Bottom-Up ILP using Large Refinement Steps*

Marta Arias and Roni Khardon

Department of Computer Science, Tufts University
Medford, MA 02155, USA
{marias,roni}@cs.tufts.edu

Abstract. The LogAn-H system is a bottom up ILP system for learning multi-clause and multi-predicate function free Horn expressions in the framework of learning from interpretations. The paper introduces a new implementation of the same base algorithm which gives several orders of magnitude speedup as well as extending the capabilities of the system. New tools include several fast engines for subsumption tests, handling real valued features, and pruning. We also discuss using data from the standard ILP setting in our framework, which in some cases allows for further speedup. The efficacy of the system is demonstrated on several ILP datasets.

1 Introduction

Inductive Logic Programming (ILP) has established a core set of methods and systems that proved useful in a variety of applications [20, 4]. Early work in the Golem system [21] (see also [19]) used Plotkin's [24] least general generalization (LGG) within a bottom up search to find a hypothesis consistent with the data. On the other hand, much of the research following this (e.g. [25, 18, 7, 3]) has used top down search methods to find useful hypotheses. However, several exceptions exist. STILL [26] uses a disjunctive version space approach which means that it has clauses based on examples but it does not generalize them explicitly. The system of [1] uses bottom up search with some ad hoc heuristics to solve the challenge problems of [11]. The LogAn-H system [13] is based on an algorithm developed in the setting of learning with queries [12] but uses heuristics to avoid asking queries and instead uses a dataset as input. This system uses a bottom up search, based on inner products of examples which are closely related to LGG. Another important feature of LogAn-H is that it does a refinement search but, unlike other approaches, it takes large refinement steps instead of minimal ones.

In previous work [13] LogAn-H was shown to be useful in a few small domains. However, it was hard to use the system in larger applications mainly due to high run time. One of the major factors in this is the cost of subsumption. Like other bottom up approaches, LogAn-H may use very long clauses early on in the search and the cost of subsumption tests for these is high. This is in contrast to top down approaches that start with short clauses for which subsumption is

* This work has been partly supported by NSF Grant IIS-0099446

easy. A related difficulty observed in Golem [21] is that LGG can lead to very large hypotheses. In LogAn-H this is avoided by using 1-1 object mappings. This helps reduce the size of the hypothesis but gives an increase in complexity in terms of the size of the search.

The current paper explores a few new heuristics and extensions to LogAn-H that make it more widely applicable both in terms of speed and range of applications. The paper describes a new implementation that includes several improved subsumption tests. In particular, for LogAn-H we need a subsumption procedure that finds all substitutions between a given clause and an example. This suggests a memory based approach that collects all substitutions simultaneously instead of using backtracking search. Our system includes such a procedure which is based on viewing partial substitutions as tables and performing "joins" of such tables to grow complete substitutions. A similar table-based method was developed in [8]. This approach can be slow or even run out of memory if there are too many partial substitutions in any intermediate step. Our system implements heuristics to tackle this, including lookahead search and randomized heuristics. The latter uses informed sampling from partial substitution tables if memory requirements are too large. In addition, for some applications it is sufficient to test for existence of substitutions between a given clause and an example (i.e. we do not need all substitutions). In these applications we are able to use the fast subsumption test engine Django [16] in our system. The paper shows that different engines can give better performance in different applications, and gives some paradigmatic cases where such differences occur.

In addition the system includes new heuristics and facilities including discretization of real valued arguments and pruning of rules. Both introduce interesting issues for bottom up learning which do not exist for top down systems. These are explored experimentally and discussed in the body of the paper.

The performance of the system is demonstrated in three domains: the Bongard domain [7, 13], the KRK-illegal domain [25], and the Mutagenesis domain [29]. All these have been used before with other ILP systems. The results show that our system is competitive with previous approaches while applying a completely different algorithmic approach. This suggests that bottom up approaches can indeed be used in large applications. The results and directions for future work are further discussed in the concluding section of the paper.

2 Learning from Interpretations

We briefly recall the setup in Learning from interpretations [6] and introduce a running example which will help explain the algorithm. The task is to learn a universally quantified function-free Horn expression, that is, a conjunction of Horn clauses. The learning problem involves a finite set of predicates whose signature, i.e. names and arities, are fixed in advance. In the examples that follow we assume two predicates $p()$ and $q()$ both of arity 2. For example, $c_1 = \forall x_1, \forall x_2, \forall x_3, [p(x_1, x_2) \wedge p(x_2, x_3) \rightarrow p(x_1, x_3)]$ is a clause in the language. An example is an *interpretation* listing a domain of elements and the extension of predicates over them. The example e_1=$([1, 2, 3], [[p(1, 2), p(2, 3), p(3, 1), q(1, 3)]])$ describes an interpretation with domain $[1, 2, 3]$ and where the four atoms listed are true in the interpretation and other atoms are false. The size of an example is the number of atoms true in it, so that $size(e_1)$=4. The example e_1 falsifies the clause above (substitute $\{1/x_1, 2/x_2, 3/x_3\}$), so it is a negative example. On the other hand e_2=$([a, b, c, d], [[p(a, b), p(b, c), p(a, c), p(a, d), q(a, c)]])$ is a positive example. We use standard notation $e_1 \not\models c_1$ and $e_2 \models c_1$ for these facts. Our system (the batch algorithm of [13]) performs the standard supervised learning task: given a set of positive and negative examples it produces a Horn expression as its output.

3 The Base System

We first review the basic features of the algorithm and system as described in [13]. The algorithm works by constructing an intermediate hypothesis and repeatedly refining it until the hypothesis is consistent with the data. The algorithm's starting point is the most specific *clause set* that "covers" a particular negative example. A clause set is a set of Horn clauses that have the same antecedent but different conclusions. In this paper, we use $[s, c]$ and variations of it to denote a clause set, where s is a set of atoms (the antecedent) and c is a set of atoms (each being a consequent of a clause in the clause set). Once the system has some such clauses it searches the dataset for a misclassified example. Upon finding one (it is guaranteed to be a negative example) the system tries to refine one of its clause sets using a generalization operation which we call *pairing*. Pairing is an operation akin to LGG [24] but it controls the size of the hypothesis by using a restriction imposed by a one to one object correspondence. If pairing succeeds, that is, the refinement is found to be good, the algorithm restarts the search for misclassified examples. If pairing did not produce a good clause, the system adds a new most specific clause set to the hypothesis. This process of refinements continues until no more examples are misclassified.

To perform the above the system needs to refer to the dataset in order to evaluate whether the result of refinements, a proposed clause set, is useful or not. This is performed by an operation we call *one-pass* . In addition the algorithm uses an initial "minimization" stage where candidate clause sets are reduced in size. The high level structure of the algorithm is given in Figure 3. We proceed with details of the various operations as required by the algorithm.

Candidate clauses: For an interpretation I, $rel\text{-}ant(I)$ is a conjunction of positive literals obtained by listing all atoms true in I and replacing each object in I with a distinct variable. So $rel\text{-}ant(e_1) = p(x_1, x_2) \wedge p(x_2, x_3) \wedge p(x_3, x_1) \wedge q(x_1, x_3)$. Let X be the set of variables corresponding to the domain of I in this transformation. The set of candidate clauses $rel\text{-}cands(I)$ includes clauses of the form $(rel\text{-}ant(I) \rightarrow p(Y))$, where p is a predicate, Y is a tuple of variables from X of the appropriate arity and $p(Y)$ is not in $rel\text{-}ant(I)$. For example, $rel\text{-}cands(e_1)$ includes among others the clauses $[p(x_1, x_2) \wedge p(x_2, x_3) \wedge p(x_3, x_1) \wedge q(x_1, x_3) \rightarrow$

```
1. Initialize $S$ to be the empty sequence.
2. Repeat until $H$ is correct on all examples in $E$.
   (a) Let $H = variabilize(S)$.
   (b) If $H$ misclassifies $I$ ($I$ is negative but $I \models H$):
       i. $[s, c] = one\text{-}pass(rel\text{-}cands(I))$.
       ii. $[s, c] = minimize\text{-}objects([s, c])$.
       iii. For $i = 1$ to $m$ (where $S = ([s_1, c_1], \ldots, [s_m, c_m])$)
              For every pairing $J$ of $s_i$ and $s$
              If $J$'s size is smaller than $s_i$'s size then
                  let $[s, c] = one\text{-}pass([J, c_i \cup (s_i \setminus J)])$.
                  If $c$ is not empty then
                 A. Replace $[s_i, c_i]$ with $[s, c]$.
                 B. Quit loop (Go to Step 2a)
       iv. If no $s_i$ was replaced then add $[s, c]$ as the last element of $S$.
```

Fig. 1. The Learning Algorithm (Input: example set E)

$p(x_2, x_2)]$, and $[p(x_1, x_2) \wedge p(x_2, x_3) \wedge p(x_3, x_1) \wedge q(x_1, x_3) \rightarrow q(x_3, x_1)]$, where all variables are universally quantified. Note that there is a 1-1 correspondence between a ground clause set $[s, c]$ and its variabilized versions. We refer to the variabilization using *variabilize*$(\cdot)$. In the following we just use $[s, c]$ with the implicit understanding that the appropriate version is used.

As described above, any predicate in the signature can be used as a consequent by the system. However, in specific domains the user often knows which predicates should appear as consequents. To match this, the system allows the user to specify which predicates are allowed as consequents of clauses. Naturally, this improves run time by avoiding the generation, validation and deletion of useless clauses.

The one-pass procedure: Given a clause set $[s, c]$ *one-pass* tests clauses in $[s, c]$ against all positive examples in E. The basic observation is that if a positive example can be matched to the antecedent but one of the consequents is false in the example under this matching then this consequent is wrong. For each example e, the procedure *one-pass* removes *all* wrong consequents identified by e from c. If c is empty at any point then the process stops and $[s, \emptyset]$ is returned. At the end of *one-pass* , each consequent is correct w.r.t. the dataset.

This operation is at the heart of the algorithm since the hypothesis and candidate clause sets are repeatedly evaluated against the dataset. Two points are worth noting here. First, once we match the antecedent we can test all the consequents simultaneously so it is better to keep clause sets together rather than split them into individual clauses. Second, notice that since we must verify that consequents are correct, it is not enough to find just one substitution from an example to the antecedent. Rather we must check all such substitutions before declaring that some consequents are not contradicted. This is an issue that affects the implementation and will be discussed further below.

Minimization: Minimization takes a clause set $[s, c]$ and reduces its size so that it includes as few objects as possible while still having at least one correct consequent. This is done by "dropping objects". For example, for $[s, c] = [[p(1, 2), p(2, 3), p(3, 1), q(1, 3)], [p(2, 2), q(3, 1)]]$, we can drop object 1 and all atoms using it to get $[s, c] = [[p(2, 3)], [p(2, 2)]]$. The system iteratively tries to drop each domain element. In each iteration it drops an object to get $[s', c']$, runs *one-pass* on $[s', c']$ to get $[s'', c'']$. If c'' is not empty it continues with it to the next iteration (assigning $[s, c] \leftarrow [s'', c'']$); otherwise it continues with $[s, c]$.

Pairing: The pairing operation combines two clause sets $[s_a, c_a]$ and $[s_b, c_b]$ to create a new clause set $[s_p, c_p]$. When pairing we utilize an injective mapping from the smaller domain to the larger one. The system first pairs the antecedents by taking the intersection under the injective mapping (using names from $[s_a, c_a]$) to produce a new antecedent J. The resulting clause set is $[s_p, c_p] = [J, (c_a \cap c_b) \cup (s_a \setminus J)]$. To illustrate this, the following example shows the two original clauses, a mapping and the resulting values of J and $[s_p, c_p]$.

- $[s_a, c_a] = [[p(1, 2), p(2, 3), p(3, 1), q(1, 3)], [p(2, 2), q(3, 1)]]$
- $[s_b, c_b] = [[p(a, b), p(b, c), p(a, c), p(a, d), q(a, c)], [q(c, a)]]$
- The mapping $\{1/a, 2/b, 3/c\}$
- $J = [p(1, 2), p(2, 3), q(1, 3)]$
- $[s_p, c_p] = [[p(1, 2), p(2, 3), q(1, 3)], [q(3, 1), p(3, 1)]$

The clause set $[s_p, c_p]$ obtained by the pairing can be more general than the original clause sets $[s_a, c_a]$ and $[s_b, c_b]$ since s_p is contained in both s_a and s_b (under the injective mapping). Hence, the pairing operation can be intuitively viewed as a generalization of both participating clause sets. However since we modify the consequent, by dropping some atoms and adding other atoms (from $s_a \setminus J$), this is not a pure generalization operation.

Clearly, any two examples have many possible pairings, one for each injective mapping of domain elements. The system reduces the number of pairings that are tested without compromising correctness as follows. We say that a mapping is *live* if every paired object appears in the extension of at least one atom in J. For example, the pairing given above is live but the mapping $\{1/c, 2/b, 3/a\}$ results in $J = [p(3, 1)]$ so object 2 does not appear in J and the pairing is not live. The system only tests live mappings and generates these from the clause sets so that non-live pairings are not enumerated. As pointed out in [13] if the target expression is range restricted (i.e. all variables in the consequent appear in the antecedent) then testing live mappings is sufficient. The new system is restricted to this case.

Caching: The operation of the algorithm may produce repeated calls to *one-pass* with the same antecedent since pairings of one clause set with several others may result in the same clause set. Thus it makes sense to cache the results of *one-pass* . Notice that there is a tradeoff in the choice of what to cache. If we try to cache a universally quantified expression then matching it requires a subsumption test which is expensive. We therefore opted to cache a ground syntactic version of the clause. In fact, the system caches interpretations rather than clauses or clause sets (i.e only the s_i part of $[s, c]$). For our purpose we only need to cache positive interpretations - if a clause set $[s, \emptyset]$ was returned by one pass then s is a positive interpretation (it does not imply any of the possible consequents). Thus any new call to one pass with s can be skipped. To achieve fast caching while increasing the chances of cache hits, the system caches and compares a normalized representation of the interpretation by sorting predicate and atom names. This is matched with the fact that pairing keeps object names of existing clause sets in the hypothesis. Thus the same object names and ordering of these are likely to cause cash hits. Caching can reduce or increase run time of the system, depending on the dataset, the cost for subsumption for examples in the dataset, and the rate of cache hits.

Implementation: [13] used an implementation of these ideas in Prolog. Our new system includes all the above, implemented in C with in an attempt to improve the representation and run time. In addition the system includes new features including several subsumption engines, discretization, and pruning that make it applicable in larger problems.

4 Performance Issues and Applicability

While the results reported in [13] were encouraging several aspects precluded immediate application to real world data sets.

The Subsumption Test: Perhaps the most important issue is run time which is dominated by the cost of subsumption tests and the *one-pass* procedure. It is well known that testing subsumption is NP-Hard [14] and therefore we do not expect a solution in the general case. However it is useful to look at the crucial parameters. In general subsumption scales exponentially in the number of variables in the clauses but polynomially with the number of predicates [23]. The problem is made worse in our system because of the bottom-up nature of the learning process[1]. When we generate the most specific clause for an example, the number of variables in the clause is the same as the number of objects in the example and this can be quite large in some domains. In some sense the minimization process tries to overcome this problem by removing as many objects as possible (this fact is used in [12] to prove good complexity bounds). However the minimization process itself runs *one-pass* and therefore forms a bottleneck. In addition, for *one-pass* it is important to find all substitutions between a clause and an example. Therefore, a normal subsumption test that checks for the existence of a substitution is not sufficient. For problems with highly redundant structure the number of substitutions can grow exponentially with the number of predicates so this can be prohibitively expensive. Thus an efficient solution for *one-pass* is crucial in applications with large examples.

[1] The same is true for other bottom up systems; see for example discussion in [26].

Suitability of Datasets and Overfitting: The system starts with the most specific clauses and then removes parts of them in the process of generalization. In this process, a subset of an interpretation that was a negative example becomes the s in the input to *one-pass* . If examples matching s exist in the dataset then we may get a correct answer from *one-pass* . In fact if a dataset is "downward closed", that is all such subsets exist as examples in the dataset, the system will find the correct expression. Note that we only need such subsets which are positive examples to exist in the dataset and that it is also sufficient to have isomorphic embeddings of such subsets in other positive examples as long as wrong consequents are missing. Under these conditions all calls to *one-pass* correctly identify all consequents of the clause. Of course, this is a pretty strong requirement but as demonstrated by experiments in [13] having a sample from interpretations of different sizes can work very well.

If this is not the case, e.g. in the challenge problems of [11] where there is a small number of large examples, then we are not likely to find positive examples matching subsets of the negative ones (at least in the initial stages of minimization) and this can lead to overfitting. This has been observed systematically in experiments in this domain.

Using Examples from Normal ILP setting: In the normal ILP setting [20] one is given a database as background knowledge and examples are simple atoms. We transform these into a set of interpretations as follows (see also [5, 12]). The background knowledge in the normal ILP setting can be typically partitioned into different subsets such that each subset affects a single example only. A similar effect is achieved for intensional background knowledge in the Progol system [18] by using mode declarations to limit antecedent structure. Given example b, we will denote $BK(b)$ as the set of atoms in the background knowledge that is relevant to b. In the normal ILP setting we have to find a theory T s.t. $BK(b) \cup T \models b$ if b is a positive example, and $BK(b) \cup T \not\models b$ if b is negative. Equivalently, T must be such that $T \models BK(b) \rightarrow b$ if b is positive and $T \not\models BK(b) \rightarrow b$ if b is negative.

If b is a positive example in the standard ILP setting then we can construct an interpretation $I = ([V], [BK(b)])$ where V is the set of objects appearing in $BK(b)$, and label I as negative. When LogAn-H finds the negative interpretation I, it constructs the set $\textit{rel-ant}(I) \rightarrow p(Y)$ from it (notice that b is among the $p(Y)$ considered in this set), and then runs *one-pass* to figure out which consequents among the candidates are actually correct. Adding another interpretation $I' = ([V], [BG(b) \cup \{b\}])$ labeled positive guarantees that all other consequents are dropped. Notice that in order for this to be consistent with the target concept, we have to assume that the antecedent $BG(b)$ only implies b.

If b is a negative example in the standard ILP setting, we construct an interpretation $I = ([V], [BG(b)])$, where V is the set of variables appearing in $BG(b)$, and label it positive. Notice that if the system ever considers the clause $BG(b) \rightarrow b$ as a candidate, *one-pass* will find the positive interpretation I and will drop b, as desired. This again assumes that no consequent can be implied by $BG(b)$.

Several ILP domains are formalized using a consequent of arity 1 where the argument is an object that identifies the example in the background knowledge. In this case, since we separate the examples into interpretations we get a consequent of arity 0. For learning with a single possible consequent of arity 0 our transformation can be simplified in that the extra positive example $I' = ([V], [BG(b) \cup \{b\}])$ is not needed since there are no other potential consequents. Thus we translate every positive example into a negative interpretation example and vice versa. As an example, suppose that in the normal ILP setting, the clause $p(a, b) \wedge p(b, c) \rightarrow q()$ is labeled positive and the clause $p(a, b) \rightarrow q()$ is labeled negative. Then, the transformed dataset contains: $([a, b, c], [p(a, b), p(b, c)])-$ and $([a, b], [p(a, b)])+$. Notice that in this case the assumptions made regarding other consequents in the general transformation are not needed.

In the case of zero arity consequents, the check whether a given clause C is satisfied by some interpretation I can be considerably simplified. Instead of checking all substitutions it suffices to check for existence of some substitution, since any such substitution will remove the single nullary consequent. This has important implications for the implementation. In addition, note that the pairing operation never moves new atoms into the consequent and is therefore a pure generalization operation in this case.

5 Further Improvements

5.1 The Subsumption Test

Table Based Subsumption: While backtracking search (as done in Prolog) can find all substitutions without substantial space overhead, the time overhead can be large. Our system implements an alternative approach that constructs all substitutions simultaneously and stores them in memory. The system maintains a table of instantiations for each predicate in the examples. To compute all substitutions between an example and a clause the system repeatedly performs joins of these tables (in the database sense) to get a table of all substitutions. We first initialize to an empty table of substitutions. Then for each predicate in the clause we pull the appropriate table from the example, and perform a join which matches the variables already instantiated in our intermediate table. Thus if the predicate in the clause does not introduce new variables the table size cannot grow. Otherwise the table can grow and repeated joins can lead to large tables. To illustrate this consider evaluating the clause $p(x_1, x_2), p(x_2, x_1), p(x_1, x_3), p(x_3, x_4)$ on an example with extension $[p(a, b), p(a, c), p(a, d), p(b, a), p(d, c)]$. Then applying the join from left to right we get partial substitution tables (from left to right):

x_1	x_2	x_3	x_4
a	b		
a	c		
a	d		
b	a		
d	c		

x_1	x_2	x_3	x_4
a	b		
b	a		

x_1	x_2	x_3	x_4
a	b	b	
a	b	c	
a	b	d	
b	a	a	

x_1	x_2	x_3	x_4
a	b	b	a
a	b	d	c
b	a	a	b
b	a	a	c
b	a	a	d

Notice how the first application simply copies the table from the extension of the predicate in the example. The first join reduces the size of the intermediate table. The next join expands both lines. The last join drops the row with `a b c` but expands other rows so that overall the table expands.

One can easily construct examples where the table in intermediate steps is larger than the memory capacity of the computer, sometimes even if the final table is small. In this case the matching procedure will fail. If it does not fail, however, the procedure can be fast since we have no backtracking overhead and we consider many constraints simultaneously.

Nonetheless this is not something we can ignore. We have observed such large table sizes in the mutagenesis domain [29] as well as the artificial challenge problems of [11]. Note that a backtracking search will not crash in this case but on the other hand it may just take too long computationally so it is not necessarily a good approach (this was observed in the implementation of [13]).

Lookahead: As in the case of database queries one can try to order the joins in order to optimize the computation time and space requirements. Our system can perform a form of one step lookahead by estimating the size of the table when using a join with each of the atoms on the clause and choosing the minimal one. This introduces a tradeoff in run time. On one hand the resulting tables in intermediate steps tend to be smaller and therefore there is less information to process and the test is quicker. On the other hand the cost of one step lookahead is not negligible so it can slow down the program. The behavior depends on the dataset in question. In general however it can allow us to solve problems which are otherwise unsolvable with the basic approach.

Randomized Table Based Subsumption: If the greedy solution is still not sufficient or too slow we can resort to randomized subsumption tests. Instead of finding all substitutions we try to sample from the set of legal substitutions. This is done in the following manner: if the size of the intermediate table grows beyond a threshold parameter 'TH' (controlled by the user), then we throw away a random subset of the rows before continuing with the join operations. The maximum size of intermediate tables is TH×16. In this way we are not performing a completely random choice over possible substitutions. Instead we are informing the choice by our intermediate table. In addition the system uses random restarts to improve confidence as well as allowing more substitutions to be found, this can be controlled by the user through a parameter 'R'.

Using Django**:** The system Django [16] uses ideas from constraint satisfaction to solve the subsumption problem. Django only solves the existence question and does not give all substitutions, but as discussed above this is sufficient for certain applications coming from the standard ILP setting. We have integrated the Django code, generously provided by Jérome Maloberti, as a module in our system.

5.2 Discretization

The system includes a capability for handling numerical data by means of discretization. Several approaches to discretization have been proposed in the literature [10, 9, 2]. We have implemented the simple "equal frequency" approach that generates a given number of bins (specified by the user) and assigns the boundaries by giving each bin the same number of occurrences of values.

To do this for relational data we first divide the numerical attributes into "logical groups". For example the rows of a chess board will belong to the same group regardless of the predicate and argument in which they appear. This generalizes the basic setup where each argument of each predicate is discretized separately. The dataset is annotated to reflect this grouping and the preferred number of thresholds is specified by the user. The system then determines the threshold values, allocates the same name to all objects in each range, and adds predicates reflecting the relation of the value to the thresholds. For example, discretizing the `logp` attribute in the mutagenesis domain with 4 thresholds (5 ranges), a value between threshold 1 and threshold 2 will yield: [`logp(logp_val.02)`, `logp_val>00(logp_val.02)`, `logp_val>01(logp_val.02)`, `logp_val<02(logp_val.02)`, `logp_val<03(logp_val.02)`, ...].

Notice that we are using both $\geq$ and $\leq$ predicates so that the hypothesis can encode intervals of values.

An interesting aspect arises when using discretization which highlights the way our system works. Recall that the system starts with an example and essentially turns objects into variables in the maximally specific clause set. It then evaluates this clause on other examples. Since we do not expect examples to be identical or very close, the above relies on the universal quantification to allow matching one structure into another. However, the effect of discretization is to ground the value of the discretized object. For example, if we discretized the `logp` attribute from above and variabilize we get `logp(X) logp_val>00(X) logp_val>01(X) logp_val<02(X) logp_val<03(X)`. Thus unless we drop some of the boundary constraints this limits matching examples to have a value in the same bin. We are therefore losing the power of universal quantification. As a result less positive examples will match in the early stages of the minimization process, less consequents will be removed, and the system may be led to overfitting by dropping the wrong objects. This is discussed further in the experimental section.

5.3 Pruning

The system performs bottom-up search and may stop with relatively long rules if the data is not sufficiently rich (i.e. we do not have enough negative examples) to warrant further refinement of the rules. Pruning allows us to drop additional parts of rules. The system can perform a greedy reduced error pruning [17] using a validation dataset. For each atom in the rule (in some order) the system evaluates whether the removal of the atom increases the error on the validation set. If not the atom can be removed. While it is natural to allow an increase in error using a tradeoff against the length of the hypothesis in an MDL fashion, we have not yet experimented with this possibility.

Notice that unlike top down systems we can perform this pruning on the training set and do not necessarily need a separate validation set. In a top down system one grows the rules until they are consistent with the data. Thus, any pruning will lead to an increase in training set error. On the other hand in a bottom up system, pruning acts like the main stage of the algorithm in that it further generalizes the rules. In some sense, pruning on the training set allows us to move from a most specific hypothesis to a most general hypothesis that matches the data. Both training set pruning and validation set pruning are possible with our system.

5.4 Consistency Checks

If the input dataset is inconsistent, step (i) of the algorithm may produce an initial version of the most specific clauses set with an empty list of consequents. Similar problems may arise with the randomized subsumption tests. The system includes simple mechanisms for ignoring such examples once a problem is detected.

6 Experiments

The LogAn-H system of [13] implements the algorithm in Section 3 using Prolog and its backtracking search engine. Our new system includes a C implementation of the ideas described above.

6.1 Bongard Problems

To illustrate the improvement in efficiency of the new system w.r.t. the previous implementation, we re-ran experiments done with artificial data akin to Bongard problems [13]. This domain was introduced previously in the ICL system [7]. In this domain an example is a "picture" composed of objects of various shapes (triangle, circle or square), triangles have a configuration (up or down) and each object has a color (black or white). Each picture has several objects (the number is not fixed) and some objects are inside other objects. For these experiments we generated random examples, where each parameter in each example was chosen uniformly at random. In particular we used between 2 and 6 objects, the shape color and configuration were chosen randomly, and each object is inside some other object with probability 0.5 where the target was chosen randomly among "previous" objects to avoid cycles. Note that since we use a "flattened" function free representation the domain size in examples is larger than the number of objects (to include: *up*, *down*, *black*, *white*). We generated (by hand) a target Horn expression of 10 clauses, with 9 atoms and 6 variables each. We used this Horn expression to label the examples. For example, one of the clauses generated in the target expression is

Table 1. Accuracy in the Bongard domain (reproduced from [13])

System	200	500	1000	2000	3000	*bl*
LogAn-H	85.9	92.8	96.8	98.4	98.9	84.7
ICL	85.2	88.6	89.1	90.2	90.9	84.7

$circle(X)$ $in(X,Y)$ $in(Y,Z)$ $colour(Y,B)$
$colour(Z,W)$ $black(B)$ $white(W)$ $in(Z,U) \rightarrow triangle(Y)$

We ran LogAn-H on several sample sizes. Table 1 summarizes the accuracy of learned expressions as a function of the size of the training set (200 to 3000) when tested on classifying an independent set of 3000 examples. The last column in the table gives the majority class percentage (marked *bl* for baseline). Each entry is an average of 10 independent runs where a new set of random examples is used in each run. We give accuracy results for both LogAn-H and ICL taken from [13]. The new system obtains exactly the same accuracy as before[2] and the speedup observed is between one and three orders of magnitude over the `Prolog` system in compiled Sicstus Prolog (which is a fast implementation) when run on the same hardware.

6.2 Illegal Positions in Chess

Our next experiment is in the domain of the chess endgame White King and Rook versus Black King. The task is to predict whether a given board configuration represented by the 6 coordinates of the three chess pieces is illegal or not. This learning problem has been studied by several authors [22, 25]. The dataset includes a training set of 10000 examples and a test set of the same size.

We use the predicate `position(a,b,c,d,e,f)` to denote that the White King is in position (`a`, `b`) on the chess board, the White Rook is in position (`c`, `d`), and the Black King in position (`e`, `f`). Additionally, the predicates "less-than" `lt(x,y)` and "adjacent" `adj(x,y)` denote the relative positions of rows and columns on the board. Note that there is an interesting question as how best to capture examples in interpretations. In "all background mode" we include all `lt` and `adj` predicates in the interpretation. In the "relevant background mode" we only include those atoms directly relating objects appearing in the head.

We illustrate the difference with the following example. Consider the configuration "White King is in position (7,6), White Rook is in position (5,0), Black King is in position (4,1)" which is illegal. In "all background mode" we use the following interpretation:

```
[position(7, 6, 5, 0, 4, 1),
lt(0,1), lt(0,2), .. ,lt(0,7),
lt(1,2), lt(1,3), .. ,lt(1,7),
⋮
```

[2] Note that the hypothesis may depend on the order of pairings produced so in principle the results are not guaranteed to be identical.

Table 2. Performance summary for KRK-illegal dataset

	25	50	75	100	200	500	1000	2000	3000
w/o disc., rel. back. mode:									
LogAn-H before pruning	75.49	88.43	93.01	94.08	97.18	99.54	99.79	99.92	99.96
LogAn-H after pruning	86.52	90.92	94.19	95.52	98.41	99.65	99.79	99.87	99.96
w/o disc., all back. mode:									
LogAn-H before pruning	67.18	71.08	75.71	78.94	85.56	94.06	98.10	99.38	99.56
LogAn-H after pruning	79.01	81.65	83.17	82.82	86.02	93.67	96.24	98.10	98.66
with disc., rel. back. mode:									
LogAn-H before pruning	43.32	43.70	45.05	44.60	52.39	72.26	84.80	90.30	92.17
LogAn-H after pruning	38.93	42.77	46.46	47.51	56.59	74.29	85.02	90.73	92.59
with disc., all back. mode:									
LogAn-H before pruning	67.27	72.69	75.15	78.00	82.68	88.60	91.03	91.81	92.01
LogAn-H after pruning	80.62	86.14	87.42	89.10	90.67	92.25	92.62	92.66	92.74
FOIL [25]				92.50			99.40		

```
lt(5,6),lt(5,7),
lt(6,7),
adj(0,1),adj(1,2), .. ,adj(6,7),
adj(7,6),adj(6,5), .. ,adj(1,0)]-
```

When considering the "relevant background mode", we include in the examples instantiations of `lt` and `adj` whose arguments appear in the position atom directly:

```
[position(7, 6, 5, 0, 4, 1),
lt(4,5),lt(4,7),lt(5,7),adj(4,5),adj(5,4),
lt(0,1),lt(0,6),lt(1,6),adj(0,1),adj(1,0)]-
```

Table 2 includes results of running our system in both modes. We trained LogAn-H on samples with various sizes chosen randomly among the 10000 available. We report accuracies that result from averaging among 10 runs over an independent test set of 10000 examples. Results are reported before and after pruning where pruning is done using the training set. Several facts can be observed in the table. First, we get good learning curves with accuracies improving with training set size. Second, the results obtained are competitive with results reported for FOIL [25]. Third, relevant background knowledge seems to make the task easier. Fourth, pruning considerably improves performance on this dataset especially for small training sets.

Our second set of experiments in this domain illustrates the effect of discretization. We have run the same experiments as above but this time with the discretization option turned on. Concretely, given an example's predicate `position(x1,x2,y1,y2,z1,z2)`, we consider the three values corresponding to columns `(x1,y1,z1)` as the same logical attribute and therefore we discretize them together. Similarly, we discretize the values of `(x2,y2,z2)` together. Versions of `adj()` for both column and row values are used. We do not include `lt()` predicates since these are essentially now represented by the threshold predicates produced by the discretization. As can be seen in Table 2 good accuracy is

Table 3. Runtime comparison for subsumption tests on KRK-illegal dataset

Subsumption Engine	*runtime in s.*	*accuracy*	*actual table size*
Django	431.6	98.11%	
Tables	19.2	98.11%	130928
Lookahead	25.4	98.11%	33530
No cache	49.4	98.11%	
Rand. TH=1	741.7	33.61%	16
Rand. TH=10	30.7	33.61%	160
Rand. TH=100	12.4	72.05%	1600
Rand. TH=1000	20.3	98.11%	16000

maintained with discretization. However, an interesting point is that now "relevant background mode" performs much worse than "all background mode". In hindsight one can see that this is a result of the grounding effect of discretizing as discussed above. With "relevant background mode" the discretization threshold predicates and the adjacent predicates are different in every example. Since, as explained above, the examples are essentially ground we expect less matches between different examples and thus the system is likely to overfit. With "all background mode" these predicates do not constrain the matching of examples.

This domain is also a good case to illustrate the various subsumption tests in our system. Note that since we put the position predicate in the antecedent the consequent is nullary. Therefore we can use Django as well as the table based subsumption and randomized tables. The comparison is given for the non-discretized "all background mode" with 1000 training examples. Table 3 gives accuracy and run time (on Linux running with Pentium IV 2.80 GHz) for various subsumption settings averaged over 10 independent runs. For randomized runs TH is the threshold of table size after which sampling is used. As can be seen, the table based method is faster than Django (both are deterministic and thus give identical hypotheses and accuracy results). The lookahead table method incurs some overhead and results in slower execution on this domain, however it saves space considerably (see third column of Table 3). Caching gives a reduction of about 60% in run time. Running the randomized test with very small tables (TH=1) clearly leads to overfitting, and in this case increases run time considerably mainly due do the large number of rules induced. On the other hand with small tables sizes (TH=1000) the randomized method does very well and reproduces the deterministic results.

6.3 Mutagenesis

The Mutagenesis dataset is a structure-activity prediction task for molecules introduced by [29]. The dataset consists of 188 compounds, labeled as active or inactive depending on their level of mutagenic activity. The task is to predict whether a given molecule is active or not based on the first-order description of the molecule. This dataset has been partitioned into 10 subsets for 10-fold cross validation estimates and has been used in this form in many studies (e.g. [29, 26,

Table 4. Runtime comparison for subsumption tests on mutagenesis dataset

Subsumption Engine	*runtime in s.*	*accuracy*
Django	1162	87.96%
Rand. TH=1	3	85.52%
Rand. TH=10	15	86.46%
Rand. TH=100	19	89.47%

7]). We therefore use the same partitions as well. Each example is represented as a set of first-order atoms that reflect the atom-bond relation of the compounds as well as some interesting global chemical properties. Concretely, we use all the information corresponding to the background level B3 of [28]. Notice that the original data is given in the normal ILP setting and hence we transformed it as described above using a single nullary consequent. In addition, since constants are meaningful in this dataset (for example whether an atom is a carbon or oxygen) we use a flattened version of the data where we add a predicate for each such constant.

This example representation uses continuous attributes (`atom-charge`, `lumo` and `logp` in particular), hence discretization is needed. Although the discretization process is fully automated it requires the number of discrete categories to be specified by the user. Here, we use a method that allows us to determine this number automatically and without any use of the test set: for each partition of the cross validation, we split the training data into two random sets, one which we call `disc-train` and consists of 80% of the training data, and another called `disc-test` which consists of the remaining training data. Then, for each of the possible values (`atom-charge`$= 5, 15, 25, 35, 45$; `lumo`$= 4, 6, 8, 10$; `logp`$= 4, 6, 8, 10$) we train and test over the sets `disc-train` and `disc-test`. This procedure is repeated 5 times and we choose the discretization values that obtain the best average accuracy on this partition. Note that these values might be different for different partitions of the global cross validation and indeed we did not get a stable choice. Once a set of values is chosen for a particular partition of the data, the learning process is performed over the entire training set and then it is tested on the corresponding independent test set.

For this domain deterministic table-based subsumption was not possible, not even with the lookahead heuristic since the table size grew beyond memory capacity of our computer. However, here the Django subsumption engine yields good run times. The average training time per fold, after the discretization sizes have been determined, is 14 min. (on Linux running with Pentium IV 2.80 GHz). Prediction accuracies obtained for each partition in this fashion are (in order from 1 to 10): 73.68%, 89.47%, 78.95%, 84.21%, 84.21%, 89.47%, 89.47%, 73.68%, 73.68%, 88.24%, which results in a final average of 82.5%. Additionally, we ran a regular 10-fold cross-validation for each combination of discretization values. The values `atom-charge`$= 45$, `lumo`$= 10$ and `logp`$= 4$ obtained the best average accuracy of 87.96%. Our result compares well to other ILP systems: PROGOL [29] reports a total accuracy of 83% with B3 and 88% with B4; STILL [26]

Table 5. Subsumption run time in linear chain family

	Django	Tables	Lookahead	TH=1	TH=10	TH=100
	100.0% 296s	100.0% 242s (14161)	100.0% 318s (118)			
R=1				6.9% 13s	18.6% 49s	100.0% 240s
R=10				32.2% 60s	66.6% 181s	100.0% 243s
R=100				96.9% 185s	100.0% 280s	100.0% 241s

reports results in the range 85%–88% on B3 depending on the values of various tuning parameters, ICL [7] reports an accuracy of 84% and finally [15] report that FOIL [25] achieves an accuracy of 83%.

Here again we ran further experiments with the randomized subsumption tests. We used the discretization values `atom-charge`= 45, `lumo`= 10 and `logp`= 4. Table 4 gives run time (on Linux running with Pentium IV 2.80 GHz) per fold and the 10 fold cross validation accuracy with various parameters. One can observe that even with small parameters the randomized methods do very well. An inspection of the hypothesis to the deterministic runs with Django shows that they are very similar.

6.4 Evaluating Randomized Subsumption Tests

The experiments above already show that there are cases where the table based method is fast and faster than Django even though it searches for all substitutions compared to just one in Django. On the other hand the table based method can be slow in other cases and even run out of memory and fail. The following experiments give simple synthetic examples where we compare the subsumption tests on their own, without reference to the learning system, showing similar behavior. In each case we generate a family of problems parametrized by size, each having a single example and single clause. We run the subsumption test 1000 times to observe run time differences as well as accuracies for the randomized methods.

For the first family both example and clause are chains of length n built using a binary predicate as in $p(x_1, x_2), p(x_2, x_3), \ldots, p(x_{n-1}, x_n)$. Thus there is exactly one matching substitution. Results for $n = 120$ are given in Table 5. As can be seen, in this case tables are faster than Django, randomized tables work well with small parameters, and both table size (marked with TH in Table 5) and repeats (marked with R in Table 5) are effective in increasing the performance of the randomized tests. This behavior was observed consistently for different values of n. The numbers in parentheses are the actual table sizes needed by the table-based methods; the lookahead heuristic saves considerable space.

Table 6. Subsumption run time in subgraph isomorphism family

	Django	TH=1	TH=10	TH=100	TH=1000
	100.00% 7.1s				
R=1		0.01% 0.8s	3.21% 1.7s	31.71% 8.9s	85.46% 52.4s
R=10		0.03% 2.6s	15.29% 7.0s	78.95% 26.8s	95.12% 76.1s
R=100		0.22% 23.6s	67.88% 38.9s	99.92% 39.1s	99.97% 103.1s

The second family is motivated by the mutagenesis domain and essentially checks for subgraph isomorphism. The clause is a randomly generated graph with n nodes and $3n$ edges, and the example is the same set plus $3n$ extra edges. The results for $n = 10$ are given in Table 6. Deterministic tables fail for values of n larger than 8 and are omitted. As can be seen Django works very well in this case and randomized tables work well even with small parameters, and both table size and repeats (marked with TH and R in Table 6, resp.) are effective in increasing the performance of the randomized tests. Similar results were obtained for different values of n where randomized tables sometimes achieve high accuracy with lower run times than Django though in general Django is faster.

7 Discussion

The paper presents a new implementation of the LogAn-H system including new subsumption engines, discretization and pruning. Interesting aspects of discretization and pruning which are specific to bottom up search are discussed in the paper. The system is sufficiently strong to handle large ILP datasets and is shown to be competitive with other approaches while using a completely different algorithmic approach. The paper also demonstrates the merits of having several subsumption engines at hand to fit properties of particular domains, and gives paradigmatic cases where different engines do better than others.

As illustrated in [13] using the Bongard domain, LogAn-H is particularly suited to domains where substructures of examples in the dataset are likely to be in the dataset as well. On the other hand, for problems with a small number of examples where each example has a large number of objects and dramatically different structure our system is likely to overfit since there is little evidence for useful minimization steps. Indeed we found this to be the case for the the artificial challenge problems of [11] where our system outputs a large number of rules and gets low accuracy. Interestingly, a similar effect can result from discretization since it results in a form of grounding of the initial clauses and thus counteracts the fact that they are universally quantified and thus likely to be contradicted by the dataset if wrong. This suggests that skipping the minimization step may lead to improved performance in such cases if pairings reduce clause size considerably. Initial experiments with this are as yet inconclusive.

Our experiments demonstrated the utility of informed randomized subsumption tests. Another interesting possibility is to follow ideas from the successful randomized propositional satisfiability tester WalkSat [27]. Here one can abandon the table structure completely and search for a single substitution using a random walk over substitutions where in each step we modify an unsuccessful substitution to satisfy at least one more atom. Repeating the above can improve performance as well as find multiple substitutions when needed. Initial experiments suggest that this indeed can be useful albeit our current implementation is slow. It would be interesting to explore this further in LogAn-H and other systems.

Our system also demonstrates that using large refinement steps with a bottom up search can be an effective inference method. As discussed above, bottom up search suffers from two aspects: subsumption tests are more costly than in top down approaches, and overfitting may occur in small datasets with large examples. On the other hand, it is not clear how large refinement steps or insights gained by using LGG can be used in a top down system. One interesting idea in this direction is given in the system of [1]. Here repeated pairing-like operations are performed without evaluating the accuracy until a syntactic condition is met (this is specialized for the challenge problems of [11]) to produce a short clause. This clause is then used as a seed for a small step refinement search that evaluates clauses as usual. Finding similar ideas that work without using special properties of the domain is an interesting direction for future work.

Acknowledgments

Much of the code for LogAn-H was written by Constantine Ashminov as part of his MS work at Tufts. We are grateful to Jérome Maloberti for providing the code for the Django system. Some of the experiments were performed on a Linux cluster provided by Tufts Central Computing Services.

References

[1] Jacques Alès Bianchetti, Céline Rouveirol, and Michèle Sebag. Constraint-based learning of long relational concepts. In *Proceedings of the International Conference on Machine Learning*, pages 35–42. Morgan Kaufmann, 2002.

[2] H. Blockeel and L. De Raedt. Lookahead and discretization in ilp. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–84. Springer-Verlag, 1997.

[3] H. Blockeel and L. De Raedt. Top down induction of first order logical decision trees. *Artificial Intelligence*, 101:285–297, 1998.

[4] I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 38(11):65–70, November 1995.

[5] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95(1):187–201, 1997. See also relevant Errata (forthcoming).

[6] L. De Raedt and S. Dzeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[7] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 6th Conference on Algorithmic Learning Theory*, volume 997. Springer-Verlag, 1995.

[8] N. Di Mauro, T.M.A. Basile, S. Ferilli, F. Esposito, and N. Fanizzi. An exhaustive matching procedure for the improvement of learning efficiency. In T. Horváth and A. Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 112–129. Springer-Verlag, 2003.

[9] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[10] U.M. Fayyad and K. B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the International Conference on Machine Learning*, pages 1022–1027, Amherst, MA, 1993. Morgan Kaufmann.

[11] Attilio Giordana, Lorenza Saitta, Michèle Sebag, and Marco Botta. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.

[12] R. Khardon. Learning function free Horn expressions. *Machine Learning*, 37:241–275, 1999.

[13] Roni Khardon. Learning horn expressions with LogAn-H. In *Proceedings of the International Conference on Machine Learning*, pages 471–478. Morgan Kaufmann, 2000.

[14] J-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237, pages 97–106. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[15] W. Van Laer, H. Blockeel, and L. De Raedt. Inductive constraint logic and the mutagenesis problem. In *Proceedings of the Eighth Dutch Conference on Artificial Intelligence*, pages 265–276, November 1996.

[16] J. Maloberti and Sebag M. Theta-subsumption in a constraint satisfaction perspective. In *Proceedings of the 11th International Conference on Inductive Logic Programming*, pages 164–178. Springer Verlag LNAI 2157, 2001.

[17] T. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[18] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[19] S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In S. Muggleton, editor, *Inductive Logic Programming.* Academic Press, 1992.

[20] S. Muggleton and L. DeRaedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, May 1994.

[21] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[22] S. H. Muggleton, M. Bain, J. Hayes-Michie, and D. Michie. An experimental comparison of human and machine learning formalisms. In *Proc. Sixth International Workshop on Machine Learning*, pages 113–118, San Mateo, CA, 1989. Morgan Kaufmann.

[23] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries (extended abstract). In *Proceedings of the 16th Annual ACM Symposium on Principles of Database Systems*, pages 12–19. ACM Press, 1997.

[24] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

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

[26] Michele Sebag and Celine Rouveirol. Resource-bounded relational reasoning: Induction and deduction through stochastic matching. *Machine Learning*, 38:41–62, 2000.

[27] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 521–532. American Mathematical Society, 1996.

[28] A. Srinivasan, S. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 199–230, 1995.

[29] A. Srinivasan, S. H. Muggleton, R. D. King, and M. J. E. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proc. 4th Int. Workshop on Inductive Logic Programming*, pages 217–232, September 1994.