Learning Horn Expressions with LogAn-H

Marta Arias
Center for Computational Learning Systems
Columbia University
New York, NY 10115, USA
marta@cs.columbia.edu

Roni Khardon
Department of Computer Science
Tufts University
Medford, MA 02155, USA
roni@cs.tufts.edu

Jérôme Maloberti
Laboratorie de Recherche en Informatique
Université Paris-Sud
F-91045, Orsay, France
malobert@lri.fr

March 30, 2005

Abstract

The paper introduces LogAn-H — a system for learning first-order function-free Horn expressions from interpretations. The system is based on an algorithm that learns by asking questions and which was proved correct in previous work. The current paper shows how the algorithm can be implemented in a practical system, and introduces a new algorithm based on it that avoids interaction and learns from examples only. The LogAn-H system implements these algorithms and adds several facilities and optimizations that allow efficient applications in a wide range of problems. As one of the important ingredients the system includes several efficient procedures for solving the subsumption problem, an NP complete problem that needs to be solved many times during the learning process. We describe qualitative and quantitative experiments in several domains. The experiments demonstrate that the system can deal with varied problems, large amounts of data, and that it achieves good classification accuracy.

1 Introduction

The field of Inductive Logic Programming (ILP) deals with the theory and the practice of generating first order logic rules from data. ILP has established a core set of methods and systems that have proved useful in a variety of applications (Muggleton & DeRaedt, 1994; Bratko & Muggleton, 1995). As in much of the work on propositional rule learning, ILP systems that learn rules can be divided into *bottom up* methods and *top down* methods. The latter typically start with an empty rule and grow the rule condition by adding one proposition at a time. Bottom up methods start with a "most specific" rule and iteratively generalize it, for example by dropping propositions from the condition. There are examples of *bottom up* systems in the early days of ILP: e.g., the Golem system (Muggleton & Feng, 1992) used Plotkin's (Plotkin, 1970) least general generalization (LGG) within a bottom up search to find a hypothesis consistent with the data. Related generalization operators were also used by Muggleton and Buntine (1992). On the other hand, much of the research following this (e.g. Quinlan, 1990; Muggleton, 1995; De Raedt & Van Laer, 1995; Blockeel

& De Raedt, 1998) used top down search methods to find useful hypotheses.[1] This paper introduces the system LogAn-H (Logical Analysis for Horn expressions) that implements a bottom up learning algorithm.

The system comes in two main modes. In *batch mode* the system performs the standard supervised learning task, taking a set of labeled examples as input and returning a hypothesis. In *interactive mode* the system learns by asking questions. The questions are the Equivalence Queries and Membership Queries (Angluin, 1988) that have been widely studied in computational learning theory, and in particular also in the context of ILP (Arimura, 1997; Reddy & Tadepalli, 1998; Krishna Rao & Sattar, 1998; Khardon, 1999b; Khardon, 1999a; Arias & Khardon, 2002).

1.1 A Motivating Example: Learning from Graphs

We introduce the problem and some of the algorithmic ideas through a simple example in the context of learning from labeled graphs. In fact this is one of the applications of learning from interpretations where the atom-bond relations of a molecule can be seen as such a graph and this framework has been proved useful for predicting certain properties of molecules.

In the following, node labels are marked by $n_1, n_2, \ldots$, edge labels are $e_1, e_2, \ldots$ and we apply them to nodes or pairs on nodes appropriately. Consider a class of graphs that is characterized by the following rules

$$R_1 = \forall x_1, x_2, x_3, x_4, \quad e_1(x_1, x_2), e_1(x_2, x_3), e_2(x_3, x_4), e_2(x_4, x_1) \rightarrow e_1(x_1, x_3)$$
$$R_2 = \forall x_1, x_2, x_3, x_4, \quad n_1(x_1), n_1(x_2), n_2(x_3), n_3(x_4), e_1(x_1, x_2), e_1(x_1, x_3), e_1(x_1, x_4) \rightarrow e_2(x_2, x_3)$$

Both rules imply the existence of an edge with a particular label if some subgraph exists in the graph. Consider the following graphs that may be part of the data, illustrated in Figure 1:

$$(g_1) = e_1(1,2), e_1(2,3), e_2(3,4), e_2(4,1), e_1(1,3), e_1(3,5), e_2(4,5)$$
$$(g_2) = e_1(1,2), e_1(2,3), e_1(3,4)$$
$$(g_3) = e_1(1,2), e_1(2,3), e_2(3,4), e_2(4,1), e_1(2,5), e_1(5,6), e_1(6,3), e_2(1,2), e_1(2,4)$$
$$(g_4) = e_1(1,2), e_1(2,3), e_2(3,4), e_2(4,1), e_1(3,5), e_1(4,5), e_2(1,2), e_2(2,3)$$
$$(g_5) = n_1(1), n_1(2), n_2(3), n_3(4), e_1(1,2), e_1(1,3), e_1(1,4), e_1(2,5), e_1(3,6), e_1(5,6), e_1(6,4)$$

Then it is easy to see that g_1 and g_2 satisfy the constraints given by the rules. We call such graphs positive examples. The graphs g_3 and g_4 violate the first rule and g_5 violates the second rule. We call such graphs negative examples. Now given a dataset of such graphs how can we go about inferring the underlying rules? In the following we ignore the question of identifying the conclusions of rules and illustrate some of the basic ideas and steps used in our algorithm.

Consider taking one negative example, say g_3 and trying to extract information from it. If we could discover that only 4 nodes are required to "make it negative" and these nodes are 1, 2, 3, 4 then we could get a smaller graph to work with and use that as a pattern for our rule condition. In particular projecting g_3 onto the nodes 1, 2, 3, 4 gives

$$(g_{3_{min}}) = e_1(1,2), e_1(2,3), e_2(3,4), e_2(4,1), e_2(1,2), e_1(2,4)$$

To discover this "minimized" version of g_3 we can drop one node from the graph and then try to find out whether that node was irrelevant, i.e. whether the resulting pattern is still a negative

[1] Some exceptions exist. STILL (Sebag & Rouveirol, 2000) 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 (Bianchetti et al., 2002) uses bottom up search with some ad hoc heuristics to solve the challenge problems of (Giordana et al., 2003).

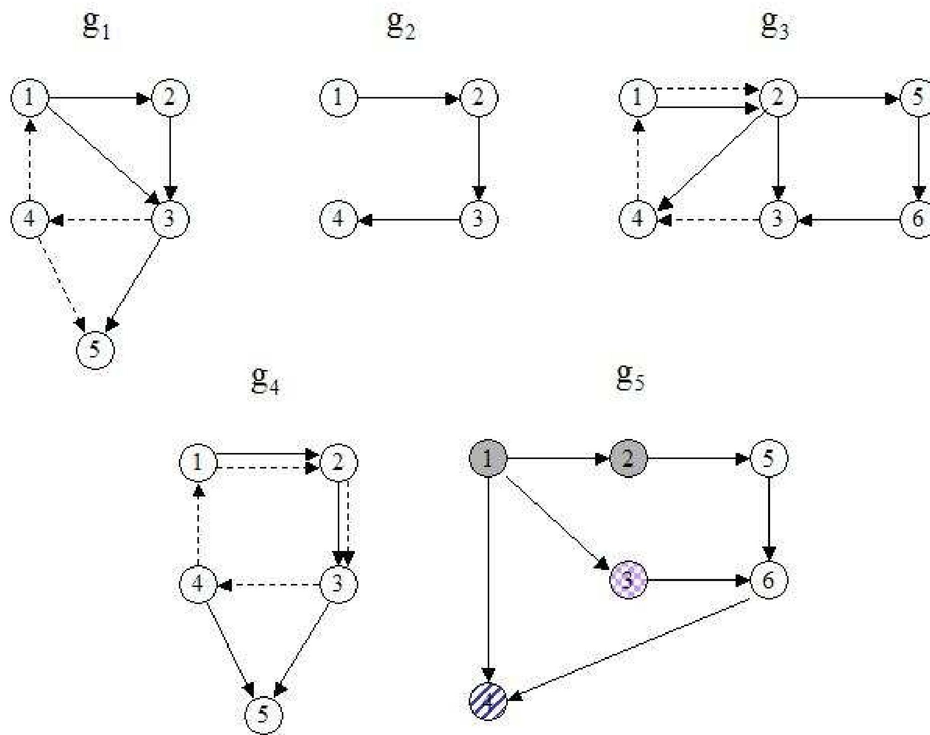


Figure 1: Representation of graphs g_1 to g_5. Solid lines represent $e_1(\cdot,\cdot)$ edges, dotted lines represent $e_2(\cdot,\cdot)$ edges. Solidly filled nodes represent $n_1(\cdot)$ nodes, checked nodes represent $n_2(\cdot)$ nodes, and nodes filled with diagonal lines represent $n_3(\cdot)$ nodes.

graph. If we are allowed to ask questions then we can do this directly. Otherwise, we use a heuristic to evaluate this question using the dataset. Iterating this procedure of dropping an object gives us a minimized negative graph. Below, we call this the minimization procedure.

Now consider taking two examples, say g_3 and g_4 and trying to extract information from them. Again the structure of examples has important implications. If we could discover that only 4 nodes are required for both examples, and that in both cases these are nodes 1, 2, 3, 4, we can focus on the shared structure in the two graphs. In this case we get

$$(g_{3,4}) = e_1(1,2), e_1(2,3), e_2(3,4), e_2(4,1), e_2(1,2)$$

so the pattern is even closer to the rule than the one in $g_{3_{min}}$. Notice that the numbering of the nodes violating the rule in the two graphs will not always coincide. Therefore, in general renaming or "aligning" of the violating nodes will be necessary and our algorithm must search for such an alignment. Again we must verify that the resulting pattern still captures one of our rules. For example if we tried this operation using g_3 and g_5 we will not find any common relevant core since they violate different rules and do not have a shared rule structure. However, if we succeed in finding a common structure then we can make substantial progress toward finding good rules. Below we call this the pairing procedure since it pairs and aligns two examples.

Our algorithm uses minimization and paring as well as some other ideas in a way that guarantees finding a consistent hypothesis. Bounds for the hypothesis size and the complexity of the algorithm are given in (Khardon, 1999a) for the case that the algorithm can ask questions and get correct answers for them. In the case answers to the questions are estimated from the data we cannot give such bounds but as our experiments show the algorithm performs well on a range of problems.

1.2 General Properties of the System

Some of the properties of LogAn-H were illustrated above. The system learns in the model of *learning from interpretations* (De Raedt & Dzeroski, 1994; De Raedt & Van Laer, 1995; Blockeel & De Raedt, 1998) where each example is an interpretation, also called a first order structure in logic terminology (Chang & Keisler, 1990). Roughly, an interpretation is the description of some scenario. Interpretations have a domain which consists of a set of objects, and a list of relations between objects in its domain that are "true" or hold in the particular scenario they describe. The system learns function free Horn rules meaning that it learns over relational data but all arguments of predicates are universally quantified variables. We do not allow for function symbols or constants in the rules.

The system solves the so-called multi-predicate learning problem, i.e., it can learn multiple rules and different rules can have different conclusions. The hypothesis of the system may include recursive clauses where the same predicate appears both in the condition and in the conclusion of a rule (obviously with different arguments). In the previous example, R_1 is a recursive rule but R_2 is not.

In contrast with most systems, LogAn-H learns all the rules simultaneously rather than learning one rule at a time (cf. Bratko, 1999). The system uses bottom up generalization in this process. In terms of rule refinement, the system can be seen as performing large refinement steps. That is, a rule structure is changed in large chunks rather than one proposition at a time. Again this is in contrast with many approaches that try to take minimal refinements of rules in the process of learning.

As mentioned above the system can run in two modes: *interactive* where the algorithm asks questions in the process of learning, and *batch* where standard supervised learning is performed.

The learning algorithms in both modes are based on the algorithm of (Khardon, 1999a) where it was shown that function-free Horn expressions are learnable from equivalence and membership queries. The interactive mode algorithm essentially uses the algorithm from (Khardon, 1999a) but adds several features to make the system more efficient. The batch mode algorithm uses the idea of simulation and can be thought of as trying to answer the queries of the interactive algorithm by appealing to the dataset it is given as input.

Efficiency considerations are important in any ILP system and in particular in bottom up learning. The system includes several techniques that speed up run time. Perhaps the most important is the use of fast implementations for subsumption, a NP-Complete matching problem that needs to be solved many times in our algorithm. The paper introduces two new methods for this problem, as well as the Django algorithm (Maloberti & Sebag, 2004) which is based on constraint satisfaction techniques. One of the new methods modifies Django to use a different representation of the subsumption problem and requires slightly different constraint satisfaction algorithms. The other method manipulates partial substitutions as tables and iteratively applies joins to the tables to find a solution.

The system incorporates further useful facilities. These include a module for pruning rules, a standard technique that may improve accuracy of learned rules. Another module is used for discretizing real valued arguments. This increases the applicability since many datasets have some numerical aspects. Both of these use known ideas but they raise interesting issues that have not been discussed in the literature, mainly since they are relevant for bottom up learners but not for top down learners.

1.3 Experimental Results

We have experimented with both modes of the system and challenging learning problems. The paper reports on qualitative experiments illustrating the performance of the system on list manipulation procedures (15-clause program including standard procedures) and a toy grammar learning problem. The grammar problem introduces the use of background information to qualify grammar rules that may be of independent interest for other settings.

We also describe quantitative experiments with several ILP benchmarks. In particular, the performance of the system is demonstrated and compared to other systems in three domains: the Bongard domain (De Raedt & Van Laer, 1995), the KRK-illegal domain (Quinlan, 1990), and the Mutagenesis domain (Srinivasan et al., 1994). The results show that our system is competitive with previous approaches, both in terms of run time and classification accuracy, while applying a completely different algorithmic approach. This suggests that bottom up approaches can indeed be used in large applications.

These experiments also demonstrate that the different subsumption algorithms are effective on a range of problems but no one dominates the others with LogAn-H. Further experiments evaluating the subsumption methods on their own show that our modified Django method is a promising approach when hard subsumption problems need to be solved. The results and directions for future work are further discussed in the concluding section of the paper.

2 The Learning Algorithms

2.1 The Model: Learning from Interpretations

Learning from interpretations has seen growing interest in recent years (De Raedt & Dzeroski, 1994; De Raedt & Van Laer, 1995; Blockeel & De Raedt, 1998). Unlike the standard ILP setting,

where examples are atoms, examples in this framework are interpretations of the underlying first order language (i.e. first order structures). We introduce the setup informally through examples. Formal definitions can be found in Khardon (1999a). 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 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. The *batch algorithm* performs the standard supervised learning task: given a set of positive and negative examples it produces a Horn expression as its output.

The interactive mode captures Angluin's (1988) model of learning from equivalence queries (EQ) and membership queries (MQ) where we assume that a *target expression* – the true expression classifying the examples – exists and denote it by T. An EQ asks about the correctness of the current hypothesis. With an EQ the learner presents a hypothesis H (a Horn expression) and, in case H is not correct, receives a counter-example, positive for T and negative for H or vice versa. With a MQ the learner presents an example (an interpretation) and is told in reply whether it is positive or negative for T. The learner asks such queries until H is equivalent to T and the answer to EQ is "yes".

2.2 Basic Operations

We first describe the main procedures used by the learning algorithm.

Candidate clauses: For an interpretation I, *rel-cands*(I) represents a set of potential clauses all sharing the same condition and violated by I. This operation is performed in two steps. The first step results in a set of clauses $[s, c]$ such that s (the condition) is the conjunction of all atoms true in I and c (the conclusions) is the set of all atoms (over the domain of I) which are false in I.[2] This resulting clause set is such that all arguments to the predicates are domain elements of I. For example when applied to the example $e_3 = ([1, 2], [p(1, 2), p(2, 2), q(2, 1)])$ this gives

$$[s, c] = [[p(1, 2), p(2, 2), q(2, 1)], [p(1, 1), p(2, 1), q(1, 1), q(1, 2), q(2, 2)]].$$

While domain elements are not constants in the language and we do not allow constants in the rules we slightly abuse normal terminology and call this intermediate form a *ground clause set*. We then replace each domain element with a distinct variable to get

$$rel\text{-}cands(I) = variabilize([s, c]) = [s', c']$$

[2] The algorithm in Khardon (1999a) allows for empty consequents as well. The system can support this indirectly by explicitly representing a predicate *false* with arity 0 (which is false in all interpretations).

where *variabilize*() replaces every domain elements with a distinct variable. For example, the clause set *rel-cands*(e_3) includes among others the clauses $[p(x_1, x_2) \wedge p(x_2, x_2) \wedge q(x_2, x_1) \rightarrow p(x_2, x_1)]$, and $[p(x_1, x_2) \wedge p(x_2, x_2) \wedge q(x_2, x_1) \rightarrow q(x_1, x_1)]$, where all variables are universally quantified.

Note that there is a 1-1 correspondence between a ground clause set $[s, c]$ and its variabilized version. In the following we often use $[s, c]$ with the implicit understanding that the appropriate version is used.

Dropping objects: This operation can be applied to an interpretation or a clause set. When dropping an object (domain element) from an interpretation we remove the element from the domain and all atoms referring to it from the extensions of predicates. Thus if we remove object 2 from $e_1 = ([1, 2, 3], [p(1, 2), p(2, 3), p(3, 1), q(1, 3)])$ we get $e_1' = ([1, 3], [p(3, 1), q(1, 3)])$. When removing an object from a clause set $[s, c]$ we remove all atoms referring to it from s and c. For example when dropping object 1 from $[s, c] = [[p(1, 2), p(2, 2), q(2, 1)], [p(1, 1), p(2, 1), q(1, 1), q(1, 2), q(2, 2)]]$ we get $[s', c'] = [[p(2, 2)], [q(2, 2)]]$

Minimization for Interactive Algorithm: Given a negative example I the algorithm iterates over domain elements. In each iteration it drops a domain element and asks a MQ to get the label of the resulting interpretation. If the label is negative the algorithm continues with the smaller example; otherwise it retains the previous example. The minimization ensures that the example is still negative and the domain of the example is not unnecessarily large. We refer to this as *minimize-objects*(I).

Removing Wrong Conclusions: Consider a clause set $[s, c]$ in the hypothesis, say as initially generated by *rel-cands*, and consider a conclusion $p \in c$ that is wrong for s. We can identify such a conclusion if we see a positive example I such that $I \not\models [s \rightarrow p]$. Notice that since I is positive it does not violate any correct rule and since it does violate $[s \rightarrow p]$ this rule must be wrong. In such a case we can remove p from c to get a better clause set $[s, c \setminus p]$. In this way we can eventually remove all wrong conclusions and retain only correct ones.

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 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.

Note that since a pairing uses a 1-1 mapping of objects we can use domain element names from either of the interpretations in the pairing. While this does not change the meaning of the clause sets this fact is used in the system to improve efficiency. This point is discussed further below.

The reasoning behind the definition of the conclusion part in the pairing operation is as follows. Consider a clause set $[s, c \setminus p]$ where a wrong conclusion p has already been removed. Now when we pair this clause set with another one the condition will be a subset of s and surely p will still be wrong. So we do not want to reintroduce such conclusions after pairing. The atoms that are removed from s while pairing have not yet been tested as conclusions for s and they are therefore added as potentially correct conclusions in the result.

2.3 The Interactive Algorithm

The algorithm is summarized in Table 1 where T denotes the target expression. Intuitively, the algorithm generates clauses from examples by using *rel-cands*(). It then uses dropping of domain elements and pairing in order to get rid of irrelevant parts of these clauses.

The algorithm maintains a sequence S of ground clause sets and the hypothesis is generated via the *variabilize*($\cdot$) operation. Once the hypothesis H is formed from S the algorithm asks an equivalence question: is H the correct expression? This is the main iteration structure which is repeated until the answer "yes" is obtained. On a positive counter-example (I is positive but $I \not\models H$), wrong clauses (s.t. $I \not\models C$) are removed from H as explained above.

On a negative counter-example (I is negative but $I \models H$), the algorithm first minimizes the number of objects in the counter-example using $I' = \textit{minimize-objects}(I)$. This is followed by generating a clause set $[s, c] = \textit{rel-cands}(I)$.

The algorithm then tries to find a "useful" pairing of $[s, c]$ with one of the clause sets $[s_i, c_i]$ in S. A useful pairing $[s_p, c_p]$ is such that s_p is a *negative* example also satisfying that s_p is *smaller* than s_i, where size is measured by the number of atoms in the set. The search is done by trying all possible matchings of objects in the corresponding clause sets and asking membership queries. The clause sets in S are tested in increasing order and the *first* $[s_i, c_i]$ for which this happens is replaced with the resulting pairing. The size constraint guarantees that measurable progress is made with each replacement. In case no such pairing is found for any of the $[s_i, c_i]$, the minimized clause set $[s, c]$ is added to S as the *last* element. Note that the order of elements in S is used in choosing the first $[s_i, c_i]$ to be replaced, and in adding the counter-example as the last element. These are crucial for the correctness of the algorithm.

2.4 The Batch Algorithm

The batch algorithm is based on the observation[3] that we can answer the interactive algorithm's questions using a given set of examples E.

Simulating equivalence queries is easy and is in fact well known. Given hypothesis H we evaluate H on all examples in E. If it misclassifies any example we have found a counter-example. Otherwise we found a consistent hypothesis. This procedure has statistical justification based in PAC learning theory (Angluin, 1988; Blumer et al., 1987). Essentially if we use a fresh sample for every query then with high probability a consistent hypothesis is good. If we use a single sample then by Occam's razor any short hypothesis is good.

For membership queries we use the following fact:

[3]Similar observations were made by Kautz et al. (1995) and Dechter and Pearl (1992) with respect to the propositional algorithm of Angluin et al. (1992).

Table 1: The Interactive Algorithm

1. Initialize S to be the empty sequence.
2. Repeat until $H \equiv T$:
   (a) Let $H = variabilize(S)$.
   (b) Ask an equivalence query to get a counter-example I in case $H \not\equiv T$.
   (c) On a positive counter-example I (s.t. $I \models T$):
       Remove wrong clauses (s.t. $I \not\models C$) from H.
   (d) On a negative counter-example I (s.t. $I \not\models T$):
      i. $I' = minimize\text{-}objects(I)$.
      ii. $[s, c] = rel\text{-}cands(I')$.
      iii. For $i = 1$ to m (where $S = ([s_1, c_1], \ldots, [s_m, c_m])$)
         For every pairing $[J, (c_i \cap c) \cup (s_i \setminus J)]$ of $[s_i, c_i]$ and $[s, c]$
         If J's size is smaller than s_i's size
            and $J \not\models T$ (ask membership query) then
         A. Replace $[s_i, c_i]$ with $[J, (c_i \cap c) \cup (s_i \setminus J)]$.
         B. Quit loop (Go to Step 2a)
      iv. If no $[s_i, c_i]$ was replaced then add $[s, c]$ as the last element of S.

Lemma 2.1 (Khardon, 1999a) *Let T be a function free Horn expression and I an interpretation over the same alphabet. Then $I \not\models T$ if and only if for some $C \in rel\text{-}cands(I)$, $T \models C$.*

Now, we can simulate the test $T \models C$ by evaluating C on all positive examples in E. If we find a positive example e such that $e \not\models C$ then $T \not\models C$. Otherwise we assume that $T \models C$ and hence that $I \not\models T$.

A straightforward application will use the lemma directly whenever the algorithm asks a membership query (this is in fact algorithm *A4* of Khardon, 1999a). However, a more careful look reveals that queries will be repeated many times. Moreover, with large data sets, it is useful to reduce the number of passes over the data. We therefore optimize the procedure as described below.

The one-pass procedure: Given a clause set $[s, c]$ the procedure *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 positive 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 issue affects the implementation and we discuss it further below.

Table 2: The Batch Algorithm

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-pass*(*rel-cands*(I)).
      ii. $[s, c] =$ *minimize-objects*($[s, c]$).
      iii. For $i = 1$ to m (where $S = ([s_1, c_1], \ldots, [s_m, c_m])$)
         For every pairing $[J, (c_i \cap c) \cup (s_i \setminus J)]$ of $[s_i, c_i]$ and $[s, c]$
         If J's size is smaller than s_i's size then
            let $[s', c'] =$ *one-pass*($[J, (c_i \cap c) \cup (s_i \setminus J)]$).
            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 $[s_i, c_i]$ was replaced then add $[s, c]$ as the last element of S.

Minimization: The minimization procedure acting on a clause set $[s, c]$ assumes the input clause set has already been validated by *one-pass*. It then iteratively tries to drop domain elements. 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 $[s, c]$. The final result of this process is $[s, c]$ in which all consequents are correct w.r.t. E.

The above simulation in *one-pass* avoids repeated queries that result from direct use of the lemma as well as guaranteeing that no positive counter-examples are ever found since they are used before clauses are put into the hypothesis. This simplifies the description of the algorithm which is summarized in Table 2.

2.5 Practical Considerations

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

Another feature used is the ability to restrict the set of predicates allowed in consequents. In this way we can restrict the system to learn a single predicate, or multiple predicates, allow or avoid recursion and so on. In addition this is a simple way to introduce knowledge about the domain into the learning task since it helps us avoid trying to evaluate irrelevant rules.

2.6 Discussion

The batch algorithm simulates the interactive one by drawing counter-examples from E. A subtle aspect concerning complexity is raised by this procedure. Assume the target expression T exists and that it uses at most k variables in each clause. It is shown in (Khardon, 1999a) that in such a case the minimization procedure outputs an interpretation with at most k domain elements. As

a result any clause C produced by the interactive algorithm has at most k variables which in turn guarantees that we can test whether $I \models C$ in time $O(n^k)$ where I has n domain elements. However, for the batch algorithm we must simulate membership queries for the minimization process itself. When doing this we generate clauses with as many variables as there are domain elements in I, which may lead to the complexity of *one-pass* growing with n^n if examples typically have n domain elements. Moreover, as mentioned above we must enumerate all substitutions between a rule and positive examples in order to test whether we can remove conclusions. This is a serious consideration that might slow down the batch system in practice. It is therefore crucial for our system to have efficient procedures to test subsumption and enumerate matching. Several such methods are described below.

The same point also suggests that for some datasets the batch algorithm may overfit the data. In particular if we do not have sufficient positive examples to contradict wrong conclusions then we may drop the wrong objects in the process of minimization. As a result the rules generated may have a low correlation with the label. Since our algorithm depends on the order of examples, one way to reduce this effect is to sort the set of examples according to size. This is useful since smaller negative examples make less demands on the richness of the data set with respect to the *one-pass* procedure. If small examples contain seeds for all rules then this sidesteps the problem. In addition, sorting the examples by size helps reduce run time since the rules generated from the small examples have less variables, a property that generally implies faster subsumption tests. In the experiments described below we have sorted examples in this manner.

Finally, it is easy to see that the batch algorithm works correctly if the data set is "complete" in the sense that every sub-structure of a negative example in the dataset is also included in the dataset. Since all of the membership queries asked by our algorithm are substructures of negative examples this guarantees that we can get the labels for the queries directly from the dataset. This holds for *one-pass* as well. In fact, a weaker condition is also sufficient. It suffices that all substructures of negative examples that are positive for the target have a "representative" in the data where I' is a representative of I if it is isomorphic to I. In this case the *one-pass* procedure is always correct for all conclusions. As a result the bounds and correctness proof given in Khardon (1999a) apply. If this does not hold then the algorithm will still find a consistent hypothesis if one exists but the hypothesis may be large. While this notion of completeness is a strong condition to require, it can can serve as a rough guide for data preparation and evaluating whether the algorithm is likely to work well for a given application. In some of the experiments below using artificial domains we generated the data in a way that is likely to include substructures of examples in other examples and indeed this led to good performance in these experiments.

3 Improving System Efficiency

3.1 Caching

The algorithms described above 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. For both algorithms the system caches interpretations rather than clauses or clause sets (i.e only the s part of $[s, c]$). In fact, for the batch algorithm we only need to cache positive interpretations — if a clause set $[s, \emptyset]$ was returned by one pass then s does not imply any of the possible consequents

and therefore it is a positive interpretation. 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.

3.2 Live Pairings

The system also includes an optimization that reduces the number of pairings that are tested without compromising correctness. Recall that the algorithm has to test all injective mappings between the domains of two interpretations. We say that a mapping is *live* if every paired 2-object appears in the extension of at least one atom in the condition of the pairing. One can show that if the target expression is range restricted (i.e. all variables in the consequent appear in the antecedent) then testing live mappings is sufficient. For example, consider pairing the examples $[s_a, c_a] = [[p(1,2), p(2,3)], [q(2,2), q(3,1)]]$ and $[s_b, c_b] = [[p(a,b), p(b,c), p(a,d)], [q(d,d), q(c,a)]]$. Then the mapping $\{1/a, 2/b, 3/c\}$ gives the pairing $[[p(1,2), p(2,3)], [q(3,1)]]$ which is live. On the other hand the mapping $\{1/a, 2/d, 3/c\}$ gives the pairing $[[p(1,2)], [p(2,3), q(2,2), q(3,1)]]$ and when forcing range restricted form we get $[[p(1,2)], [q(2,2)]]$. This pairing is not live since $3/c$ does not appear in it. So, this pairing can be ignored. Technically due to range restricted form it is enough to check this only on the condition part. For efficiency, instead of generating all injective matchings and filtering irrelevant ones, one can first collect a set of potential matched objects-pairs and generate matchings from these. The system includes this optimization as an option and it was used in some of the experiments below.

3.3 Efficient Subsumption Tests

The *one-pass* procedure must enumerate all substitutions that embed a clause in an example. This problem is NP-Hard and hence we cannot expect a solution that is always efficient. The system includes several different procedures to achieve this that may be useful in different contexts.

3.3.1 Table Based Subsumption

While backtracking search (as done in Prolog) can find all substitutions without substantial space overhead, the time overhead can be very 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 given in Table 3 (from left to right):

Table 3: Example of Table Based Subsumption

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.

The code incorporates some heuristics to speed up computation. For example we check that each variable has at least one substitution common to all its instances in the clause. Otherwise we know that there is no substitution matching the clause to the examples. We also sort predicates in an example by their table size so that we first perform joins of small tables. We have also experimented with a module that performs lookahead to pick a join that produces the smallest table in the next step. This can substantially reduce the memory requirements and thus run time but on the other hand introduces overhead for the lookahead. Finally, taking inspiration from Django (see description below) we have also implemented a form of arc-consistency for the tables. In this case we first compute joins of table pairs and project them back onto the original variables. This removes rows that are clearly inconsistent from the tables. This is repeated until all tables are pairwise consistent and then the tables method is started as before.

One can easily construct examples where the table in intermediate steps is larger than the memory capacity of the computer, even if the final table is small. In this case the matching procedure will fail. This indeed occurs in practice and we have observed such large table sizes in the mutagenesis domain (Srinivasan et al., 1994) as well as the artificial challenge problems of (Giordana et al., 2003).

3.3.2 Randomized Table Based Subsumption

If lookahead 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$\times$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.

3.3.3 Subsumption Based on Constraint Satisfaction Algorithms

The idea of using constraint satisfaction algorithms to solve subsumption problems has been investigated in (Maloberti & Sebag, 2004), where a very effective system Django is developed. The Django system was originally designed to find a single solution for the subsumption problem but this can be easily extended to give all solutions through backtracking. In the following we describe

the ideas behind the original system that we refer to as DjangoDual as well as introduce a new method we call DjangoPrimal that uses similar techniques but a different representation.
A Constraint Satisfaction Problem (CSP) (Tsang, 1993) is defined by a triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ such that:

$\mathcal{V}$ is a set of variables;

$\mathcal{D}$ is a set of domains, each domain D_i associates a set of admissible values to a variable V_i. The function $dom(V_i)$ returns the domain associated to the variable V_i.

$\mathcal{C}$ is a set of constraints, each constraint C_i involves a tuple $\langle S_i, R_i \rangle$ such that :

S_i is a subset of $\mathcal{V}$, denoted *scope* of a constraint. The function $vars(C_i)$ returns the variables in the scope of C_i, i.e., $vars(C_i) = S_i$.

R_i is a set of tuples, denoted *relations* of a constraint, each tuple is a set of simultaneously admissible values for each variable in S_i. The function $rel(C_i)$ returns the relations associated to the constraint C_i.

A constraint C_i is satisfied by an assignment of values to variables if and only if the tuple $\langle v_1, v_2, \ldots, v_k \rangle$ is in R_i where for all j variable $V_j \in S_i$ and $v_j \in dom(V_j)$ is assigned to V_j. A CSP is *satisfiable* iff it admits a solution, assigning to each variable V_i in $\mathcal{V}$ a value v_i in $dom(V_i)$ such that all constraints are satisfied.

In general the relation R_i can be implemented in 3 different ways: as a set of tuples of simultaneously admissible values; as a set of tuples of simultaneously inadmissible values; or as a boolean function which returns *true* if the values assigned to each variable in S_i satisfy the constraint.

To illustrate how constraint satisfaction can be used consider the following subsumption problem where the clause is $\mathcal{C}$ and the example is Ex:

$$\begin{array}{rl} \mathcal{C}: & p(X_0, X_1), q(X_0, X_2, X_3), r(X_0) \\ Ex: & p(a_0, a_1), p(a_1, a_2), q(a_0, a_2, a_3), q(a_0, a_1, a_3), r(a_0) \end{array}$$

This problem can be trivially transformed into a n-ary CSP problem such that:

- The set of variables $\mathcal{V}$ is equivalent to the set of variables of $\mathcal{C}$; $\{X_0, X_1, X_2, X_3\}$;
- Each domain D_i is equivalent to the set of objects in Ex, $D_i = \{a_0, a_1, a_2, a_3\}$;
- Each literal l_i in $\mathcal{C}$ is transformed into a constraint C_i such that:
  - the scope S_i of C_i corresponds to the set of variables of l_i;
  - the set of relation R_i of C_i are built with the set L of literals of Ex with the same predicate symbol as l_i, each tuple of R_i corresponds to the set of variables of one literal of L.

Therefore, there are three constraints C_1, C_2 and C_3 in our example, which respectively account for the literals $r(X_0)$, $p(X_0, X_1)$ and $q(X_0, X_2, X_3)$, such that:

$S_1 = \{X_0\}$ and $R_1 = \{\langle a_0 \rangle\}$

$S_2 = \{X_0, X_1\}$ and $R_2 = \{\langle a_0, a1 \rangle, \langle a_1, a_2 \rangle\}$

$S_3 = \{X_0, X_2, X_3\}$ and $R_3 = \{\langle a_0, a_2, a_3 \rangle, \langle a_0, a_1, a_3 \rangle\}$.

Thus, finding an assignment for all variables of this CSP which satisfy all the constraints, is equivalent to finding a substitution θ such that $\mathcal{C}\theta \subseteq Ex$.

In order to distinguish this representation from others, this CSP is called the *Primal representation.*

Since a n-ary CSP can always be transformed into a binary CSP, i.e. a CSP such that all constraints involve at most 2 variables, most algorithms in CSP are restricted to binary CSPs. Thus, in order to simplify the implementation of DjangoDual, a transformation named *dualization* was used. In order to distinguish variables from the primal CSP to the variables of the dual representation which have a different semantic, dual variables are denoted Δ-variables. Dualization encapsulates each constraint of the primal CSP in a Δ-variable, and a constraint is added in the dual CSP between each pair of Δ-variables which share at least one variable.

Therefore, the problem of the previous example is transformed into a dual CSP, where:

- Each literal l in $\mathcal{C}$ gives rise to a *dual variable* Δ_l. Therefore, $\mathcal{V} = \{\Delta_r, \Delta_p, \Delta_q\}$. If more than one literal have the same predicate symbol p, Δ-variables are noted $\{\Delta_{p.1}, \Delta_{p.2}, \ldots, \Delta_{p.n}\}$.

- Each domain of a Δ-variable Δ_l is equivalent to the set of literals in Ex with the same predicate symbol as l, $dom(\Delta_r) = \{r(a_0)\}$, $dom(\Delta_p) = \{p(a_0, a_1), p(a_1, a_2)\}$ and $dom(\Delta_q) = \{q(a_0, a_2, a_3), q(a_0,\ a_1,\ a_3)\}$.

- A constraint is created for each pair of literals which share at least one variable in $\mathcal{C}$, or in the primal CSP. The relation R_i contains all pairs of literals in Ex which share the same terms. Since X_0 is shared by all the literals, there are three constraints C_1, C_2 and C_3 in our example, where:

  $S_1 = \{\Delta_r, \Delta_p\}$, $R_1 = \{\langle r(a_0), p(a_0, a_1)\rangle\}$
  $S_2 = \{\Delta_r, \Delta_q\}$, $R_2 = \{\langle r(a_0), q(a_0, a_2, a_3)\rangle, \langle r(a_0), q(a_0, a_1,\ a_3)\rangle\}$
  $S_3 = \{\Delta_p, \Delta_q\}$, $R_3 = \{\langle p(a_0, a_1), q(a_0, a_2, a_3)\rangle, \langle p(a_0, a_1), q(a_0, a_1, a_3)\rangle\}$.

In the particular case of multiple occurrences of a variable in a same literal, a unary constraint is created in order to check that all literals in the domain also have at least the same multiple occurrences of a term.

Dualization performs $\frac{n\times(n-1)}{2}$ comparisons of literals, where n is the number of literals in $\mathcal{C}$. Each comparison of literals involves a^2 tests of equivalence between variables, where a is the maximal arity of the predicates. Thus, building a dual CSP from a clause is obviously a polynomial transformation. It does however incur an overhead before starting to solve the subsumption problem and can represent an important part of the execution time in case of easy instances of subsumption problems.

It has been shown in (Chen, 2000) that polynomially solvable instances of Primal CSPs may require exponential time in the Dual representation, and vice versa. The system of (Maloberti & Sebag, 2004) uses the dual representation and we therefore refer to it DjangoDual. LogAn-H includes both options and in particular we have implemented a new solver using the primal representation, which we call DjangoPrimal. Despite the different representations, both versions use a similar method to solve the subsumption problem. We describe the main ideas below.

Both methods first build the domain of the variables applying a node-consistency procedure. Node-consistency removes from variable domains all values that do not satisfy unary constraints. Unary constraints are created when a variable has multiple occurrences in a literal.

Both methods continue by checking for arc-consistency. Arc consistency means that we remove from the domains all values which are not supported. A value is consistent with respect to a

constraint, if there exists a value in the domains of all other variables of a constraint such that the constraint is satisfied. A value is supported if it is consistent for all constraints involving this variable. Obviously, a value not supported cannot be assigned to a variable, since a constraint will not be satisfied. DjangoDual uses a standard algorithm of arc-consistency named AC3 (Mackworth, 1977), which maintains a queue of all constraints that must be checked. The queue is initially full, and if a value is removed from a domain, all other constraints involving the variable are added to the queue. The algorithm checks and removes all elements in the queue until it is empty. AC3 is not optimal since it can check a same constraint more than once, however it has good average execution times. DjangoPrimal uses a version of AC3 adapted to n-ary CSPs, named CN (Mackworth, 1977) or GAC3, for General Arc Consistency. Finally both methods perform a depth-first search using lookahead strategy; a partial solution is incrementally build assigning a value to one variable at a time. The variable is selected using Dynamic Variable Ordering (DVO), for each variable the size of its domain is divided by the number of constraints involving it, the variable with the minimal ratio is selected. This heuristic follows the First Fail Principle that tries to reduce the search tree by causing failure earlier during the search. All values inconsistent with this partial solution are removed from the domains of the variables not yet assigned. Three different strategies of propagation the effects of the last assignment are used:

Forward Checking (FC) only checks variables connected to the last assigned variable.

Maintaining Arc Consistency (MAC), also called Full Arc Consistency, extends FC by checking arc-consistency on adjacent variables if a value has been removed from their domain.

Forward Checking Singleton extends FC by also checking all constraints involving a domain containing a single value. Checking such constraint is cheaper than full arc consistency and can reach to a failure sooner than simple FC.

DjangoPrimal uses FC, while DjangoDual uses a more sophisticated strategy, named Meta-Django, which selects FC or MAC according to a measure, denoted κ (Gent et al., 1996), estimating the position of the problem relatively to the Phase Transition (Giordana & Saitta, 2000). Instances in the YES region use FC, while instances in the PT and the NO regions use MAC.

In addition to the techniques above DjangoDual uses the idea of *signatures* to further prune the search. A signature captures the idea that when we map a literal in a clause onto a literal in the example all the neighborhood of the first literal must exist in the example as well. By encoding neighborhoods of literals which are Δ-variables in the dual representation we can prune the domain in similarity to arc-consistency. The details of this heuristic are given in (Maloberti & Sebag, 2004).

3.3.4 Discussion of Subsumption Methods

The main difference between the table based method and the Django methods is that all substitutions are calculated simultaneously in the tables method and by backtracking search in Django. This has major implications for run time. In cases where the tables are applicable and where the intermediate tables are small the tables can save a substantial amount of overhead. On the other hand the tables can get too large to handle. Even when tables are of moderate size then memory copying operations when handling joins can be costly so we are introducing another overhead. Within LogAn-H the backtracking approach can benefit when we have many substitutions of which only a few are needed to remove all consequents from a clause set.

The table-based subsumption method incorporates aspects from both the primal and dual representation of Django. The way partial substitutions are represented as tables is similar to

the dual variables and the method can be seen as reducing the number of dual variables through the join operations. The test that guarantees that each variable has at least one potential value to match is related to arc-consistency in the primal representation since it works by reducing domains of primal variables. On the other hand the arc-consistency improvement is done over the tables, that is in the dual representation.

Finally, calculating the dual form representation of the subsumption problem incurs some overhead that can be noticeable if the subsumption problems themselves are quickly solved. Since in LogAn-H we have many subsumption tests with the same example or the same clause these can be avoided with additional caching. This gives a substantial improvement for the overall run time of LogAn-H when using DjangoDual.

4 Processing Examples and Rules

4.1 Discretization of Real-Valued Arguments

The system includes a capability for handling numerical data by means of discretization. To discretize the 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 then discretization can be applied. Given a specified number of thresholds, the system then determines the threshold values, allocates the same name to all objects in each range, and adds predicates reflecting the linear 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 fewer 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. Thus while including all possible boundaries gives maximum flexibility in the hypothesis language this may not be desirable due to the danger of overfitting. This is discussed further in the experimental section.

Several approaches to discretization i.e. for choosing the number of bins and the boundaries have been proposed in the literature (Fayyad & Irani, 1993; Dougherty et al., 1995; Blockeel & De Raedt, 1997). Some of these are completely automatic in the sense that they decide on the number of partitions and the location of the partitions. Others require some user input as to the granularity and decide on the location of the partitions. One approach to automate methods that require user input is to perform an "internal" cross validation in each fold and use it to select the

parameter normally given by the user.

We have experimented with several approaches as follows. The *equal frequency* approach requires the number of bins as input and it assigns the boundaries by giving each bin the same number of occurrences of values. The *proportional* approach similarly requires a bin rate r as input and it assigns boundaries every r distinct values in the data. The rate used is the same for all logical groups requiring discretization and this reduces the cost of searching for good parameters. Both approaches do not use the labels of the examples. As a result they can also be used in “transductive” mode where test data is used as well to generate the boundaries. The *Information Gain* (IG) approach introduced by (Fayyad & Irani, 1993) uses the information gain criterion to split the range of values using a decision tree (Quinlan, 1993) and stops splitting using a minimum description length measure. Finally, we also used the method we call *All* where we put a threshold at every class boundary. The last two methods use the labels of the training examples in constructing the boundaries and they were demonstrated in (Fayyad & Irani, 1993; Dougherty et al., 1995) to provide good performance in attribute value settings. For relational data the IG method must be refined since more than one value of the same type may appear in the same example. This was already addressed by (Blockeel & De Raedt, 1998) where values were weighted according to the number of times they appear in the examples. However, (Blockeel & De Raedt, 1998) set the number of bins by hand since IG provided too few regions.

4.2 Pruning Rules

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 (Mitchell, 1997) using a validation dataset. For each atom in the rule 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.

4.3 Using Examples from Normal ILP setting

In the normal ILP setting (Muggleton & DeRaedt, 1994) 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 De Raedt, 1997; Khardon, 1999a). 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 (Muggleton, 1995) 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 $[s, c] = \textit{rel-cands}(I)$ from it (notice that b is among the conclusions 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. As a result subsumption procedures that enumerate solutions one by one can quit early, after the first substitution, and are likely to be faster in this case. In addition, note that the pairing operation never moves new atoms into the consequent and is therefore a pure generalization operation.

5 Experimental Evaluation

The ideas described above have been implemented in two different systems. The first one, initially reported in (Khardon, 2000), implements the interactive and batch algorithms in the Prolog language but does not include discretization, pruning or special subsumption engines. The second implementation, recently reported in (Arias & Khardon, 2004), was done using the C language and implements only the batch algorithm but otherwise includes all the techniques described above.

We first describe two experiments that illustrate the behavior of the system on list manipulation programs and a toy grammar learning problem. The intention is to exemplify the scope and type of applications that may be appropriate. We then describe a set of quantitative experiments in three benchmark ILP domains: the Bongard domain (De Raedt & Van Laer, 1995), the KRK-illegal domain (Quinlan, 1990), and the Mutagenesis domain (Srinivasan et al., 1994). The experiments illustrate applicability in terms of scalability and accuracy of the learned hypotheses as well as providing a comparison with other systems. They also demonstrate that different subsumption engines may lead to faster execution in different problems.

We also compare the subsumption methods in an experiment based on the Phase Transition

phenomenon similar the experiments performed in (Maloberti & Sebag, 2004). These experiments show that DjangoPrimal is very effective and may perform better than the other two approaches if hard subsumption problems around the phase transition region need to be solved.

5.1 Learning List Manipulation Programs

To facilitate experiments using the interactive mode, we have implemented an "automatic-user mode" where (another part of) the system is told the expression to be learned (i.e. the target T). The algorithm's questions are answered automatically using T. Since implication for function-free Horn expressions is decidable this can be done reliably. In particular, Lemma 13 in Khardon (1999a) provides an algorithm that can test implication and construct counter-examples to equivalence queries. Membership queries can be evaluated on T. We note that this setup is generous to our system since counter-examples produced by the implemented "automatic user mode" are in some sense the smallest possible counter-examples.

Using this mode we ran the interactive algorithm to learn a 15-clause program including a collection of standard list manipulation procedures. The program includes: 2 clauses defining $list(L)$, 2 clauses defining $member(I, L)$, 2 clauses defining $append(L1, L2, L3)$, 2 clauses defining $reverse(L1, L2)$, 3 clauses defining $delete(L1, I, L2)$, 3 clauses defining $replace(L1, I1, I2, L2)$, and 1 clause defining $insert(L1, I, L2)$ (via $delete()$ and $cons()$). The definitions have been appropriately flattened so as to use function free expressions. All these clauses for all predicates are learned simultaneously. This formalization is not range-restricted so the option reducing the number of pairings was not used here. The Prolog system required 35 equivalence queries, 455 membership queries and about 8 minutes (running Sicstus Prolog on a Linux platform using a Pentium 2/366MHz processor) to recover the set of clauses exactly. This result is interesting since it shows that one can learn a complex program defining multiple predicates in interactive mode with a moderate number of questions. However, the number of questions is probably too large for a program development setting where a human will answer the questions. It would be interesting to explore the scope for such use in a real setting.

5.2 A Toy Grammar Learning Problem

Consider the problem of learning a grammar for English from examples of parsed sentences. In principle, this can be done by learning a Prolog parsing program. In order to test this idea we generated examples for the following grammar, which is a small modification of ones described by Pereira and Shieber (1987).

```
s(H2,P0,P)  :- np(H1,P0,P1),vp(H2,P1,P),number(H1,X),number(H2,X).
np(H,P0,P)  :- det(HD,P0,P1),n(H,P1, P),number(HD,X),number(H,X).
np(H,P0,P)  :- det(HD,P0,P1),n(H,P1, P2),rel(H2,P2,P),number(H,X),
               number(H2,X),number(HD,X).
np(H,P0,P)  :- pn(H,P0,P).
vp(H,P0,P)  :- tv(H,P0,P1),np(_,P1,P).
vp(H,P0,P)  :- iv(H,P0,P).
rel(H,P0,P) :- relw(_,P0,P1),vp(H,P1,P,L1).
```

Note that the program corresponds to a chart parser where we identify a "location parameter" at beginning and end of a sentence as well as between every two words, and true atoms correspond to edges in the chart parse. Atoms also carry "head" information for each phrase and this is used to decide on number agreement. The grammar allows for recursive sub-phrases. A positive example for

this program is a sentence together with its chart parse, i.e. all atoms that are true for this sentence. The base relations identifying properties of words, i.e., $det(), n(), pn(), iv(), tv(), relw(), number()$ are assumed to be readable from a database or simply listed in the examples.

We note that the grammar learning problem as formalized here is interesting only if external properties such as $number()$ are used. Otherwise, one can read-off the grammar rules from the structure of the parse tree. It is precisely because such information is important for the grammar but normally not supplied in parsed corpora that this setup may be useful. Of course, it is not always known which properties are the ones crucial for correctness of the rules as this implies that the grammar is fully specified. In order to model this aspect in our toy problem we included all the relations above and in addition a spurious property of words $confprop()$ (for "confuse property") whose values were selected arbitrarily. For example a chart parse of the sentence "sara writes a program that runs" is represented using the positive example:

```
([sara,writes,a,program,that,runs,alpha,beta,singular,0,1,2,3,4,5,6],
 [s(writes,0,4), s(writes,0,6),
  np(sara,0,1), np(program,2,4), np(program,2,6),
  vp(writes,1,4), vp(runs,5,6), vp(writes,1,6),
  rel(runs,4,6), pn(sara,0,1), n(program,3,4), iv(runs,5,6),
  tv(writes,1,2), det(a,2,3), relw(that,4,5),
  number(runs,singular), number(program,singular), number(a,singular),
  number(writes,singular), number(sara,singular),
  confprop(runs,beta), confprop(program,alpha),
  confprop(writes,beta),confprop(sara,alpha)
 ]).
```

Note also that one can generate negative examples from the above by removing implied atoms of $s(), np(), vp(), rel()$ from the interpretation. It may be worth emphasizing here that negative examples are not non-grammatical sentences but rather partially parsed strings. Similarly, a non-grammatical sequence of words can contribute a positive example if all parse information for it is included. For example "joe joe" can contribute the positive example

```
([joe,0,1,2,alpha,singular],
 [pn(joe,0,1), pn(joe,1,2), np(joe,0,1), np(joe,1,2),
  number(joe,singular),confprop(joe,alpha)
 ]).
```

or a negative one such as

```
([joe,0,1,2,alpha,singular],
 [pn(joe,0,1), pn(joe,1,2), np(joe,0,1),
  number(joe,singular),confprop(joe,alpha)
 ]).
```

A simple analysis of the learning algorithm and problem setup shows that we must use non-grammatical sentences as well as grammatical ones and we have done this in the experiments. For example, if all examples have agreement in number, the algorithm has no way of finding out whether the $number()$ atoms in an example can be dropped or not since nothing would contradict dropping them.

A final issue to consider is that for the batch algorithm to work correctly we need to include positive sub-structures in the data set. While it is not possible to take all sub-structures we approximated this by taking all *continuous* substrings. Given a sentence with k words, we generated from it all $O(k^2)$ substrings, and from each we generated positive and negative examples as described above.

We ran two experiments with this setup using the restriction to live pairings. In the first we hand picked 11 grammatical sentences and 8 non-grammatical ones that "exercise" all rules in the grammar. With the arrangement above this produced 133 positive examples and 351 negative examples. The batch algorithm recovered an exact copy of the grammar, making 12 equivalence queries and 88 calls to *one-pass*.

In the second experiment we produced all grammatical sentences with "limited depth" restricting arguments of $tv()$ to be $pn()$ and allowing only $iv()$ in relative clauses. This was done simply in order to restrict the number of sentences, resulting in 120 sentences and 386 sub-sentences. We generated 614 additional random strings to get 1000 base strings. These together generated 1000 positive examples and 2397 negative examples. With this setup the batch algorithm found a hypothesis consistent with all the examples, using 12 equivalence queries and 81 calls to *one-pass*. The hypothesis included all correct grammar rules plus 2 wrong rules. This is interesting as it shows that although some examples are covered by wrong rules other examples reintroduced seeds for the correct rules and then succeeded in recovering the rules.

The experiments demonstrate that it is possible to apply our algorithms to problems of this type even though the setup and source of examples is not clear from the outset. They also show that it may be possible to apply our system (or other ILP systems) to some NLP problems but that data preparation will be an important issue in such an application.

5.3 Bongard Problems

The Bongard domain is an artificial domain that was introduced with the ICL system (De Raedt & Van Laer, 1995) to test systems that learn from interpretations. 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 our 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 function free representation the domain size in examples is larger than the number of objects (to include: *up*, *down*, *black*, *white*). As in the previous experiment, this mode of data generation has some similarity to the "closure" property of datasets that guarantees good performance with our algorithm.

In order to label examples we arbitrarily picked 4 target Horn expressions of various complexities. Table 4 gives an indication of target complexities. The first 3 targets have 2 clauses each but vary in the number of atoms in the antecedent and the fourth one has 10 clauses of the larger kind making the problem more challenging. The numbers of atoms and variables are meant as a rough indication as they vary slightly between clauses. To illustrate the complexity of the targets, one of the clauses in target IV is

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

We ran the batch algorithm on several sample sizes. Table 5 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. Each entry is an average of 10 independent runs where a fresh set of random examples is used in each run. The last column in the table gives the majority class percentage.

Table 4: Complexity of Targets

Target	*Clauses*	*Atoms*	*Variables*
I	2	4	2
II	2	6	4
III	2	9	6
IV	10	9	6

Table 5: Performance Summary

System	Target	200	500	1000	2000	3000	Majority Class
LogAn-H	I	99.7	100	100	100	100	64.4
LogAn-H	II	97.6	99.4	99.9	99.9	100	77.8
LogAn-H	III	90.8	97.2	99.3	99.9	99.9	90.2
LogAn-H	IV	85.9	92.8	96.8	98.4	98.9	84.7
ICL	IV	85.2	88.6	89.1	90.2	90.9	84.7

Clearly, the algorithm is performing very well on this problem setup. Note that the baseline is quite high, but even in terms of relative error the performance is good. We also see a reasonable learning curve obtained for the more challenging problems. Notice that for target I the task is not too hard since it is not unlikely that we get a random example matching the antecedent of a rule exactly (so that discovering the clause is easy) but for the larger targets this is not the case. We have also run experiments with up to 10 shapes per example with similar performance.

To put these experiments in perspective we applied the systems ICL (De Raedt & Van Laer, 1995) and Tilde (Blockeel & De Raedt, 1998) to the same data. Exploratory experiments suggested that ICL performs better on these targets so we focus on ICL. We ran ICL to learn a Horn CNF and otherwise with default parameters. ICL uses a scheme of declarative bias to restrict its search space. With a general pattern implying little bias, success was limited. We thus used a bias allowing up to 4 shapes and identifying the relation between $config$ and *up*, *down* and similarly *color* and *black*, *white*. Interestingly, for ICL the change in performance from target I to IV was less drastic than in LogAn-H. This may well be due to the fact that LogAn-H builds antecedents directly from examples. The last line in Table 5 gives the performance of ICL on target IV. As can be seen our system performs better than ICL on this problem.

We ran the Prolog implementation (using compiled code in Sicstus Prolog) and the new C implementation on the same hardware and observed a speedup of over 400-fold when using the tables method or DjangoDual and a speedup of 320 when using DjangoPrimal. Recall that the number of objects in the examples is relatively small for this experiment. For larger examples as in the experiments that follow the improvement is even more dramatic as the Prolog code cannot complete a run and the C code is still pretty fast.

5.4 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 (Muggleton et al., 1989; Quinlan, 1990). 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

Table 6: 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 (Quinlan, 1990)				92.50			99.40		

(`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 position atom.

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),
⋮
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)]-
```

The top part of Table 6 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 (Quinlan, 1990). Third, relevant background knowledge seems to make the task easier. Fourth, pruning considerably improves performance on this dataset especially for small training sets.

Table 7: Runtime comparison for subsumption tests on KRK-illegal dataset

Subsumption Engine	*runtime in s.*	*accuracy*	*actual table size*
DjangoDual	40.27	98.10%	n/a
DjangoPrimal	78.74	98.10%	n/a
Tables	136.43	98.10%	130928
Tables ARC Consistency	92.47	98.10%	109760
Lookahead	191.50	98.10%	33530
No cache	503.92	98.10%	130928
Rand. TH=1	3804.52	33.61%	16
Rand. TH=10	178.69	33.61%	160
Rand. TH=100	58.41	72.04%	1600
Rand. TH=1000	126.48	98.10%	16000

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 discretization. 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. Since in this domain the number of bins and the boundaries for discretization are obvious we used the equal frequency method with the appropriate number of bins. The main goal of this experiment was to evaluate the effect of discretization on the performance. As can be seen in Table 6 good accuracy is 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 so iterative subsumption tests are likely to be faster. The comparison is given for the non-discretized "all background mode" with 1000 training examples. Table 7 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 DjangoDual is faster than DjangoPrimal in this domain and both are faster than the tables method. Adding arc-consistency to the tables method improves both space requirements and run time. 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 7). Caching gives a significant reduction 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 larger tables sizes (TH=1000) the randomized method does very well and reproduces the deterministic results.

5.5 Mutagenesis

The Mutagenesis dataset is a structure-activity prediction task for molecules introduced in (Srinivasan et al., 1994). 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. Srinivasan et al., 1994; Sebag & Rouveirol, 2000; De Raedt & Van Laer, 1995). For the sake of comparison we 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 numerical chemical properties and some elementary chemical concepts such as aromatic rings, nitro groups, etc. Concretely, we use all the information corresponding to the background level B3 of (Srinivasan et al., 1995). Notice that the original data is given in the normal ILP setting and hence we transformed it according to Section 4.3 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`; hence discretization is needed. We have experimented with all the methods explained in Section 4.1. For the equal frequency method and the proportional method we report results of double cross validation performing automatic parameter selection. For reference we also report the best result that can be obtained in hindsight by searching for good partitions using the test set. The equal frequency method uses values only in the training set. The proportional method was used in the "transductive" mode where the tests examples (but not their labels) are used as well. For the equal frequency method we searched over the bin settings as follows: for `atom-charge` we used $\{5, 15, 25, 35, 45\}$ for `lumo` we used $\{4, 6, 8, 10, 20, 30, 80\}$ and for `logp` we used $\{4, 6, 8, 10, 30, 50\}$. For the proportional method we used bin rates in $\{1, 2, 3, 5, 10, 20, 50\}$. For the IG method and All method the number of bins is selected automatically so a single value is reported.

Table 8 summarizes some of the accuracies obtained by the various discretization methods. As can be seen the overoptimistic "best" results give pretty high accuracy. The automatic selection methods perform quite well with the equal frequency method giving 83.62%, and the proportional method yielding 83.96%. The IG method comes close with 82.25%. Given the major increase in run time for the double cross validation, especially when multiple variables are being discretized, the proportional method or IG may provide a good tradeoff to take in larger applications.

Our result compares well to other ILP systems: PROGOL (Srinivasan et al., 1994) reports a total accuracy of 83% with B3 and 88% with B4; STILL (Sebag & Rouveirol, 2000) reports results in the range 85%–88% on B3 depending on the values of various tuning parameters, ICL (De Raedt & Van Laer, 1995) reports an accuracy of 84% and finally (Laer et al., 1996) report that FOIL (Quinlan, 1990) achieves an accuracy of 83%.

We have also run experiments comparing run time with the different subsumption engines. For this domain deterministic table-based subsumption was not possible, not even with lookahead and arc-consistency since the table size grew beyond memory capacity of our computer. However, both modes of Django are very efficient on this domain. For these runs we used the equal frequency discretization method with `atom-charge`$= 25$, `lumo`$= 8$ and `logp`$= 30$ that gave top-scoring accuracy as described above. Table 9 gives average run time (on Linux running with a 2.80 GHz Xeon processor) per fold as well as the average accuracy obtained. One can observe that DjangoPrimal is faster than the DjangoDual and that even with small parameters the randomized methods do very well. An inspection of the hypothesis to the deterministic runs shows

Table 8: Comparison of accuracies with various discretization methods.

Method	Avg. accuracy
Equal Frequency best split `atom-charge`=25 `lumo`=8 `logp`=30	89.41%
Equal Frequency automatic selection	83.62%
Proportional best rate boundary every 20 values	87.83%
Proportional automatic selection	83.96%
Information Gain	82.25%
All label boundaries	80.27%

that they are very similar.

Table 9: Runtime comparison for subsumption tests on mutagenesis dataset

Subsumption Engine	*runtime*	*accuracy*
DjangoDual	3.48 *sec.*	89.41%
DjangoPrimal	0.82 *sec.*	89.41%
Rand. TH=1 R=1	0.74 *sec.*	88.88%
Rand. TH=10 R=1	0.93 *sec.*	90.46%
Rand. TH=100 R=1	3.01 *sec.*	90.46%
Rand. TH=1 R=10	0.91 *sec.*	90.46%
Rand. TH=1 R=100	2.92 *sec.*	89.93%
Rand. TH=10 R=100	9219.68 *sec.*	89.93%

5.6 Subsumption and Phase Transition

The previous experiments have demonstrated that different subsumption engines may lead to faster performance in different domains. In this section we further compare the different algorithms but purely on subsumption problems, that is, not in the context of learning. Previous work (Giordana & Saitta, 2000) has shown that one can parameterize subsumption problems so that there is a sharp transition between regions where most problems have a solution to regions where most problems do not have a solution. This is known as the phase transition phenomenon, and it has been used to evaluate subsumption and learning algorithms (Giordana & Saitta, 2000; Giordana et al., 2003; Maloberti & Sebag, 2004).

For these experiments, a set of clauses and a set of examples are generated using four parameters:

n The number of variables in each clause, which is set to 12;

m The number of literals in each clause, which varies in $[12, 50]$;

N The number of literals built on each predicate symbol in each example, i.e. the size of each domain in dual representation, which is set to 50;

L The number of constants in each example, i.e. the size of each domain in primal representation, which varies in $[12, 50]$.

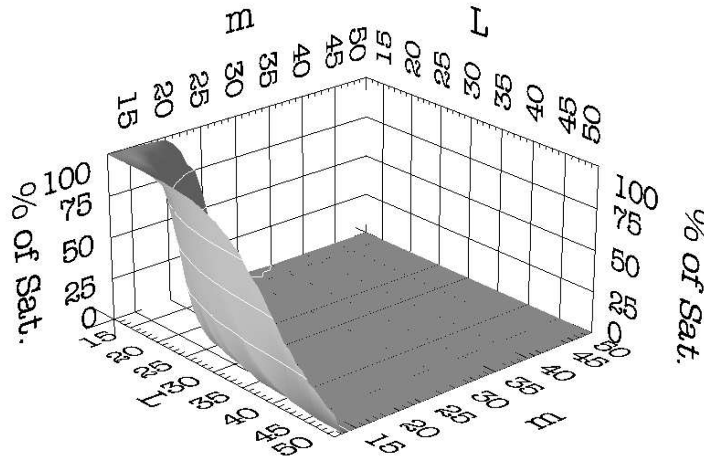


Figure 2: Percentage of subsumption tests satisfied on 50×50 pairs $(\mathcal{C}, Ex)$, where $\mathcal{C}$ is a clause uniformly generated with $n = 12$ variables and m literals (m in [12,50]), and Ex is an example uniformly generated with $N = 50$ literals built on each one of the m predicate symbols in $\mathcal{C}$, and L constants (L in [12,50]).

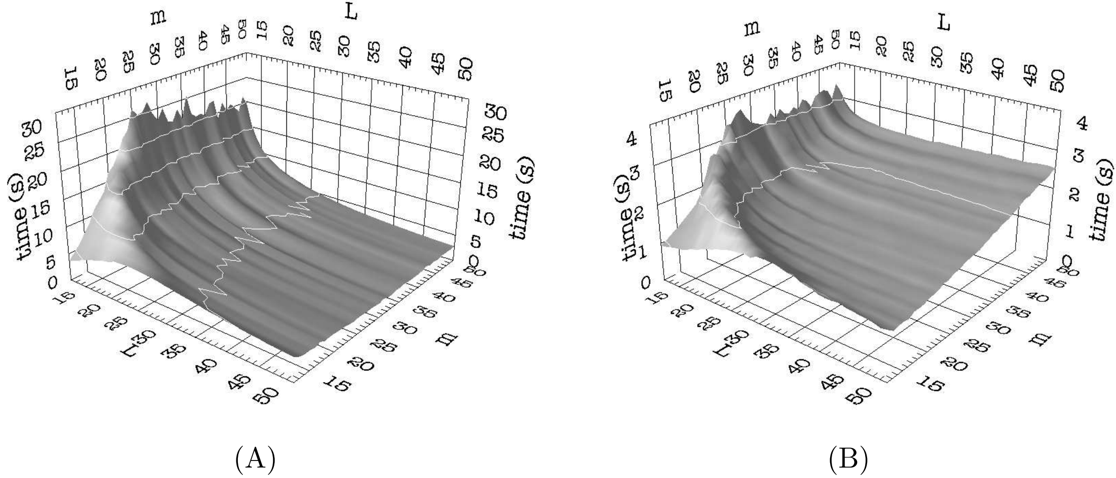
All m literals in a generated clause $\mathcal{C}$ are built on distinct binary predicate symbols and clause $\mathcal{C}$ is connected, i.e. all n variables are linked. The latter requirement prevents the subsumption problem from being decomposable into simpler problems.

Each generated example Ex is a conjunction of ground literals. Each literal in Ex is built on a predicate symbol occurring in $\mathcal{C}$ (other literals are irrelevant to the subsumption problem). The number of literals in Ex per predicate symbol, L, is constant, thus all domains of the literals of $\mathcal{C}$ have the same size, and each example contains $L \times m$ literals.

For each pair of values of $\langle m, L \rangle$, 50 clauses and 50 examples are generated, each clause is tested against all examples. Run times are measured on an Athlon 900 MHz with 512MB of memory.

As shown in Figure 2, the probability for $\mathcal{C}$ to subsume Ex abruptly drops from 100% to 0% in a very narrow region. This region is called Phase Transition, and is particularly important for the subsumption problem, since computationally hard problems are located in this region. This phenomenon can be observed in Figure 3 (A) representing the average computational costs for Meta-Django which uses the dual representation. Figure 3 (B) shows the average execution times for DjangoPrimal. The Phase Transition phenomenon is also apparent, however DjangoPrimal is 7 times faster than Django on average, despite the simple versions of algorithms used in DjangoPrimal.

The experiments performed here are slightly different from the ones in (Maloberti & Sebag, 2004). First, the computational cost of the transformation to the dual representation is also measured here. This is important in the context of a system that dynamically builds clauses and tests subsumption for them. Although the cost of dualization is generally moderate, it can be significant in some experiments so it must be included in the execution time. Second, settings used in former experiments were: $N = 100$, $n = 10$, L and m in $[10, 50]$. In the dual representation, the worst case complexity is N^m, while in primal representation it is L^n. Therefore, $N = 100$ is too large



(A) (B)

Figure 3: Subsumption cost(m, L) in secods for (A) Meta-Django and (B) DjangoPrimal averaged over 50×50 pairs $(\mathcal{C}, Ex)$. A reduction in running time of a factor of 7 is observed in DjangoPrimal compared to Meta-Django (notice difference in scale of the time axis).

compared to L, and $n = 10$ is too small compared to m. It is difficult to change these settings because a slight variation can shift the phase transition to be outside of reasonable ranges of values. Thus, we reduced N to 50 and adjusted n to 12. L and m have been set to $[12, 50]$ in order to keep the property of non decomposability of $\mathcal{C}$. Finally, in (Maloberti & Sebag, 2004), 100 clauses and 100 examples were generated, each clause was tested against one example.

The table based methods are less well suited for the problems in this test suite. The deterministic table methods ran out of memory and could not be used. The randomized methods give a tradeoff between run time and accuracy but they perform worse than Django.

For example, the randomized tables method with TH= 1 and R= 1 is very fast with an average execution time between $0.37s$ and $0.54s$. However, almost no solution can be found, even in the region of the test with a very high satisfiability. Therefore, as shown in Figure 4 (A), its error rate is close the percentage of satisfiability.

As shown in Figure 5 (A), the randomized tables method with TH= 10 and R= 10 is slower than Django, and its average cost does not rely on the phase transition. On the other hand, Figure 5 (B) shows that its error rate is very high in the phase transition region, while it is now very low in the region with high satisfiability.

These experiments suggest that in general the django methods are more robust than the tables methods and that the DjangoPrimal is a promising alternative. However, these results must be interpreted with caution, since the tables methods look for all solutions while DjangoPrimal and Django only search a solution. Moreover, the settings chosen are probably better for the primal representation and other parameters, such as arity of literals, must also be investigated. In the context of LogAn-H, the success of the table methods may be explained by the fact that the algorithm starts with long clauses and makes them shorter in the process. Thus it starts in the NO region and moves toward the phase transition and then possibly the YES region. Since tables perform well in the NO region and boundary to the phase transition they can do well overall. DjangoPrimal was also faster in some of the experiments with LogAn-H. Therefore, further optimization of DjangoPrimal and adding randomized facilities to Django are natural

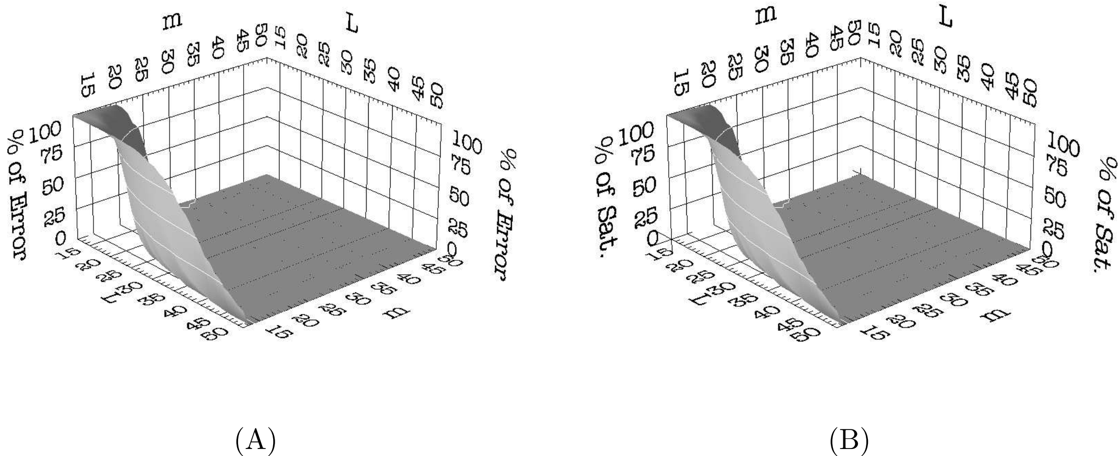


Figure 4: (A) Percentage of wrong subsumption tests for randomized tables method on 50×50 pairs $(\mathcal{C}, Ex)$, with TH= 1 and R= 1. (B) Percentage of satisfiable subsumption tests. Notice that the table based method with such low valued parameters is not able to discover any subsumption substitution, and hence the error rate corresponds to the satisfiability rate of the subsumption problem suite.

directions to improve the system.

6 Conclusion

The paper introduced the system LogAn-H implementing new algorithms for learning function free Horn expressions. The system is based on algorithms proved correct in Khardon (1999a) but includes various improvements in terms of efficiency as well as a new batch algorithm that learns from examples only. The batch algorithm can be seen as performing a refinement search over multi-clause hypotheses. The main difference from other algorithms is that our algorithm is using a bottom up search and that it is using large refinement steps in this process. We demonstrated through qualitative and quantitative experiments that the system performs well in several benchmark ILP tasks. Thus our system gives competitive performance on a range of tasks while taking a completely different algorithmic approach — a property that is attractive when exploring new problems and applications.

The paper also introduced new algorithms for solving the subsumption problem and evaluated their performance. The table based methods give competitive performance with LogAn-H and DjangoPrimal is a promising new approach where hard subsumption problems in the phase transition region are solved.

As illustrated using the Bongard domain, LogAn-H is particularly well 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 (Giordana et al., 2003) where our system outputs a large number of rules and gets low accuracy. Interestingly, a similar effect can result from discretization since it results

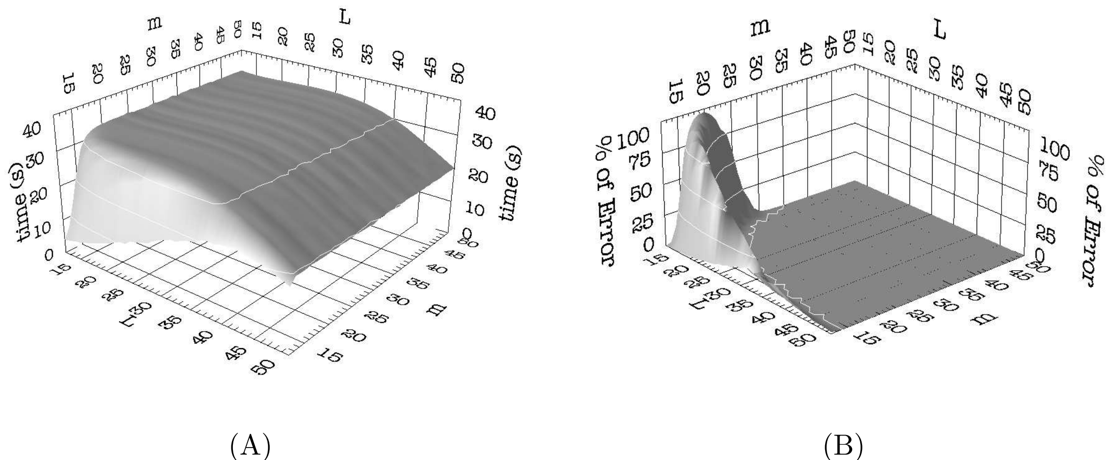


Figure 5: (A) Subsumption cost(m, L) and (B) error percentage for randomized tables method with TH= 10 and R= 10 averaged over 50×50 pairs $(\mathcal{C}, Ex)$.

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 system 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 (Bianchetti et al., 2002). 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 Giordana et al., 2003) 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.

Finally, it would be interesting to explore real world applications of the interactive algorithm. The work on the robot scientist project (King et al., 2004) provides one such possibility. This work uses a chemistry lab robot to perform experiments on essays to aid in the process of learning. The lab experiments are very similar to the membership queries used by our algorithm, however queries may be limited to chemicals that can be synthesized by the system. Thus such an application will require adapting our algorithm to use only appropriate queries. This raises another interesting challenge for future work.

Acknowledgments

This work was partly supported by EPSRC grant GR/M21409 in the UK and by NSF Grant IIS-0099446 in the USA. Stephen Kitt at the University of Edinburgh and Constantine Ashminov at

Tufts University made large contributions to the code in the system. We are grateful to Hendrik Blockeel who provided the Tilde system as well as a generator for Bongard examples, and Win Van Laer provided the ICL system. Mark Steedman made useful comments on earlier drafts. Some of the experiments were performed on a Linux cluster provided by Tufts Central Computing Services.

References

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, *2*, 319–342.

Angluin, D., Frazier, M., & Pitt, L. (1992). Learning conjunctions of Horn clauses. *Machine Learning*, *9*, 147–164.

Arias, M., & Khardon, R. (2002). Learning closed Horn expressions. *Information and Computation*, *178*, 214–240.

Arias, M., & Khardon, R. (2004). Bottom-up ILP using large refinement steps. *Proceeding of the International conference on Inductive Logic Programming*.

Arimura, H. (1997). Learning acyclic first-order Horn sentences from entailment. *Proceedings of the International Conference on Algorithmic Learning Theory* (pp. 432–445). Sendai, Japan: Springer-verlag. LNAI 1316.

Bianchetti, J. A., Rouveirol, C., & Sebag, M. (2002). Constraint-based learning of long relational concepts. *Proceedings of the International Conference on Machine Learning* (pp. 35–42). Morgan Kaufmann.

Blockeel, H., & De Raedt, L. (1997). Lookahead and discretization in ILP. *Proceedings of the 7th International Workshop on Inductive Logic Programming* (pp. 77–84). Springer-Verlag.

Blockeel, H., & De Raedt, L. (1998). Top down induction of first order logical decision trees. *Artificial Intelligence*, *101*, 285–297.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters*, *24*, 377–380.

Bratko, I. (1999). Refining complete hypotheses in ILP. *International Workshop on Inductive Logic Programming* (pp. 44–55). Bled, Slovenia: Springer. LNAI 1634.

Bratko, I., & Muggleton, S. (1995). Applications of inductive logic programming. *Communications of the ACM*, *38*, 65–70.

Chang, C., & Keisler, J. (1990). *Model theory*. Amsterdam, Holland: Elsevier.

Chen, X. (2000). *A theoretical comparison of selected csp solving and modeling techniques*. Doctoral dissertation, University of Alberta, Canada.

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

De Raedt, L., & Dzeroski, S. (1994). First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, *70*, 375–392.

De Raedt, L., & Van Laer, W. (1995). Inductive constraint logic. *Proceedings of the 6th Conference on Algorithmic Learning Theory*. Springer-Verlag.

Dechter, R., & Pearl, J. (1992). Structure identification in relational data. *Artificial Intelligence*, *58*, 237–270.

Dougherty, J., Kohavi, R., & Sahami, M. (1995). Supervised and unsupervised discretization of continuous features. *Proceedings of the International Conference on Machine Learning* (pp. 194–202).

Fayyad, U., & Irani, K. B. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. *Proceedings of the International Conference on Machine Learning* (pp. 1022–1027). Amherst, MA.

Gent, I., MacIntyre, E., Prosser, P., & Walsh, T. (1996). The constrainedness of search. *AAAI/IAAI, Vol. 1* (pp. 246–252).

Giordana, A., & Saitta, L. (2000). Phase transitions in relational learning. *Machine Learning*, *2*, 217–251.

Giordana, A., Saitta, L., Sebag, M., & Botta, M. (2003). Relational learning as search in a critical region. *Journal of Machine Learning Research*, *4*, 431–463.

Kautz, H., Kearns, M., & Selman, B. (1995). Horn approximations of empirical data. *Artificial Intelligence*, *74*, 129–145.

Khardon, R. (1999a). Learning function free Horn expressions. *Machine Learning*, *37*, 249–275.

Khardon, R. (1999b). Learning range-restricted Horn expressions. *Proceedings of the Fourth European Conference on Computational Learning Theory* (pp. 111–125). Nordkirchen, Germany: Springer-verlag. LNAI 1572.

Khardon, R. (2000). Learning horn expressions with LogAn-H. *Proceedings of the International Conference on Machine Learning* (pp. 471–478). Morgan Kaufmann.

King, R. D., Whelam, K., Jones, F., Reiser, P., Bryant, C., Muggleton, S., Kell, D., & Oliver, S. (2004). Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, *427*, 247–252.

Krishna Rao, M., & Sattar, A. (1998). Learning from entailment of logic programs with local variables. *Proceedings of the International Conference on Algorithmic Learning Theory.* Otzenhausen, Germany: Springer-verlag. LNAI 1501.

Laer, W. V., Blockeel, H., & Raedt, L. D. (1996). Inductive constraint logic and the mutagenesis problem. *Proceedings of the Eighth Dutch Conference on Artificial Intelligence* (pp. 265–276).

Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, *8*, 99–118.

Maloberti, J., & Sebag, M. (2004). Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, *55*, 137–174.

Mitchell, T. (1997). *Machine learning.* McGraw-Hill.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, *13*, 245–286.

Muggleton, S., & Buntine, W. (1992). Machine invention of first order predicates by inverting resolution. In S. Muggleton (Ed.), *Inductive logic programming.* Academic Press.

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

Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In S. Muggleton (Ed.), *Inductive logic programming*, 281–298. Academic Press.

Muggleton, S. H., Bain, M., Hayes-Michie, J., & Michie, D. (1989). An experimental comparison of human and machine learning formalisms. *Proc. Sixth International Workshop on Machine Learning* (pp. 113–118). San Mateo, CA: Morgan Kaufmann.

Pereira, F., & Shieber, S. (1987). *Prolog and natural-language analysis.* Stanford : Center for the Study of Language and Information.

Plotkin, G. D. (1970). A note on inductive generalization. *Machine Intelligence*, *5*, 153–163.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning.* Morgan Kaufmann.

Reddy, C., & Tadepalli, P. (1998). Learning first order acyclic Horn programs from entailment. *International Conference on Inductive Logic Programming* (pp. 23–37). Madison, WI: Springer. LNAI 1446.

Sebag, M., & Rouveirol, C. (2000). Resource-bounded relational reasoning: Induction and deduction through stochastic matching. *Machine Learning*, *38*, 41–62.

Srinivasan, A., Muggleton, S., & King, R. (1995). Comparing the use of background knowledge by inductive logic programming systems. *Proceedings of the 5th International Workshop on Inductive Logic Programming* (pp. 199–230).

Srinivasan, A., Muggleton, S. H., King, R. D., & Sternberg, M. J. E. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. *Proc. 4th Int. Workshop on Inductive Logic Programming* (pp. 217–232). Bad Honnef / Bonn.

Tsang, E. (1993). *Foundations of constraint satisfaction*. Academic Press, London.