

TopLog: ILP using a logic program declarative bias

Stephen H. Muggleton, José C. A. Santos and Alireza Tamaddoni-Nezhad

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

Abstract. This paper introduces a new Inductive Logic Programming (ILP) framework called Top Directed Hypothesis Derivation (TDHD). In this framework each hypothesised clause must be derivable from a given logic program called top theory (\top). Top theory can be viewed as a logic program declarative bias which defines the hypothesis space. This replaces the metalogical mode statements which are used in many ILP systems and has all the expressive power of a logic program, and can be efficiently reasoned with using standard logic programming techniques. In this paper firstly we present a theoretical framework for TDHD and show that a standard SLD derivation can be used to efficiently derive hypotheses from \top . Secondly, we present a prototype implementation of TDHD within a new ILP system called TopLog. Thirdly, we show that the accuracy and efficiency of TopLog, on several benchmark datasets, is comparable with the accuracy and efficiency of a state of the art ILP system like Aleph.

Many Inductive Logic Programming (ILP) papers deal with various aspects of search efficiency. Recent techniques explored include methods for learning declarative bias [2], the use of special purpose data structures [3] and the use of stochastic search [15, 17, 18].

In this paper we introduce a new approach to providing declarative bias called Top-Directed Hypothesis Derivation (TDHD). The approach extends the use of the \perp clause in Mode-Directed Inverse Entailment (MDIE) [14]. In Inverse Entailment \perp is constructed for a single, arbitrarily chosen training example. Refinement graph search is then constrained by the requirement that all hypothesised clauses considered must subsume \perp . In TDHD we further restrict the search associated with each training example by requiring that each hypothesised clause must also be entailed by a given logic program, \top . \top can be viewed as a form of first-order declarative bias which defines the hypothesis space, since each hypothesised clause must be derivable from \top .

The use of the \top theory in TopLog is in some ways comparable to grammar-based declarative biases [4, 10]. In a context-free grammar it is usual to differentiate between *terminal* and *non-terminal* symbols, where *terminal* symbols are those which can appear in a sentence generated by the grammar. Likewise in \top it is useful to distinguish between *terminal* and *non-terminal* predicates. *Terminal* predicates represent those that can appear in hypotheses derived from \top .

Non-terminal predicates are simply used for control purposes within \top . However, compared with a grammar-based declarative bias, \top has all the expressive power of a logic program, and can be efficiently reasoned with using standard logic programming techniques.

The SPECTRE system [1] employs an approach related to the use of \top . SPECTRE also relies on an overly general logic program as a starting point. However, unlike the TopLog system described in this paper, SPECTRE proceeds by successively unfolding clauses in the initial theory. The explicit distinction between \top and first-order background knowledge allows TopLog to avoid some of the problems early versions of SPECTRE encountered in learning recursive programs. TDHD is also related to Explanation-Based Generalisation (EBG) [11]. EBG starts with an initial general theory, which is used to explain individual examples. EBG distinguishes between *operational* and *non-operational* predicates in a similar fashion to the terminal and non-terminal distinction made in TDHD. As with TDHD, the proofs of individual examples in EBG are generalised to provide clausal explanations by dropping all non-operational predicates. However, like SPECTRE, EBG does not make the key MDHD distinction between the \top theory and background knowledge. Thus in EBG the initial starting theory is interpreted as domain knowledge, which means that the explanations can be assumed correct by derivation. For this reason EBG and the associated EBL [9] were viewed as a form of deductive learning, aimed at speeding up program performance. By contrast, the clauses generated by TDHD represent inductive hypotheses.

In this paper we show how hypotheses can be efficiently derived using Prolog's inbuilt theorem-proving techniques by re-ordering and pruning the individual refutations of examples. This allows a form of test-incorporation to be used in the derivation of hypotheses, avoiding the less efficient generate-and-test approach employed in many ILP systems. The TopLog system, described in this paper, uses this technique to generate all hypotheses in a single pass through the training data. The example coverage information in these hypotheses is then used by TopLog in a second phase to support the choice of a set of hypothesised clauses. Maximisation of information compression is used as the criterion to judge sets of clauses in this second phase. One of the key efficiency strengths of TopLog is that there is no requirement for theorem proving in this second phase.

This paper is arranged as follows. Section 1 provides a theoretical framework for the work. This starts with an introduction to the notation used for SLD derivations and refutations in Section 1.1. We then review Mode-Directed Inverse Entailment in Section 1.2 before defining TDHD in Section 1.3. We prove a theorem concerning the MDHD technique of re-ordering and pruning refutations to obtain derivations of hypotheses. A worked example is provided. The re-ordering theorem is then used as the basis for the Hypothesis Generation system in the TopLog described in Section 2. Experiments comparing the speed and accuracy of TopLog and Aleph are given in Section ???. In Section 4 we conclude and discuss further work.

1 Theoretical framework

In this section we survey MDIE and then introduce the TDHD framework. We start by introducing the notation used in this section.

1.1 Mathematical preliminaries

In this paper we will use standard notation from Inductive Logic Programming [16] to describe general clauses, Horn clauses, definite clauses, entailment, resolution and subsumption. Special notation will be used for SLD derivations and refutations as follows.

Definition 1. SLD derivation. Let C_1, C_2, \dots, C_n be definite clauses and G_0 be a Horn clause. An SLD derivation, denoted by $R = \langle G_0, C_1, \dots, C_n \rangle$, is a sequence of Horn clauses G_0, G_1, \dots, G_n such that for each $1 \leq i \leq n$, G_i is a binary resolvent of G_{i-1} and C_i , using the head of C_i and the leftmost atom in the body of G_{i-1} as the literals resolved upon. We say that R derives the Horn clause G in the case that G is the final resolvent of R .

SLD derivation is exemplified below.

Example 1. Example of SLD derivation. Figure 1 shows the SLD derivation $R = \langle G_0, C_1, C_2 \rangle$. Clauses G_0 and C_1 are first resolved on their leftmost respective literals to give the resolvent G_1 . Similarly in the second resolution C_2 is resolved with G_1 to give the final resolvent G_2 . Thus R derives G_2 .

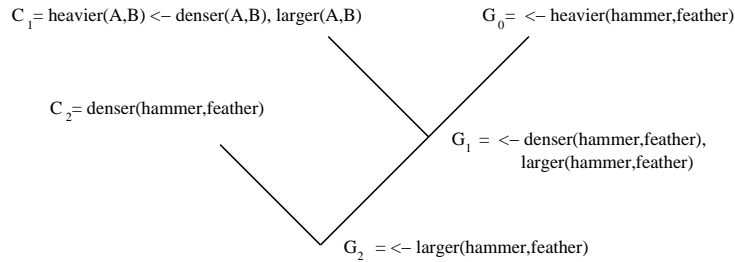


Fig. 1. Example of SLD derivation

We now define SLD refutation as a special case of SLD derivation.

Definition 2. SLD refutation. An SLD refutation is an SLD derivation which derives the empty clause, \square .

1.2 Mode-Directed Inverse Entailment

Mode-Direct Inverse Entailment (MDIE) was introduced in [14] as the basis for the ILP system Progol. The input to an MDIE system is the vector $S_{MDIE} = \langle M, B, E \rangle$ where M is a set of mode statements, B is a logic program representing the background knowledge and E is set of examples. M can be viewed as a set of metalogical statements used to define the hypothesis language \mathcal{L}_M . The aim of the system is to find a set of consistent hypothesised clauses H such that for each clause $h \in H$ there is at least one positive example $e \in E$ such that the following holds.

$$B, h \models e$$

For any B, h, e this is equivalent to the following.

$$B, \neg e \models \neg h$$

This form allows hypotheses to be derived from B and e using standard Prolog theorem proving techniques. Since $\neg h$ takes the form of a ground conjunction of literals, for any finitely bound hypothesis language \mathcal{L}_M there is a maximal ground conjunction \perp_e for which the following holds.

$$B, \neg e \models \neg \perp_e \models \neg h$$

Having selected an example e and constructed \perp_e Progol conducts a refinement graph search which considers hypotheses h in the interval

$$\square \succ h \succeq \perp_e$$

where “ \succeq ” denotes θ -subsumption.

1.3 Top-Directed Hypothesis Derivation

The input to an TDHD system is the vector $S_{TDHD} = \langle NT, \top, B, E \rangle$ where NT is a set of “non-terminal” predicate symbols, \top is a logic program representing the declarative bias over the hypothesis space, B is a logic program representing the background knowledge and E is a set of examples. The following two conditions hold for clauses in \top : (a) each clause in \top must contain at least one occurrence of an element of NT while clauses in B and E must not contain any occurrences of elements of NT and (b) clauses in B cannot call clauses in \top , i.e. any predicate appearing in the head of some clause in \top must not occur in the body of any clause in B . The aim of a TDHD system is to find a set of consistent hypothesised clauses H , containing no occurrence of NT , such that for each clause $h \in H$ there is at least one positive example $e \in E$ such that the following hold.

$$\top \models h \tag{1}$$

$$B, h \models e \tag{2}$$

Given the assumptions above we can now show the following lemma.

Lemma 1. Example derivability. *Given $S_{TDHD} = \langle NT, \top, B, E \rangle$ assumptions (1) and (2) hold only if for each positive example $e \in E$ there exists an SLD refutation R of $\neg e$ from \top, B .*

Proof. *Assuming false there must exist a positive example e for which there does not exist an SLD refutation of $\neg e$ from \top, B . However, given S_{TDHD} we can show from (1) that for each h*

$$\top, B \models B, h. \quad (3)$$

From (3) and (2) it follows that for each positive example $e \in E$

$$\top, B \models e. \quad (4)$$

From (4) and the completeness of SLD derivation [12] it follows that there exists an SLD refutation R of $\neg e$ from \top, B . This contradicts the assumption and completes the proof.

According to the following theorem, the results of the preceding lemma can be used to extract implicit hypotheses from the refutations of a positive example $e \in E$.

Theorem 1. Re-ordering theorem. *Given $S_{TDHD} = \langle NT, \top, B, E \rangle$ and a positive example $e \in E$, each SLD refutation R of $\neg e$ from \top, B can be re-ordered to give $R' = D_h R_e$ where D_h is an SLD derivation of a hypothesis h for which (1) and (2) hold.*

Sketch proof. *Assume the theorem is false. Therefore R cannot be re-ordered to give $R' = D_h R_e$ where D_h is an SLD derivation of a hypothesis h for which (1) and (2) hold. It should be noted that each clause in R can be identified as originating from either \top or B depending on whether or not it contains an occurrence of a predicate symbol from NT respectively. Now, while preserving the order of clauses in R let D_h and R_e be the sequence of clauses from \top and B respectively. According to the assumption, D_h is not an SLD derivation of a hypothesis h for which (1) and (2) hold. However, by construction D_h contains all and only \top clauses from R . By definition each \top clause contains at least one literal containing an occurrence of a predicate symbol from NT . Since R is a refutation, each literal in each clause must be resolved away in order to derive \square . Since clauses in B cannot call clauses in \top and by construction all NT literals in R can be found in D_h , these literals must form a contiguous sequence of complementary pairs within R . Applying resolution to each of these complementary pairs in D_h , in the same order as in R will lead to the SLD derivation of a clause h . Therefore, according to the assumption either (1) or (2) does not hold for h . However, (1) does hold for h since by construction D_h is an SLD derivation of h from \top . Therefore it must be that (2) does not hold. However, since all resolution steps involving NT complementary pairs have been applied in D_h , it follows that h must consist of literals not containing predicate symbols from NT . Also since R is a refutation, all these literals must have complementary pairs found in clauses*

in R_e . Thus $\langle h \rangle R_e$ must be an SLD refutation of e from B , which implies that (2) holds. This contradicts the assumption and completes the proof.

Let us now consider a simple example of this theorem.

Example 2. Simple example. Let $S_{TDHD} = \langle NT, \top, B, E \rangle$ where NT , B , e and \top are defined as follows:

$$\begin{aligned} NT &= \{\$body\} \\ B &= b_1 = \text{pet}(\text{lassy}) \leftarrow \\ e &= \text{nice}(\text{lassy}) \leftarrow \end{aligned} \quad \top = \begin{cases} \top_1 : \text{nice}(X) \leftarrow \$body(X) \\ \top_2 : \$body(X) \leftarrow \text{pet}(X) \\ \top_3 : \$body(X) \leftarrow \text{friend}(X) \end{cases}$$

Figure 2 shows the linear refutation $R = \langle \neg e, \top_1, \top_2, b_1 \rangle$. We now construct

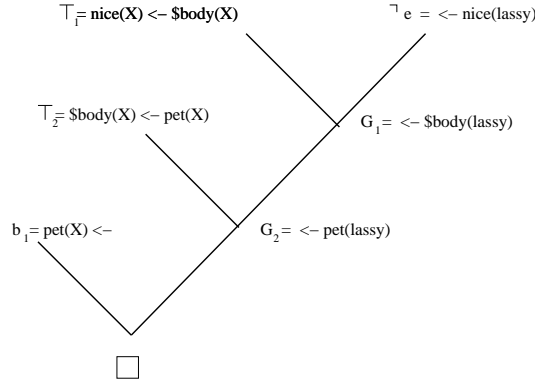


Fig. 2. SLD refutation of $\neg e$

the re-ordered refutation $R' = D_h R_e$ where $D_h = \langle \top_1, \top_2 \rangle$ which derives the clause $h = \text{nice}(X) \leftarrow \text{pet}(X)$ for which (1) and (2) hold.

2 System Description

TopLog is a prototype ILP system developed to implement the TDHD scheme described in the previous sections. It is totally implemented in Prolog and is ensured to run at least in YAP, SWI and Sicstus Prolog. It is publicly available at [22] and may be freely used for academic purposes.

It is about 3000 lines of Prolog code and is composed of ten modules with only one, TopLog, required to be imported by the user. This module exports a few predicates and has all the user needs to interact with the system. Some effort has been taken to ensure TopLog remains input compatible with existing ILP systems, namely Aleph and Progol.

2.1 From mode declarations to \top theory

As the user of TopLog may not be familiar with specifying a search bias in the form of a logic program, TopLog has a module to build a general \top theory automatically from user specified mode declarations. In this way input compatibility is ensured with existing ILP systems. Below is a simplified example of user specified mode declarations and the automatically constructed \top theory.

$$\begin{array}{l} \text{modeh(mammal(+animal)).} \\ \text{modeb(has_milk(+animal)).} \\ \text{modeb(has_eggs(+animal)).} \end{array} \quad \top = \begin{cases} \top_1 : \text{mammal}(X) \leftarrow \text{\$body}(X) \\ \top_2 : \text{\$body}(X) \leftarrow \\ \top_3 : \text{\$body}(X) \leftarrow \text{has_milk}(X), \text{\$body}(X) \\ \top_4 : \text{\$body}(X) \leftarrow \text{has_eggs}(X), \text{\$body}(X) \end{cases}$$

Fig. 3. Mode declarations and a \top theory automatically constructed from it

The above illustrated \top theory is extremely simplified. The actual implementation has stricter control rules like: variables may only bind with others of the same type, a newly added literal must have its input variables already bound.

It is worth pointing out that the user could directly write a \top theory specific for the problem, potentially restricting the search better than the generic \top theory built automatically from the mode declarations.

2.2 Hypothesis derivation

In TopLog hypotheses are generated in a novel way. There is no construction of the bottom clause but rather an example guided generalization, deriving all hypotheses that entail the example with respect to the background knowledge.

The hypothesis derivation procedure is composed of two distinct steps. In the first step an example is proved from the background knowledge and the \top theory. That is, the \top theory is executed having the example as its start clause. This execution yields a proof consisting of a sequence of clauses from the \top theory and background knowledge.

For instance, using the \top theory from figure 3 and $B = b_1 = \text{has_milk}(\text{dog})$ to derive refutations for example $e = \text{mammal}(\text{dog})$, the following two refutations would be yielded: $r_1 = \langle \neg e, \top_1, \top_2 \rangle$ and $r_2 = \langle \neg e, \top_1, \top_3, b_1, \top_2 \rangle$.

In the second step, Theorem 1 is applied to r_1 and r_2 deriving, respectively, the clauses $h_1 = \text{mammal}(X)$ from $\langle \top_1, \top_2 \rangle$ and $h_2 = \text{mammal}(X) \leftarrow \text{has_milk}(X)$ from $\langle \top_1, \top_3, \top_2 \rangle$.

2.3 Basic TopLog Learning Algorithm

The table below shows the pseudo code for the basic TopLog Learning algorithm. Notice that the hypothesis derivation procedure described above corresponds to step 2.1 below.

-
1. Let H be an empty set of pairs (hypothesis, examples that generated it)
 2. For each positive example e in E do
 - 2.1. Generate all hypotheses, H_e that are generalizations of e
 - 2.2 Merge H_e into H (annotating which examples generated a given hypothesis)
 3. Compute the coverage of each hypothesis in H
 4. Build final theory, T , by choosing a subset of hypothesis in H
-

Table 1. Main TopLog algorithm

In step 1, H is a set of pairs: $\langle h, Eg_h \rangle$, where Eg_h is the set of (positive) examples that generated h . Step 2.2, adds more pairs to the set in case the hypothesis was not generated by any example before or updates existing pairs adding the current example as another generator of a previously found hypothesis. For efficiency we assume two hypotheses are equivalent if their derivations are identical.

The third step of the algorithm, computing the coverage of each hypothesis, would not be needed if the user program is a pure logic program (i.e. no cut symbol, nor usage of Prolog built in operators) and no negative examples exist. This is because, by construction, the TDHD algorithm generates all hypotheses that entail a given example with respect to the user supplied mode declarations. This implies that the coverage of an hypothesis is exactly the set of examples that have it as their generalization.

However, this coverage computation step is needed for the negative examples, as they were not used to build the hypothesis set, and we also apply it to the positive examples because of two practical reasons. First, it is often the case that the user program is not a pure logic program (e.g. uses the cut symbol or built in comparison operators like $<$ and $>$). Second, it is not guaranteed that all hypotheses generalizing an example are generated from it in the case not enough resources (e.g. maximum proof depth, maximum hypotheses per example) are given to the program.

Notice that, contrary to the default setting in Aleph and Progol, no examples are retracted during the hypothesis construction stage (possibly changing the hypotheses that may be generated later) and thus the final theory is not influenced by the relative example order in the user program.

2.4 Building the final theory

The data available for building the final theory is a set of hypotheses, H , where each hypothesis has associated the set of examples from which it was derived and the set of examples which it entails. As we have seen above, for pure logic programs and when the given search resources are enough, these two sets have the same elements.

The final theory, T , is a subset H' of H that maximizes a score function. Two important metrics are defined for T : 1) set of examples covered, E_{c_T} , and 2) number of literals. The set of examples a theory covers is the union of the examples covered by its individual hypotheses and the total number of literals in a theory is the sum of all literals present in all its hypotheses. The ultimate goal of

a theory is to, while keeping good comprehensibility, achieve the highest accuracy on unseen data. In order to fulfill that goal, TopLog’s default compression-based evaluation function for theories is:

$$\sum_{e \in E_{c_T}} \text{weight}(e) - \sum_{h \in T} |h| \quad (5)$$

The weight associated to an example, $\text{weight}(e)$, is user defined with positive examples having weight 1 and negative examples weight -1 by default. Weights greater than zero mean the example is positive, smaller than zero means it is negative.

The rationale behind this score function is to, assuming each example is worth one literal, measure by how many literals a theory compresses a dataset. Notice that this score function is similar to Progol’s and Aleph’s compression measure.

The problem of building a final theory, in our setting, is similar to the set covering problem, which is known to be NP-Complete and has plenty of literature on other related theoretical results [23].

As there is no known efficient algorithm to generate the optimal T , an approximation will have to suffice. Natural choices for generating T are optimization approaches like simulated annealing or genetic algorithms, however we opted for a much simpler greedy algorithm which has several advantages. It is deterministic, has no parameters to tune, is efficient, simple to implement in Prolog and, for the similar case of the set covering problem, there are theoretical results proving that a greedy algorithm is the best-possible polynomial time approximation [13].

-
1. Let H be the unique set of hypotheses generated from the examples
 2. Let T , the final theory, be an empty set of hypotheses
 3. While there exists an H_i in H that may increase T ’s score
 - 3.1. Pick the H_i that, when added to T , most increases T ’s score
 - 3.2. Remove H_i from H
 4. Output all the hypotheses on T as the final theory
-

Table 2. Greedy final theory generation algorithm

This greedy approach is computationally efficient and returns reasonably good solutions.

2.5 Efficient cross-validation

When the user provides examples they may also indicate which fold each example belongs to or let TopLog randomly and uniformly assign examples to folds. TopLog builds all hypotheses that are generated from each example in a first stage. In a second stage the final theory is chosen, allowing a efficient cross-validation implementation.

In the first stage all hypotheses are generated for each example regardless of the fold it belongs to, thus allowing TopLog to have associated with each hypothesis two sets of examples: 1) the examples that generated it, 2) the examples it covers (a superset of the first). Only the theory generation algorithm, detailed in table 2, needs to take folds into account.

In step 1 the hypotheses considered are only the ones generated by at least one example from a training fold (i.e. hypotheses generated exclusively from examples from the current test fold are removed). In step 3, the computation of T 's score by adding a new hypothesis to it now also has to take folds into account. For the purpose of scoring the merit of adding a H_i to T , any positive or negative examples H_i covers are ignored, thus we only consider H_i 's coverage on the training folds.

The generation and coverage computation dominates execution times. This is done only once, which minimizes the cost of the greedy algorithm and its application in cross-validation. Notice that this efficient cross-validation, which is internal to TopLog is in sharp contrast with the one performed in most other ILP systems (e.g. Aleph, Progol) where the time consuming hypothesis generation step needs to take place for each fold of the dataset. These ILP systems typically do not support directly cross-validation but rather rely on external scripts to separate the data into folds and run the system fold times in each of them.

2.6 Comparison with other ILP Systems

There are many existing ILP systems. Two of the widely used systems are Progol [14] and Aleph [21]. Aleph tries to integrate ideas from many systems and, with the proper settings, can emulate Progol. Aleph's main algorithm is: 1) Select an example to be generalized, 2) Build most-specific-clause, \perp , for the example, 3) Heuristic search to find a clause more general than \perp but bounded by it, 4) Remove examples covered by the clause found in step 3, 5). If more positive examples exist go to step 1 otherwise stop.

Some of TopLog's advantages are: irrelevance of example order, global optimization of the theory after all hypothesis are available (possible leading to higher accuracy), efficient cross-validation. The main disadvantage is that the hypothesis derivation procedure is not heuristically guided which may lead to many non interesting hypothesis being generated leading to decreased efficiency.

3 Experimental Evaluation

In this section we empirically evaluate TopLog and Aleph using four well known datasets and compare their results.

3.1 Materials

The datasets we chose are: mutagenesis [20], carcinogenesis [19], alzheimers-amine [24] and DSSTox [25] mainly because they are well known to the ILP

community and are good examples of practical problems where relational knowledge is important.

In these datasets the purpose is to characterize an active molecule (for the problem at hand). It is given examples of molecules that exhibit the property (i.e. positives) and examples of molecules that do not exhibit the property (i.e. negatives). The task of the ILP system is to find a theory that entails as most of the positive examples while entailing as few of the negative examples as possible.

Notice that to ensure the problems are interesting from an ILP perspective we have only used structural features (e.g. atoms, bonds, and structural motifs formed of atoms and bonds). When we use quantitative features (e.g. logp and lumo) the accuracies are higher but the hypotheses found are poorer in the sense that they offer less clues on how to build such molecules. The table below summarizes the dataset information.

Dataset	#E ⁺	#E ⁻	# E	Def. Acc.	#Modeb	#Background
Mutagenesis	125	63	188	66.5%	4	14,379
Carcinogenesis	162	136	298	54.4%	40	24,672
Alzheimers	343	343	686	50.0%	32	628
DSSTox	220	356	576	61.8%	2	27,793

Table 3. Dataset statistics

The meaning of the columns is: number of positive examples, number of negative examples, total number of examples, default accuracy, number of body mode declarations and size of the background knowledge measured in number of clauses. The default accuracy is the accuracy yielded by the simple model that classifies an example as belonging to the most common class, its value is thus $\max(\#E^+, \#E^-)/\#E$ and should be considered the baseline that any non-trivial classifier should beat.

3.2 Methods

We chose to compare TopLog against Aleph because Aleph is a Mode Directed Inverse Entailment ILP system¹ and is also implemented in Prolog.

The Prolog interpreter underlying an ILP system plays an important role in its overall performance as the majority of the CPU time is spent building dynamic clauses (i.e. inducing hypotheses) and computing their coverage (i.e. scoring them). Most Prolog interpreters are not optimized for these tasks which can degrade tremendously the overall system performance.

YAP distinguishes itself from other Prolog interpreters in this respect. It implements demand driven indexing [5], even for dynamic predicates [7], which is crucial for good performance in ILP engines. When running on Sicstus, TopLog is about 10 times slower and on SWI about 100 times slower!

¹ Another well known ILP system using MDIE is Progol. In spite of being implemented in C, Progol is considerably slower than Aleph mainly due to the merit of Aleph's underlying Prolog interpreter (YAP).

The experiments were performed on a Intel Core 2 Duo @ 2.13 GHz with 2Gb of RAM using Ubuntu Linux with kernel version 2.6.0.3. Aleph current version (5.0) is publicly available at [21] and TopLog current version (0.2) is publicly available at [22]. Both TopLog and Aleph were executed on the latest YAP (a CVS pre-release of version 5.1.3).

Aleph and TopLog were executed with as close as possible settings to ensure a fair test. The clause length (i.e. maximum number of literals in the body of an hypothesis) was set to 4 (except in DSSTox where it was set to 10), noise (i.e. maximum percentage of negative examples a clause may cover 100%), evaluation function set to compression, and number of search nodes (i.e. hypotheses) per example set to 1000.

Aleph was called both with *induce* and *induce_max* settings. The difference between the two is that *induce* (the default), after finding a compressive clause for an example, retracts all the positive examples covered by that clause while *induce_max* does not. TopLog also does not retract any example during the search and thus we make a fair test if we compare the *induce_max* times rather than the *induce* times.

3.3 Results and Discussion

In the table below the time column has the running time, in CPU seconds, the ILP system took to build the model in the training data (Train column) and the total time it took to build the models for the ten folds (CV column). We distinguish between the two times to highlight the benefits of the efficient cross validation in TopLog².

The accuracy column has the average (over the ten folds) percentage of correct predictions made by model with the respective standard deviation.

Dataset	Aleph with induce			Aleph with induce_max			TopLog		
	CV Accuracy	Times		CV Accuracy	Times		CV Accuracy	Times	
Mutagenesis	77.2%±9.2%	0.4s	4s	68.6%±11.4%	2s	17s	70.2%±11.9%	0.5s	0.7s
Carcinogenesis	60.9%±8.2%	6s	54s	65.1%±8.6%	29s	245s	64.8%±6.9%	22s	23s
Alzheimers	67.2%±5.0%	5s	40s	72.6%±6.2%	18s	156s	70.4%±5.6%	50s	50s
DSSTox	70.5%±6.5%	30s	253s	71.3%±3.4%	82s	684s	75.5%±5.3%	2s	3s

Table 4. Accuracy and time comparison between Aleph and TopLog

If we only consider the training time, TopLog, except for the Alzheimers dataset, is always faster than Aleph with the *induce_max* setting. Comparing with the *induce* setting the advantage is not clear (e.g. in Alzheimers Aleph is much faster than TopLog but in DSSTox the reverse occurs).

² In TopLog building the cross validated model takes nearly the same time as building the training model whereas in an MDIE system it takes approximately $N - 1/N$ times the time per fold, with N being the total number of folds.

Considering cross validation then TopLog is often clearly faster. Although this may seem a side point, built-in efficient cross validation is important in practical applications in order to assess properly the model accuracy. Furthermore, this is not possible to do efficiently (i.e. without having to recompute all the hypotheses) in a MDIE system like Aleph because there is no association between hypotheses and the examples from which they were derived.

4 Conclusions and Future work

The key innovation of the TDHD framework is the introduction of a first order \top theory which constrains the search space. We prove that SLD derivation can be used to efficiently derive hypotheses from \top . A multi-pass algorithm is described, together with its implementation, in a new general ILP system TopLog.

Unlike other forms of declarative bias, in TDHD the \top theory is a logic program, allowing it first class status for logic program-based reasoning mechanisms. Thus, for instance, one could apply efficiency improving program transformations to \top such as fold/unfold operations, and other forms of partial evaluation. In line with recent interest in learning declarative bias [2], in the TDHD setting \top could potentially be learned using ILP techniques as a logic program.

The empirical comparison demonstrates that this new approach is competitive, both in predictive accuracy and speed, with a state of the art system like Aleph. Below we discuss future work and some limitations of TopLog.

4.1 Natural opportunities for parallelization

Due to the way hypothesis are built in TopLog, where the construction of the set of hypothesis that covers one example is independent of the construction of the hypothesis set for the other examples, it is straightforward to parallelize the algorithm presented in Table 1. The idea is simply to divide the number of examples by the number of processors available. Each child process (running on its own processor) would collect the unique set of hypothesis for the subset of examples that was given to it, still keeping a list of unique hypothesis generated. At the end the parent process would merge this lists of hypothesis into a unique list. It should be noted that the same technique can be used for parallelizing the hypothesis coverage computation and the cross fold validation step.

4.2 Learning recursive theories

TopLog can learn the recursive definition for simple recursive programs like member/2, append/3, and even more complex ones like qsort/2, nchoosem/3 and fibonacci/2. Notice that fibonacci/2 and nchoosem/3 are particularly problematic to be learned in Aleph or Prolog due to requiring the construction of a very large \perp clause.

However, TopLog currently requires the theory base cases to be present in the background knowledge. This is because hypotheses are generated individually

and, when computing their coverage, are assumed to be independent of each other which does not hold for recursive theories.

4.3 Sampling hypothesis space

Currently TopLog searches the hypothesis space according to the \top theory automatically constructed from the user mode declarations. Although this approach seems to work well in practice, for large hypotheses spaces with a low recall only the surface of the space is searched. Possible clusters of interesting hypotheses will go unnoticed if they are further than the defined recall.

If the hypothesis space is sampled according to a user defined bias it is possible to cover more areas of the hypothesis space and reduce the overlap of identical hypothesis being generated by distinct examples. These two factors allow for an increase in the hypothesis space searched while using the same memory and CPU time resources.

Acknowledgments

We would like to thank James Cussens for illuminating discussions on the TDHD framework. We would also like to acknowledge Vítor Santos Costa for a great Prolog system and his prompt help in fixing some of YAP's problems allowing us to stretch it to the limits. The first author would like to thank the Royal Academy of Engineering and Microsoft for funding his present 5 year Research Chair. The second author would like to thank the financial support by his Wellcome Trust Ph.D. scholarship. The third author was supported by the BBSRC CISBIC grant.

References

1. H. Boström and P. Idestam-Almquist. Specialisation of logic programs by pruning SLD-trees. In S. Wrobel, editor, *Proceedings of the Fourth Inductive Logic Programming Workshop (ILP94)*, pages 31–48, Bonn, 1994. GDM-studien Nr. 237.
2. W. Bridewell and L. Todorovski. Learning declarative bias. In *Proceedings of the 17th International Conference on Inductive Logic Programming*, pages 63–77, Berlin, 2007. Springer-Verlag. LNAI 4894.
3. R. Camacho, N. Fonseca, R. Rocha, and V. Santos Costa. Ilp :- just trie it. In *Proceedings of the 17th International Conference on Inductive Logic Programming*, pages 78–87, Berlin, 2007. Springer-Verlag. LNAI 4894.
4. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.
5. Vítor Santos Costa, Konstantinos F. Sagonas, and Ricardo Lopes. Demand-driven indexing of prolog clauses. In Dahl and Niemelä [8], pages 395–409.
6. Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart De-moen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 4:465–491, 2003.

7. Anderson Faustino da Silva and Vítor Santos Costa. Design, implementation, and evaluation of a dynamic compilation framework for the yap system. In Dahl and Niemelä [8], pages 410–424.
8. Verónica Dahl and Ilkka Niemelä, editors. *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.
9. G. DeJong and R. Mooney. Explanation-based learning: an alternative view. *Machine Learning*, 1(2):145–176, 1986.
10. S. Džeroski and L. Todorovski. Discovering dynamics: From inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4(1):89–108, 1995.
11. S.T. Kedar-Cabelli and L.T. McCarty. Explanation-based generalization as resolution theorem proving. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, Los Altos, 1987. Morgan Kaufmann.
12. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
13. Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.
14. S.H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
15. S.H. Muggleton and A. Tamaddoni-Nezhad. QG/GA: A stochastic search for Progol. *Machine Learning*, 70(2–3):123–133, 2007. DOI: 10.1007/s10994-007-5029-3.
16. S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.
17. L. Oliphant and J. Shavlik. Using bayesian networks to direct search in inductive logic programming. In *Proceedings of the 17th International Conference on Inductive Logic Programming*, Berlin, 2007. Springer-Verlag. LNAI 4894.
18. A. Paes, G. Zaverucha, and V. Santos Costa. Revising first-order logic theories from examples through stochastic local search. In *Proceedings of the 17th International Conference on Inductive Logic Programming*, pages 201–210, Berlin, 2007. Springer-Verlag. LNAI 4894.
19. A. Srinivasan, R.D. King S.H. Muggleton, and M. Sternberg. Carcinogenesis predictions using ILP. In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 273–287. Springer-Verlag, Berlin, 1997. LNAI 1297.
20. A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the Fourth International Inductive Logic Programming Workshop*. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994. GMD-Studien Nr 237.
21. Ashwin Srinivasan. *The Aleph Manual*. University of Oxford, 2007. <http://web2.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html>
22. José Santos. *TopLog site*. Imperial College, 2008. <http://www.doc.ic.ac.uk/~jcs06/TopLog>
23. Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.
24. R.D. King, A. Srinivasan, and M.J.E. Sternberg. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing*, 13:411–433, 1995.
25. A.M. Richard and C.R. Williams. Distributed structure-searchable toxicity (DSSTox) public database network: A proposal. *Mutation Research*, 499:27–52, 2000.