated. An algorithm for ARMG is given in Section 5. This algorithm is implemented in the ILP system ProGolem which is described in Section 6. Empirical evaluation of ProGolem on several datasets is given in Section 7. Related work is discussed in Section 8. Section 9 concludes the paper.

2 Preliminaries

We assume the reader to be familiar with the basic concepts from logic programming and inductive logic programming [19]. This section is intended as a brief reminder of some of the concepts from the ILP systems Golem [13] and Progol [15] which are the basis for the system ProGolem described in this paper.

The general subsumption order on clauses, also known as θ-subsumption, is defined in the following.

Definition 1 (Subsumption). *Let C and D be clauses. We say C subsumes D, denoted by $C \succeq D$, if there exists a substitution θ such that $C\theta$ is a subset of D. C properly subsumes D, denoted by $C \succ D$, if $C \succeq D$ and $D \not\succeq C$. C and D are subsume-equivalent, denoted by $C \sim D$, if $C \succeq D$ and $D \succeq C$.*

Proposition 1 (Subsumption lattice). *Let $\mathcal{C}$ be a clausal language and $\succeq$ be the subsumption order as defined in Definition 1. Then the equivalence classes of clauses in $\mathcal{C}$ and the $\succeq$ order define a lattice. Every pair of clauses C and D in the subsumption lattice have a least upper bound called* least general generalisation (lgg)*, denoted by $lgg(C, D)$ and a greatest lower bound called* most general specialisation (mgs)*, denoted by $mgs(C, D)$.*

Plotkin [21] investigated the problem of finding the least general generalisation (lgg) for clauses ordered by subsumption. The notion of lgg is important for ILP since it forms the basis of generalisation algorithms which perform a bottom-up search of the subsumption lattice. Plotkin also defined the notion of relative least general generalisation of clauses ($rlgg$) which is the lgg of the clauses relative to clausal background knowledge B. The cardinality of the lgg of two clauses is bounded by the product of the cardinalities of the two clauses. However, the $rlgg$ is potentially infinite for arbitrary B. When B consists of ground unit clauses only the $rlgg$ of two clauses is finite. However the cardinality of the $rlgg$ of m clauses relative to n ground unit clauses has worst-case cardinality of order $O(n^m)$, making the construction of such $rlgg$'s intractable.

The ILP system Golem [13] is based on Plotkin's notion of relative least general generalisation of clauses ($rlgg$). Golem uses extensional background knowledge to avoid the problem of non-finite $rlggs$. Extensional background knowledge B can be generated from intensional background knowledge B' by generating all ground unit clauses derivable from B' in at most h resolution steps. The parameter h is provided by the user. The $rlggs$ constructed by Golem were forced to have only a tractable number of literals by requiring the ij-determinacy.

An ij-determinate clause is defined as follows.

Definition 2 (ij-determinate clause). *Every unit clause is $0j$-determinate. An ordered clause $h \leftarrow b_1, .., b_m, b_{m+1}, .., b_n$ is ij-determinate if and only if a) $h \leftarrow b_1, .., b_m$ is $(i-1)j$-determinate, b) every literal b_k in $b_{m+1}, .., b_n$ contains only determinate terms and has arity at most j.*

The ij-determinacy is equivalent to requiring that predicates in the background knowledge must represent functions. This condition is not met in many real-world applications, including the learning of chemical properties from atom and bond descriptions.

One of the motivations of the ILP system Progol [15] was to overcome the determinacy limitation of Golem. Progol extends the idea of inverting resolution proofs used in the systems Duce [14] and Cigol [16] and uses the general case of Inverse Entailment which is based on the model-theory which underlies proof.

Progol uses Mode-Directed Inverse Entailment (MDIE) to develop a most specific clause $\bot$ for each positive example, within the user-defined mode language, and uses this to guide an A^*-like search through clauses which subsume $\bot$.

The Progol algorithm is based on successive construction of definite clause hypotheses H from a language $\mathcal{L}$. H must explain the examples E in terms of background knowledge B. Each clause in H is found by choosing an uncovered positive example e and searching through the graph defined by the refinement ordering $\succeq$ bounded below by a bottom clause $\bot$ associated with e. In general $\bot$ can have infinite cardinality. Progol uses mode declarations to constrain the search for clauses which subsume $\bot$. Progol's mode declaration (M), definite mode language ($\mathcal{L}(M)$) and depth-bounded mode language ($\mathcal{L}_i(M)$) are defined in Appendix A.

Progol searches a bounded sub-lattice for each example e relative to background knowledge B and mode declarations M. The sub-lattice has a most general element which is the empty clause, $\square$, and a least general element $\bot$ which is the most specific element in $\mathcal{L}_i(M)$ such that $B \wedge \bot \wedge \overline{e} \vdash_h \square$ where $\vdash_h \square$ denotes derivation of the empty clause in at most h resolutions. The following definition describes a bottom clause for a depth-bounded mode language $\mathcal{L}_i(M)$.

Definition 3 (Most-specific clause or bottom clause $\bot_e$). *Let h and i be natural numbers, B be a set of Horn clauses, E be a set of positive and negative examples with the same predicate symbol a, e be a positive example in E, M be a set of mode declarations, as defined in Definitions 13, containing exactly one modeh m such that $a(m) \succeq a$, $\mathcal{L}_i(M)$ be a depth-bounded mode language as defined in Definitions 16 and $\hat{\bot}_e$ be the most-specific (potentially infinite) definite clause such that $B \wedge \hat{\bot}_e \wedge \overline{e} \vdash_h \square$. $\bot_e$ is the most-specific clause in $\mathcal{L}_i(M)$ such that $\bot_e \succeq \hat{\bot}_e$. C is the most-specific clause in $\mathcal{L}$ if for all C' in $\mathcal{L}$ we have $C' \succeq C$. $\overrightarrow{\bot}_e$ is $\bot_e$ with a defined ordering over the literals.*

In this paper, we refer to $\bot_e$ as $\overrightarrow{\bot}$ or $\bot$ depending on whether we use the ordering of the literals or not. Progol's algorithm for constructing the bottom clause is given in [15]. The Proposition below follows from Theorem 26 in [15].

Proposition 2. *Let $\bot_e$ be as defined in Definition 3, M be a set of mode declarations as defined in Definitions 13, $\mathcal{L}_i(M)$ be a depth-bounded mode language as defined in Definitions 16, i the maximum variable depth in $\mathcal{L}_i(M)$ and j be the maximum arity of any predicate in M. Then the length of $\bot_e$ is polynomially bounded in the number of mode declarations in M for fixed values of i and j.*

3 Subsumption Relative to a Bottom Clause

In a previous paper [28] we introduced a subsumption order relative to a bottom clause and demonstrated how clause refinement in a Progol-like ILP system can be characterised with respect to this order. It was shown that, unlike for the general subsumption order, efficient least general generalisation operators can

be designed for subsumption order relative to a bottom clause (i.e. $lgg_{\perp}$). In this section we briefly review the notion of subsumption order relative to bottom clause which is essential for the definition of Asymmetric Relative Minimal Generalisations (ARMGs) in this paper.

Clauses which are considered by Progol, i.e. clauses in $\mathcal{L}(M)$ (Definition 14), as well as determinate clauses considered by Golem (Definition 2), are defined with a total ordering over the literals. Moreover, the subsumption order which characterises clause refinement in a Progol-like ILP system is defined on ordered clauses. In the following we adopt an explicit representation for ordered clauses. We use the same notion used in [19] and an ordered clause is represented as a disjunction of literals (i.e. $L_1 \vee L_2 \vee \ldots \vee L_n$). The set notation (i.e. $\{L_1, L_2, \ldots, L_n\}$) is used to represent conventional clauses.

Definition 4 (Ordered clause). *An ordered clause $\overrightarrow{C}$ is a sequence of literals $L_1, L_2, \ldots, L_n$ and denoted by $\overrightarrow{C} = L_1 \vee L_2 \vee \ldots \vee L_n$. The set of literals in $\overrightarrow{C}$ is denoted by C.*

Unlike conventional clauses, the order and duplication of literals matter for ordered clauses. For example, $\overrightarrow{C} = p(X) \vee \neg q(X)$, $\overrightarrow{D} = \neg q(X) \vee p(X)$ and $\overrightarrow{E} = p(X) \vee \neg q(X) \vee p(X)$ are different ordered clauses while they all correspond to the same conventional clause, i.e. $C = D = E = \{p(X), \neg q(X)\}$.

Selection of two clauses is defined as a pair of compatible literals and this concept was used by Plotkin to define least generalisation for clauses [21]. Here we use selections to define mappings of literals between two ordered clauses.

Definition 5 (Selection and selection function). *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \ldots \vee L_n$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \ldots \vee M_m$ be ordered clauses. A selection of $\overrightarrow{C}$ and $\overrightarrow{D}$ is a pair (i, j) where L_i and M_j are compatible literals, i.e. they have the same sign and predicate symbol. A set s of selections of $\overrightarrow{C}$ and $\overrightarrow{D}$ is called a selection function if it is a total function of $\{1, 2, \ldots, n\}$ into $\{1, 2, \ldots, m\}$.*

Definition 6 (Subsequence). *Let $\overrightarrow{C} = L_1 \vee L_2 \vee \ldots \vee L_l$ and $\overrightarrow{D} = M_1 \vee M_2 \vee \ldots \vee M_m$ be ordered clauses. $\overrightarrow{C}$ is a subsequence of $\overrightarrow{D}$, denoted by $\overrightarrow{C} \sqsubseteq \overrightarrow{D}$, if there exists a strictly increasing selection function $s \subseteq \{1, \ldots, l\} \times \{1, \ldots, m\}$ such that for each $(i, j) \in s$, $L_i = M_j$.*

Example 1. Let $\overrightarrow{B} = p(x, y) \vee q(x, y) \vee r(x, y) \vee r(y, x)$, $\overrightarrow{C} = p(x, y) \vee r(x, y) \vee r(y, x)$ and $\overrightarrow{D} = p(x, y) \vee r(y, x) \vee r(x, y)$ be ordered clauses. $\overrightarrow{C}$ is a subsequence of $\overrightarrow{B}$ because there exists increasing selection function $s_1 = \{(1, 1), (2, 3), (3, 4)\}$ which maps literals from $\overrightarrow{C}$ to equivalent literals from $\overrightarrow{D}$. However, $\overrightarrow{D}$ is not a subsequence of $\overrightarrow{B}$ because an increasing selection function does not exist for $\overrightarrow{D}$ and $\overrightarrow{B}$. $\Diamond$

As shown in [28], clause refinement in Progol-like ILP systems cannot be described by the general subsumption order. However, subsumption order relative to $\perp$ (i.e. $\succeq_{\perp}$) can capture clause refinement in these systems. In the following we first define $\overrightarrow{\mathcal{L}}^{s}_{\perp}$ which can be used to represent the hypotheses language of a Progol-like ILP system.

Definition 7 ($\overrightarrow{\mathcal{L}}^s_\perp$). *Let $\overrightarrow{\perp}$ be the bottom clause as defined in Definition 3 and $\overrightarrow{C}$ a definite ordered clause. $\overrightarrow{\top}$ is $\overrightarrow{\perp}$ with all variables replaced with new and distinct variables. $\theta_\top$ is a variable substitution such that $\overrightarrow{\top}\theta_\top = \overrightarrow{\perp}$. $\overrightarrow{C}$ is in $\overrightarrow{\mathcal{L}}^s_\perp$ if $\overrightarrow{C}\theta_\top$ is a subsequence of $\overrightarrow{\perp}$.*

Example 2. Let $\overrightarrow{\perp} = p(X) \leftarrow q(X), r(X), s(X,Y), s(Y,X)$ and according to Definition 7, we have $\overrightarrow{\top} = p(V_1) \leftarrow q(V_2), r(V_3), s(V_4,V_5), s(V_6,V_7)$ and $\theta_\top = \{V_1/X, V_2/X, V_3/X, V_4/X, V_5/Y, V_6/Y, V_7/X\}$. Then $\overrightarrow{C}$=$p(V_1)$←$r(V_2), s(V_6,V_7)$, $\overrightarrow{D} = p(V_1) \leftarrow r(V_1), s(V_6,V_1)$ and $\overrightarrow{E} = p(V_1) \leftarrow r(V_1), s(V_4,V_5)$ are in $\overrightarrow{\mathcal{L}}^s_\perp$ as $\overrightarrow{C}\theta_\top$, $\overrightarrow{D}\theta_\top$ and $\overrightarrow{E}\theta_\top$ are subsequences of $\overrightarrow{\perp}$. ◊

Definition 8 (Subsumption relative to $\perp$). *Let $\overrightarrow{\perp}$, $\theta_\top$ and $\overrightarrow{\mathcal{L}}^s_\perp$ be as defined in Definition 7 and $\overrightarrow{C}$ and $\overrightarrow{D}$ be ordered clauses in $\overrightarrow{\mathcal{L}}^s_\perp$. We say $\overrightarrow{C}$ subsumes $\overrightarrow{D}$ relative to $\perp$, denoted by $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$, if $\overrightarrow{C}\theta_\top$ is a subsequence of $\overrightarrow{D}\theta_\top$. $\overrightarrow{C}$ is a proper generalisation of $\overrightarrow{D}$ relative to $\perp$, denoted by $\overrightarrow{C} \succ_\perp \overrightarrow{D}$, if $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$ and $\overrightarrow{D} \not\succeq_\perp \overrightarrow{C}$. $\overrightarrow{C}$ and $\overrightarrow{D}$ are equivalent with respect to subsumption relative to $\perp$, denoted by $\overrightarrow{C} \sim_\perp \overrightarrow{D}$, if $\overrightarrow{C} \succeq_\perp \overrightarrow{D}$ and $\overrightarrow{D} \succeq_\perp \overrightarrow{C}$.*

Example 3. Let $\overrightarrow{\perp}$, $\theta_\top$, $\overrightarrow{\mathcal{L}}^s_\perp$, $\overrightarrow{C}$, $\overrightarrow{D}$ and $\overrightarrow{E}$ be as in Example 2. Then, $\overrightarrow{C}$ subsumes $\overrightarrow{D}$ relative to $\perp$ since $\overrightarrow{C}\theta_\top$ is a subsequence of $\overrightarrow{D}\theta_\top$. However, $\overrightarrow{C}$ does not subsume $\overrightarrow{E}$ relative to $\perp$ since $\overrightarrow{C}\theta_\top$ is not a subsequence of $\overrightarrow{E}\theta_\top$. Note that $\overrightarrow{C}$ subsumes $\overrightarrow{E}$ with respect to normal subsumption. ◊

The following Proposition is a special case of Lemma 5 in [28] and follows directly from Definition 8.

Proposition 3. *Let $\overrightarrow{\perp}$ be as defined in Definition 3 and $\overrightarrow{C}$ be an ordered clause obtained from $\overrightarrow{\perp}$ by removing some literals without changing the order of the remaining literals. Then, $\overrightarrow{C} \succ_\perp \overrightarrow{\perp}$.*

The subsumption order relative to $\perp$ was studied in [28]. It was shown that the refinement space of a Progol-like ILP system can be characterised using $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$. It was also shown that $\langle \overrightarrow{\mathcal{L}}_\perp, \succeq_\perp \rangle$ is a lattice which is isomorphic to an atomic lattice and that the most general specialisation relative to $\perp$ ($mgs_\perp$) and the least general generalisation relative to $\perp$ ($lgg_\perp$) can be defined based on the most general specialisation and the least general generalisation for atoms.

4 Asymmetric Relative Minimal Generalisations

The construction of the least general generalisation (lgg) of clauses in the general subsumption order is inefficient as the cardinality of the lgg of two clauses can grow very rapidly (see Section 2). For example, with Plotkin's Relative Least General Generalisation (RLGG), clause length grows exponentially in the number of examples [21]. Hence, an ILP system like Golem [13] which uses RLGG is constrained to ij-determinacy to guarantee polynomial-time construction. However, the determinacy restrictions make an ILP system inapplicable in many key

application areas, including the learning of chemical properties from atom and bond descriptions. On the other hand, as shown in [28], efficient operators can be implemented for least generalisation and greatest specialisation in the subsumption order relative to a bottom clause. In this section we define a variant of Plotkin's RLGG which is based on subsumption order relative to a bottom clause and does not need the determinacy restrictions. The relative least general generalisation ($lgg_\perp$) in [28] is defined for a lattice bounded by a bottom clause $\perp_e$. This bottom clause is constructed with respect to a single positive example e and as in Progol we need a search guided by coverage testing to explore the hypotheses space. However, the asymmetric relative minimal generalisation (ARMG) described in this paper is based on pairs of positive examples and as in Golem, by construction it is guaranteed to cover all positive examples which are used to construct it. Hence, ARMGs have the same advantage as RLGGs in Golem but unlike RLGGs the length of ARMGs is bounded by the length of $\perp_e$. The asymmetric relative minimal generalisation of examples e' and e relative to $\perp_e$ is denoted by $armg_\perp(e'|e)$ and in general $armg_\perp(e'|e) \neq armg_\perp(e|e')$. In the following we define asymmetric relative minimal generalisation and study some of their properties. It is normal in ILP to restrict attention to clausal hypotheses which are "head-connected" in the following sense.

Definition 9 (Head-connectness). *A definite ordered clause $h \leftarrow b_1, .., b_n$ is said to be head-connected if and only if each body atom b_i contains at least one variable found either in h or in a body atom b_j, where $1 \leq j < i$.*

Definition 10 (Asymmetric relative common generalisation). *Let E, B and $\perp_e$ be as defined in Definition 3, e and e' be positive examples in E and $\overrightarrow{C}$ is a head-connected definite ordered clause in $\overrightarrow{\mathcal{L}}^s_\perp$. $\overrightarrow{C}$ is an asymmetric common generalisation of e' and e relative to $\perp_e$, denoted by $\overrightarrow{C} \in arcg_\perp(e'|e)$, if $\overrightarrow{C} \succeq_\perp \perp_e$ and $B \wedge C \vdash e'$.*

Example 4. Let $M = \{p(+), q(+,-), r(+,-)\}$ be mode definition, $B = \{q(a,a), r(a,a), q(b,b), q(b,c), r(c,d)\}$ be background knowledge and $e = p(a)$ and $e' = p(b)$ be positive examples. Then we have $\perp_e = p(X) \leftarrow q(X,X), r(X,X)$ and clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1,V_1)$, $\overrightarrow{D} = p(V_1) \leftarrow q(V_1,V_3), r(V_3,V_5)$ and $\overrightarrow{E} = p(V_1) \leftarrow q(V_1,V_3)$ are all in $arcg_\perp(e'|e)$. ◊

Definition 11 (Asymmetric relative minimal generalisation). *Let E and $\perp_e$ be as defined in Definition 3, e and e' be positive examples in E and $arcg_\perp(e'|e)$ be as defined in Definition 10. $\overrightarrow{C}$ is an asymmetric minimal generalisation of e' and e relative to $\perp_e$, denoted by $\overrightarrow{C} \in armg_\perp(e'|e)$, if $\overrightarrow{C} \in arcg_\perp(e'|e)$ and $\overrightarrow{C} \succeq_\perp \overrightarrow{C'} \in arcg_\perp(e'|e)$ implies $\overrightarrow{C}$ is subsumption-equivalent to $\overrightarrow{C'}$ relative to $\perp_e$.*

Example 5. Let B, $\perp_e$, e and e' be as in Example 4. Then clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1,V_1)$ and $\overrightarrow{D} = p(V_1) \leftarrow q(V_1,V_3), r(V_3,V_5)$ are both in $armg_\perp(e'|e)$. ◊

This example shows that ARMGs are not unique.

Theorem 1. *The set $armg_\perp(e'|e)$ can contain more than one clause which are not subsumption-equivalent relative to $\perp_e$.*

Proof. In Example 4, clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1, V_1)$ and $\overrightarrow{D} = p(V_1) \leftarrow q(V_1, V_3), r(V_3, V_5)$ are both in $armg_\perp(e'|e)$ but not subsumption-equivalent relative to $\perp_e$. □

The following theorem shows that the length of ARMG is bounded by the length of $\perp_e$.

Theorem 2. *For each $\overrightarrow{C} \in armg_\perp(e'|e)$ the length of $\overrightarrow{C}$ is bounded by the length of $\perp_e$.*

Proof. Let $\overrightarrow{C} \in armg_\perp(e'|e)$. Then by definition $\overrightarrow{C} \succeq_\perp \perp_e$ and according to Definition 8, $\overrightarrow{C}$ is a subsequence of $\perp_e$. Hence, the length of $\overrightarrow{C}$ is bounded by the length of $\perp_e$. □

It follows from Theorem 2 that the number of literals in an ARMG is bounded by the length of $\perp_e$, which according to Proposition 2 is polynomially bounded in the number of mode declarations for fixed values of i and j, where i is the maximum variable depth and j is the maximum arity of any predicate in M. Hence, unlike the RLGGs used in Golem, ARMGs do not need the determinacy restrictions and can be used in a wider range of problems including those which are non-determinate. In Section 7 we apply ARMGs to a range of determinate and non-determinate problems and compare the results with Golem and Aleph. But first we give an algorithm for constructing ARMGs in Section 5 and describe an implementation of ARMGs in Section 6.

5 Algorithm for ARMGs

It was shown in the previous section that ARMGs do not have the limitations of RLGGs and that the length of ARMGs is bounded by the length of $\perp_e$. In this section we show that there is also an efficient algorithm for constructing ARMGs. The following definitions are used to describe the ARMG algorithm.

Definition 12 (Blocking atom). *Let B be background knowledge, E^+ the set of positive examples, $e \in E^+$ and $\overrightarrow{C} = h \leftarrow b_1, \ldots, b_n$ be a definite ordered clause. b_i is a blocking atom if and only if i is the least value such that for all θ, $e = h\theta, B \nvdash (b_1, \ldots, b_i)\theta$.*

An algorithm for constructing ARMGs is given in Figure 1. Given the bottom clause $\perp_e$ associated with a particular positive example e, this algorithm works by dropping a minimal set of atoms from the body to allow coverage of a second example. Below we prove the correctness of the ARMG algorithm.

Theorem 3 (Correctness of ARMG algorithm). *Let E and $\perp_e$ be as defined in Definition 3, e and e' be positive examples in E, $armg_\perp(e'|e)$ be as defined in Definition 11 and $ARMG(\perp_e, e')$ as given in Figure 1. Then $\overrightarrow{C} = ARMG(\perp_e, e')$ is in $armg_\perp(e'|e)$.*

Asymmetric Relative Minimal Generalization (ARMG) Algorithm
Input: Bottom clause $\bot_e$, Positive example e'
$\overrightarrow{C}$ is $\bot_e = h \leftarrow b_1, .., b_n$
While there is a blocking atom b_i wrt e' in the body of $\overrightarrow{C}$
Remove b_i from $\overrightarrow{C}$
Remove atoms from $\overrightarrow{C}$ which are not head-connected
Repeat
Output: $\overrightarrow{C}$

Fig. 1. ARMG algorithm

Proof. Assume $\overrightarrow{C} \notin armg_\bot(e'|e)$. In this case, either $\overrightarrow{C}$ is not an asymmetric common generalisation of e and e' or it is not minimal. However, by construction $\overrightarrow{C}$ is a subsequence of $\bot_e$ in which all blocking literals with respect to e' are removed. Then according to Proposition 3, $\overrightarrow{C} \succeq_\bot \bot_e$ and by construction $B \wedge C \vdash e'$. Hence, $\overrightarrow{C}$ is an asymmetric common generalisation of e and e'. So, $\overrightarrow{C}$ must be non-minimal. If $\overrightarrow{C}$ is non-minimal then $\overrightarrow{C} \succeq_\bot \overrightarrow{C'}$ for $\overrightarrow{C'} \in armg_\bot(e'|e)$ which must either have literals not found in $\overrightarrow{C}$ or there is a substitution θ such that $\overrightarrow{C}\theta = \overrightarrow{C'}$. But we have deleted the minimal set of literals. This is a minimal set since leaving a blocking atom would mean $B \wedge C \nvdash e'$ and leaving a non-head-connected literal means $\overrightarrow{C} \notin armg_\bot(e'|e)$. So it must be the second case. However, in the second case θ must be a renaming since the literals in $\overrightarrow{C}$ are all from $\bot_e$. Hence, $\overrightarrow{C}$ and $\overrightarrow{C'}$ are variants which contradicts the assumption and completes the proof. □

The following example shows that the ARMGs algorithm is not complete.

Example 6. Let B, $\bot_e$, e and e' be as in Example 4. Then clauses $\overrightarrow{C} = p(V_1) \leftarrow q(V_1, V_1)$ and $\overrightarrow{D} = p(V_1) \leftarrow q(V_1, V_3), r(V_3, V_5)$ are both in $armg_\bot(e'|e)$. However, the ARMGs algorithm given in Figure 1 cannot generate clause $\overrightarrow{D}$. ◊

Example 6 shows that the ARMGs algorithm does not consider hypotheses which require 'variable splitting'. As shown in [28] (Example 2), there are some group of problems which cannot be learned by a Progol-like ILP system without variable splitting. The concept of variable splitting and the ways it has been done in Progol and Aleph were discussed in [28]. Similar approaches could be adopted for ProGolem, however, the current implementation does not support variable splitting.

Figure 2 gives a comparison between Golem's determinate RLGG and the ARMGs generated by the ARMG algorithm on Michalski's trains dataset from [12]. Note that Golem's RLGG cannot handle the predicate *has_car* because it is non-determinate. The first ARMG (2) subsumes the target concept which is eastbound(A) ← has_car(A,B), closed(B), short(B). Note that in this example RLGG (1) is shorter than ARMGs (2,3) since it only contains determinant literals.

1. $RLGG(e_1, e_2)$ = $RLGG(e_2, e_1)$ = eastbound(A) ← infront(A,B), short(B), open(B), shape(B,C), load(B,triangle,1), wheels(B,2), infront(B,D), shape(D, rectangle), load(D,E,1), wheels(D,F), infront(D,G), closed(G), short(G), shape(G,H), load(G,I,1), wheels(G,2).
2. $ARMG(\bot_{e_1}, e_2)$ = eastbound(A) ← has_car(A,B), has_car(A,C), has_car(A,D), has_car(A,E), infront(A,E), closed(C), short(B), short(C), short(D), short(E), open(B), open(D), open(E), shape(B,F), shape(C,G), shape(D,F), shape(E,H), load(D,I,J),2), wheels(E,2)
3. $ARMG(\bot_{e_2}, e_1)$ = eastbound(A) ← has_car(A,B), has_car(A,C), has_car(A,D), infront(A,D), closed(C), short(B), short(D), open(D), shape(B,E), shape(D,E), load(B,F,G), load(D,H,G), wheels(B,2), wheels(D,2)

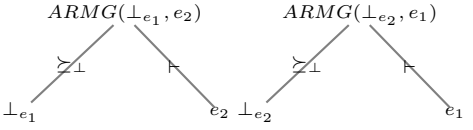


Fig. 2. A comparison between Golem's determinate RLGG (1) and the non-determinate ARMGs (2,3). Note that Golem's RLGG cannot handle the predicate *has_car* because it is non-determinate. The first ARMG (2) subsumes the target concept which is eastbound(A) ← has_car(A,B), closed(B), short(B).

6 Implementation

This section describes ProGolem's implementation. As in Golem and Progol, ProGolem uses the cover set approach to construct a theory consisting of more than one clause. ProGolem's cover set algorithm is shown in Fig. 3. This algorithm repeatedly constructs a clause from a set of best ARMGs, uses negative examples to reduce the clause (see below), adds this clause to the theory and removes the positive examples which are covered.

As in Golem, ProGolem uses negative-based reduction to generalise clauses with respect to negative examples. This algorithm is described in section 6.1. ProGolem uses a greedy beam search to select the best ARMG with respect to $\bot_e$. This algorithm is shown in Fig. 4. The basic idea is to repeatedly extend ARMGs using positive examples and keep the best ARMGs at each iteration to

ProGolem's Cover Set Algorithm
Input: Examples E, mode declarations M, background knowledge B

Let $T = \{\}$
Let S = all positive examples in E
While $S \neq \{\}$
  Let e be the first example in S
  Construct the bottom clause $\bot_e$ from e, M and B ([15])
  Let $\overrightarrow{C}$ = Best_ARMG($\bot_e$, E) (see Fig. 4)
  Let $\overrightarrow{C'}$ = Negative_based_reduction($\overrightarrow{C}$, E) (see section 6.1)
  $T = T \cup \overrightarrow{C'}$
  Let S' = all examples from S which are covered by $\overrightarrow{C'}$
  $S = S - S'$
Repeat
Output: T

Fig. 3. ProGolem's cover set algorithm

```
Best ARMG Algorithm
    Input: ⊥_e, Examples E
     sample size K, beam width N

      Let best_armgs = {⊥_e}
      Repeat
        Let best_score = highest score from best_armgs
        Let Ex = K random positive examples from E
        Let new_armgs = {}
        for each C ∈ best_armgs do
          for each e' ∈ Ex do
            Let C' = ARMG(C, e') (see Fig. 1)
            if score(C') > best_score then
              new_armgs = new_armgs ∪ C'
            end if
          end for
        end for
        if (new_armgs ≠ {}) then
          best_armgs = highest scoring N clauses from new_armgs
      Until new_armgs = {}
    Output: highest scoring clause from best_armgs
```

Fig. 4. Best ARMG algorithm

be extended in the next iteration until the ARMGs' score no longer increases. The score of an ARMG is computed in the same way a normal clause is evaluated in ProGolem. The evaluation function can be selected by the user (e.g. compression, accuracy, precision, coverage). By default it is compression, that is, the positives covered minus negatives covered minus length of the clause (i.e. its number of literals). At each iteration and for each ARMG in the set of ARMGs under consideration, K examples which are not covered by the current ARMG are selected and used to extend it. The best N (beam width) ARMGs of each iteration are selected to be used as the initial set for the next iteration. The initial set of ARMGs at iteration 0 is the bottom clause $\bot_e$. K and N are user defined parameters with default values of 10 and 2 respectively.

ProGolem is a bottom-up ILP system and unlike in a top-down system such as Progol, the intermediate clauses considered may be very long. The coverage testing of long non-determinate clauses in ProGolem could be very inefficient as it involves a large number of backtracking. Note that clauses considered by Golem are also relatively long but these clauses are determinate which makes the subsumption testing less expensive. In order to address this problem, efficient subsumption testing algorithms are implemented in ProGolem. The following sections describe the negative-based clause reduction and the efficient coverage testing algorithms.

6.1 Negative-Based Clause Reduction

ProGolem implements a negative-based clause reduction algorithm which is similar to the reduction algorithms used in QG/GA [18] and Golem [13]. The aim of negative-based reduction is to generalise a clause by keeping only literals which block negative examples from being proved. The negative-based reduction

algorithm works as follows. Given a clause $h \leftarrow b_1, \ldots, b_n$, find the first literal, b_i such that the clause $h \leftarrow b_1, \ldots, b_i$ covers no negative examples. Prune all literals after b_i and move b_i and all its supporting literals to the front, yielding a clause $h \leftarrow S_i, b_i, T_i$, where S_i is a set of supporting literals needed to introduce the input variables of b_i and T_i is $b_1, \ldots, b_{i-1}$ with S_i removed. Then reduce this new clause in the same manner and iterate until the clause length remains the same within a cycle.

6.2 Efficient Coverage Testing

The intermediate clauses considered by ProGolem can be non-determinate and very long. Prolog's standard left-to-right depth-first literal evaluation is extremely inefficient for testing the coverage of such clauses. An efficient algorithm for testing the coverage of long non-determinant clauses is implemented in ProGolem. This algorithm works by selecting at each moment the literal which has fewest solutions, from the ones which had their input variables instantiated. This algorithm was further improved by an approach inspired by constraint satisfaction algorithms of [11] and [9]. The core idea is to enumerate variables (rather than literals as before) based on the ones which have the smallest domains. The domain of a variable is the intersection of all the values a variable can assume in the literals it appears. This works well because normally clauses have much fewer distinct variables than literals.

7 Empirical Evaluation

In this section we evaluate ProGolem on several well-known determinate and non-determinate ILP datasets and compare the results with Golem and Aleph. Aleph [27] is a well-known ILP system which works in different modes and can emulate the functionality of several other ILP systems including Progol. ProGolem and Aleph are both implemented in YAP Prolog which makes the time comparison between them more accurate.

The materials to reproduce the experiments in this section, including datasets and programs are available from http://ilp.doc.ic.ac.uk/ProGolem/.

7.1 Experiment 1 – Determinate and Non-determinate Applications

Materials and Methods. Several well-known ILP datasets have been used: Proteins [17], Pyrimidines [6], DSSTox [24], Carcinogenesis [26], Metabolism [5] and Alzheimers-Amine [7]. The two determinate datasets, Proteins and Pyrimidines, were used with a hold out test strategy. The data split between training and test sets was done by considering 4/5 for Proteins and 2/3 for Pyridimines as training data and the remaining for test. For the Carcinogenesis, Metabolism and Alzheimers-Amine datasets a 10-fold cross-validation was performed and for DSSTox it was a 5-fold cross validation. Whenever cross-validation was used the accuracy's standard deviation over all the folds is also reported. Both Aleph and ProGolem were executed in YAP Prolog version 6 with $i = 2$,

Table 1. Predictive accuracies and learning times for *Golem*, *ProGolem* and *Aleph* on different datasets. Golem can only be applied on determinate datasets, i.e. Proteins and Pyrimidines.

dataset	*Golem*		*ProGolem*		*Aleph*	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
Alz-Amine	N/A	N/A	76.1±4.4	36	76.2±3.8	162
Carcino	N/A	N/A	63.0±7.2	649	59.7±6.3	58
DSSTox	N/A	N/A	68.6±4.5	993	72.6±6.9	239
Metabolism	N/A	N/A	63.9±11.6	691	62.1±6.2	32
Proteins	62.3	3568	62.3	2349	50.5	4502
Pyrimidines	72.1	68	75.3	19	73.7	23

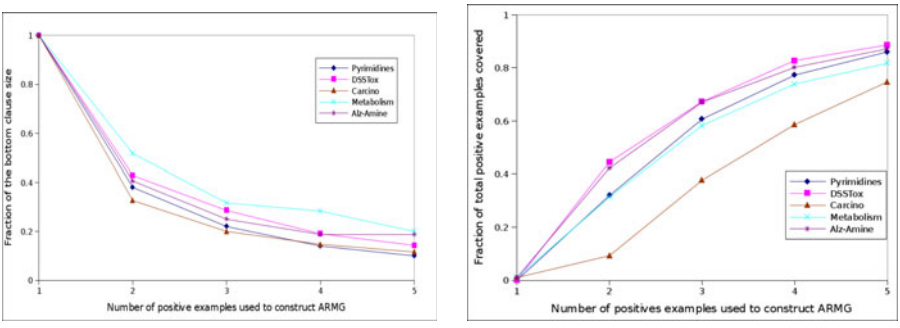
$maxneg = 10$ (except for Carcinogenesis and Proteins where $maxneg = 30$) and $evalfn = compression$ (except for DSSTox where $evalfn = coverage$). Aleph was executed with $nodes = 1000$ and $clauselength = 5$ (except for Proteins where $nodes = 10000$ and $clauselength = 40$). ProGolem was executed with $N = 2$ (beam-width) and $K = 5$ (sample size at each iteration). ProGolem's coverage testing was Prolog's standard left-to-right strategy on all these datasets (the same as Aleph). All experiments were performed on a 2.2 Ghz dual core AMD Opteron processor (275) with 8gb RAM.

Results and discussion. Table 1 compares predictive accuracies and average learning times for Golem, ProGolem and Aleph. ProGolem is competitive with Golem on the two determinate datasets. On the Proteins dataset which requires learning long target concepts, Aleph cannot generate any compressive hypothesis and is slower. This is the type of problems where a bottom-up ILP system has an advantage over a top-down one. Golem is inapplicable on the remaining non-determinate problems and ProGolem and Aleph have comparable times and accuracies.

Fig. 5 compares the length and positive coverage of ARMGs in ProGolem. In Fig. 5.(a) the ARMG length (as a fraction of the bottom clause size) is plotted against the number of examples used to construct the ARMG. In Fig. 5.(b) the ARMG positive coverage is plotted against the same X axis. For number of examples equal to 1, the ARMG (i.e. bottom clause) coverage is almost invariably the example which has been used to construct the bottom clause and has the maximum length. The coverage increases with the number of examples used to construct the ARMGs. The ARMGs clause lengths follow an exponential decay and, symmetrically, the positive coverage has an exponential growth since shorter clauses are more general.

7.2 Experiment 2 – Complex Artificial Target Concepts

The results of Experiment 1 suggested that for the Proteins dataset which requires learning long clauses, the performance of Aleph is significantly worse than Golem and ProGolem. In this experiment we further examine whether ProGolem will have any advantages in situations when the clauses in the target theory are long and complex.



(a) (b)

Fig. 5. (a) ARMGs length and (b) ARMGs positive coverage as number of examples used to construct the ARMGs increases

Materials and Methods. In this experiment we use a set of eight artificially generated learning problems with varying concept sizes from 6 to 17. These problems are selected from the phase transition study [4] and correspond to problems $m6.l12$ to $m17.l12$. There are two parameters that characterise each problem in the dataset: m the target concept size, and L the number of distinct constants occurring in an example. These problems are selected from the first row of the (m, L) plane, i.e. $L = 12$ so that they only approach the phase transition region. Each problem has 200 training and 200 test examples and the positive and negative examples are balanced in both partitions (100 for each). We use a hold-out test strategy and compare the performance of ProGolem and Aleph. This dataset is noise free and highly non-determinate having about 100 solutions per predicate. In order to curb the combinatorial explosion the recall for constructing the bottom clauses was set to 2 for both Aleph and ProGolem.

Aleph and ProGolem were executed with $evalfn = compression$, $i = 2$, $noise = 0$. Other Aleph parameters are $clauselength = 20$, $search = heuristic$ and $nodes = 100,000$. ProGolem was used with $N = 2$, $K = 10$ and the efficient coverage testing algorithm was used in place of Prolog's left-to-right evaluation. All the experiments were performed, as before, on a 2.2 Ghz dual core AMD Opteron processor (275) with 8gb RAM.

Results and discussion. Table 2 shows predictive accuracies and average learning times for ProGolem and Aleph. Aleph fails to find any solution in four out of eight problems whereas ProGolem can find good approximations of the target concepts. Moreover, ProGolem is significantly faster than Aleph. This is partly because long clauses generally require an overwhelming amount of search in top-down systems like Progol and Aleph, due to a search bias which favours simplicity. This tends to limit the maximum complexity of learned clauses. Note that Golem is inapplicable on this phase transition dataset as it is non-determinate.

Table 2. Predictive accuracies and learning times for *ProGolem* and *Aleph* on a set of learning problems with varying concept sizes from 6 to 17

m	*ProGolem*		*Aleph*	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
6	98.0	7	99.5	1
7	99.5	15	99.5	254
8	97.5	40	100	23
10	97.0	45	50.0	3596
11	99.0	36	50.0	3708
14	93.5	47	96.0	365
16	76.0	501	50.0	4615
17	71.0	485	50.0	4668

8 Related Work and Discussion

It was shown in a previous paper [28] that, unlike for the general subsumption order, efficient least generalisation operators can be designed for the subsumption order relative to a bottom clause. This idea is the basis for ProGolem which implements efficient asymmetric relative minimal generalisations for the subsumption order relative to a bottom clause. The lattice structure and refinement operators for the subsumption order relative to a bottom clause were studied in [28]. The relationship between this subsumption order and some of related subsumption orders including weak subsumption [2], ordered subsumption [8] and sequential subsumption in SeqLog [10] were also discussed.

The least and minimal generalisations relative to a bottom clause can be compared with other approaches which use lgg-like operators but instead of considering all pairs of compatible literals they only consider one pair. For example, LOGAN-H [1] is a bottom-up system which is based on inner products of examples which are closely related to lgg operator. This system constructs lgg-like clauses by considering only those pairs of literals which guarantee an injective mapping between variables. In other words, it assumes one-one object mappings. Other similar approaches use the same idea of simplifying the lgg-like operations by considering only one pair of compatible literals but they select this pair arbitrarily (e.g. [3]).

As already mentioned in the previous sections, ProGolem is closely related to Golem which is based on generalisation relative to background knowledge B. ProGolem is based on generalisation relative to a bottom clause $\bot_e$ which is the result of compiling background knowledge B. Hence, subsumption relative to a bottom clause can be viewed as subsumption relative to a compilation of B which makes it more efficient than subsumption relative to B. Moreover, as already shown in this paper, generalisation relative to a bottom clause allows ProGolem to be used for non-determinate problems where Golem is inapplicable.

9 Conclusions

In this paper we have proposed an asymmetric variant of Plotkin's RLGG, called ARMG. In comparison to the determinate RLGGs used in Golem, ARMGs are

capable of representing non-determinate clauses. Although this is also possible using Plotkin's RLGG, the cardinality of the Plotkin RLGG grows exponentially in the number of examples. By contrast, an ARMG is built by constructing a bottom clause for one example and then dropping a minimal set of literals to allow coverage of a second example. By construction the clause length is bounded by the length of the initially constructed bottom clause.

An algorithm is described for constructing ARMGs and this has been implemented in an ILP system called ProGolem which combines bottom-clause construction in Progol with a Golem control strategy which uses ARMG in place of determinate RLGG. It is shown that ProGolem has a similar or better predictive accuracy and learning time compared to Golem on two determinate real-world applications where Golem was originally tested. Moreover, ProGolem was also tested on several non-determinate real-world applications where Golem is inapplicable. In these applications, ProGolem and Aleph have comparable times and accuracies. ProGolem has also been evaluated on a set of artificially generated learning problems with large concept sizes. The experimental results suggest that ProGolem significantly outperforms Aleph in cases where clauses in the target theory are long and complex. These results suggest that while ProGolem has the advantages of Golem for learning large target concepts, it does not suffer from the determinacy limitation and can be used in problems where Golem is inapplicable.

The use of top-down ILP algorithms such as Progol, tends to limit the maximum complexity of learned clauses, due to a search bias which favours simplicity. Long target clauses generally require an overwhelming amount of search for systems like Progol and Aleph. We believe that such targets should be more effectively learned by a bottom-up systems such as ProGolem since long clauses are easier to construct using bottom-up search.

Acknowledgments

We would like to thank Gerson Zaverucha, Erick Alphonse and Filip Zelezny for their helpful discussions and advice on different subjects related to this paper. The first author thanks the Royal Academy of Engineering and Microsoft for funding his present 5 year Research Chair. The second author was supported by a Wellcome Trust Ph.D. scholarship. The third author was supported by the BBSRC grant BB/C519670/1.

References

1. Arias, M., Khardon, R.: Bottom-up ILP using large refinement steps. In: Camacho, R., King, R., Srinivasan, A. (eds.) ILP 2004. LNCS (LNAI), vol. 3194, pp. 26–43. Springer, Heidelberg (2004)
2. Badea, L., Stanciu, M.: Refinement operators can be (weakly) perfect. In: Džeroski, S., Flach, P.A. (eds.) ILP 1999. LNCS (LNAI), vol. 1634, pp. 21–32. Springer, Heidelberg (1999)

3. Basilio, R., Zaverucha, G., Barbosa, V.C.: Learning logic programs with neural networks. In: Rouveirol, C., Sebag, M. (eds.) ILP 2001. LNCS (LNAI), vol. 2157, pp. 15–26. Springer, Heidelberg (2001)
4. Botta, M., Giordana, A., Saitta, L., Sebag, M.: Relational learning as search in a critical region. J. Mach. Learn. Res. 4, 431–463 (2003)
5. Cheng, J., Hatzis, C., Hayashi, H., Krogel, M., Morishita, S., Page, D., Sese, J.: Kdd cup 2001 report. SIGKDD Explorations 3(2), 47–64 (2002)
6. King, R.D., Muggleton, S.H., Lewis, R., Sternberg, M.: Drug design by machine learning. Proceedings of the National Academy of Sciences 89(23), 11322–11326 (1992)
7. King, R.D., Srinivasan, A., Sternberg, M.J.E.: Relating chemical activity to structure: an examination of ILP successes. New Generation Computing 13, 411–433 (1995)
8. Kuwabara, M., Ogawa, T., Hirata, K., Harao, M.: On generalization and subsumption for ordered clauses. In: Washio, T., Sakurai, A., Nakajima, K., Takeda, H., Tojo, S., Yokoo, M. (eds.) JSAI Workshop 2006. LNCS (LNAI), vol. 4012, pp. 212–223. Springer, Heidelberg (2006)
9. Kuzelka, O., Železný, F.: Fast estimation of first-order clause coverage through randomization and maximum likelihood. In: Proceedings of the 25th International Conference (ICML 2008), pp. 504–511 (2008)
10. Lee, S.D., De Raedt, L.: Constraint Based Mining of First Order Sequences in SeqLog. In: Database Support for Data Mining Applications, pp. 155–176 (2003)
11. Maloberti, J., Sebag, M.: Fast theta-subsumption with constraint satisfaction algorithms. Machine Learning 55(2), 137–174 (2004)
12. Muggleton, S.: Progol datasets (1996), `http://www.doc.ic.ac.uk/~shm/software/progol4.2/`
13. Muggleton, S., Feng, C.: Efficient induction of logic programs. In: Muggleton, S. (ed.) Inductive Logic Programming, pp. 281–298. Academic Press, London (1992)
14. Muggleton, S.H.: Duce, an oracle based approach to constructive induction. In: IJCAI 1987, pp. 287–292. Kaufmann, San Francisco (1987)
15. Muggleton, S.H.: Inverse entailment and Progol. New Generation Computing 13, 245–286 (1995)
16. Muggleton, S.H., Buntine, W.: Machine invention of first-order predicates by inverting resolution. In: Proceedings of the 5th International Conference on Machine Learning, pp. 339–352. Kaufmann, San Francisco (1988)
17. Muggleton, S.H., King, R., Sternberg, M.: Protein secondary structure prediction using logic-based machine learning. Protein Engineering 5(7), 647–657 (1992)
18. Muggleton, S.H., Tamaddoni-Nezhad, A.: QG/GA: A stochastic search for Progol. Machine Learning 70(2-3), 123–133 (2007), doi:10.1007/s10994-007-5029-3
19. Nienhuys-Cheng, S.-H., de Wolf, R.: Foundations of Inductive Logic Programming. LNCS (LNAI), vol. 1228, pp. 168–169. Springer, Heidelberg (1997)
20. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga, Palo Alto (1980)
21. Plotkin, G.D.: Automatic Methods of Inductive Inference. PhD thesis, Edinburgh University (August 1971)
22. Quinlan, J.R.: Learning logical definitions from relations. Machine Learning 5, 239–266 (1990)
23. De Raedt, L., Bruynooghe, M.: A theory of clausal discovery. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, San Francisco (1993)
24. Richard, A.M., Williams, C.R.: Distributed structure-searchable toxicity (DSSTox) public database network: A proposal. Mutation Research 499, 27–52 (2000)

25. Shapiro, E.Y.: Algorithmic program debugging. MIT Press, Cambridge (1983)
26. Srinivasan, A., King, R.D., Muggleton, S.H., Sternberg, M.: Carcinogenesis predictions using ILP. In: Džeroski, S., Lavrač, N. (eds.) ILP 1997. LNCS, vol. 1297, pp. 273–287. Springer, Heidelberg (1997)
27. Srinivasan, A.: The Aleph Manual. University of Oxford (2007)
28. Tamaddoni-Nezhad, A., Muggleton, S.H.: The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. Machine Learning 76(1), 37–72 (2009)

Appendix A Progol's Definite Mode Language

The following definitions describe Progol's mode declaration (M), definite mode language ($\mathcal{L}(M)$) and depth-bounded mode language ($\mathcal{L}_i(M)$).

Definition 13. Mode declaration M**.** *A mode declaration has either the form modeh(n,atom) or modeb(n,atom) where n, the recall, is either an integer, $n > 1$, or '*' and atom is a ground atom. Terms in the atom are either normal or place-marker. A normal term is either a constant or a function symbol followed by a bracketed tuple of terms. A place-marker is either +type, -type or #type, where type is a constant. If m is a mode declaration then $a(m)$ denotes the atom of m with place-markers replaced by distinct variables. The sign of m is positive if m is a modeh and negative if m is a modeb.*

Definition 14. Definite mode language $\mathcal{L}(M)$**.** *Let C be a definite clause with a defined total ordering over the literals and M be a set of mode declarations. $C = h \leftarrow b_1, .., b_n$ is in the definite mode language $\mathcal{L}(M)$ if and only if 1) h is the atom of a modeh declaration in M with every place-marker +type and -type replaced by variables and every place-marker #type replaced by a ground term and 2) every atom b_i in the body of C is the atom of a modeb declaration in M with every place-marker +type and -type replaced by variables and every place-marker #type replaced by a ground term and 3) every variable of +type in any atom b_i is either of +type in h or of -type in some atom b_j, $1 \leq j < i$.*

Definition 15. Depth of variables. *Let C be a definite clause and v be a variable in C. Depth of v is defined as follows:*

$$d(v) = \begin{cases} 0 & \text{if } v \text{ is in the head of } C \\ (max_{u \in U_v} d(u)) + 1 & \text{otherwise} \end{cases}$$

where U_v are the variables in atoms in the body of C containing v.

Definition 16. Depth-bounded mode language $\mathcal{L}_i(M)$**.** *Let C be a definite clause with a defined total ordering over the literals and M be a set of mode declarations. C is in $\mathcal{L}_i(M)$ if and only if C is in $\mathcal{L}(M)$ and all variables in C have depth at most i according to Definition 15.*