Journal of Machine Learning Research 8 (2007) 549-587 Submitted 12/05; Revised 8/06; Published 3/07

Learning Horn Expressions with LOGAN-H

Marta Arias MARTA@CS.COLUMBIA.EDU
Center for Computational Learning Systems
Columbia University
New York, NY 10115, USA

Roni Khardon RONI@CS.TUFTS.EDU
Department of Computer Science
Tufts University
Medford, MA 02155, USA

Jérôme Maloberti MALOBERT@LRI.FR
Laboratoire de Recherche en Informatique
Université Paris-Sud
F-91405, Orsay, France

Editor: Stefan Wrobel

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 that 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 fast 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.

Keywords: inductive logic programming, subsumption, bottom-up learning, learning with queries

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 and DeRaedt, 1994; Bratko and 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: for example, the GOLEM system (Muggleton and Feng, 1992) used relative least general generalization (Plotkin, 1970, 1971) 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 (Quinlan, 1990; Muggleton, 1995; De Raedt and Van Laer, 1995; Blockeel and 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 and Tadepalli, 1998; Krishna Rao and Sattar, 1998; Khardon, 1999b,a; Arias and 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 where both nodes and edges may have labels. 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 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 of nodes appropriately. Consider a class of graphs that is characterized by the following rules:

$$\begin{aligned}
R_1 &= \forall x_1, x_2, x_3, x_4, \\
&\quad e_1(x_1,x_2), e_1(x_2,x_3), e_2(x_3,x_4), e_2(x_4,x_1) \rightarrow e_1(x_1,x_3), \\
R_2 &= \forall x_1, x_2, x_3, x_4, \\
&\quad n_1(x_1), n_1(x_2), n_2(x_3), n_3(x_4), e_1(x_1,x_2), e_1(x_1,x_3), e_1(x_1,x_4) \rightarrow e_2(x_2,x_3).
\end{aligned}$$

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

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

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 data set 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.

1. Some exceptions exist. STILL (Sebag and 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).

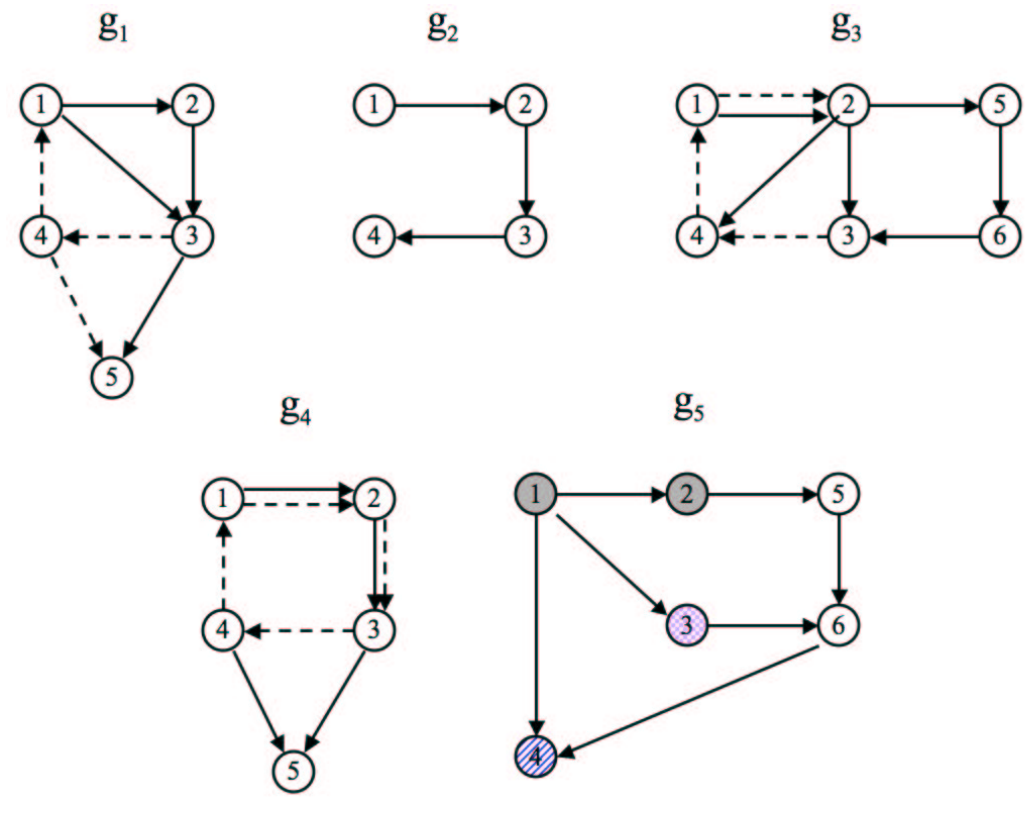


Figure 1: Representation of graphs g_1 to g_5. Solid lines represent $e_1(\cdot,\cdot)$ edges, dotted lines represent $e_2(\cdot,\cdot)$ edges. Solidly filled nodes represent $n_1(\cdot)$ nodes, checked nodes represent $n_2(\cdot)$ nodes, and nodes filled with diagonal lines represent $n_3(\cdot)$ nodes.

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, that is, whether the resulting pattern is still a negative 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 data set. 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 pairing 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 by Khardon (1999b) 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 can provide similar bounds under somewhat strong assumptions on the data set. 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 and Dzeroski, 1994; De Raedt and Van Laer, 1995; Blockeel and De Raedt, 1998) where each example is an interpretation, also called a first order structure in logic terminology (Chang and 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, that is, 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 of the rule and in the conclusion (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 (1999b) where it was shown that function-free Horn expressions are learnable from equivalence and membership queries. The interactive mode algorithm essentially uses this algorithm 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 data set it is given as input.

Efficiency considerations are important in any ILP system and in particular in bottom up learning, since these systems tend to start with larger and more complex clauses thus requiring more resources in the beginning. The system includes several techniques that speed up run time. Perhaps the most important is the use of fast implementations for subsumption, an NP-Complete matching problem that needs to be solved many times in our algorithm. The paper introduces two new methods for this problem, in addition to using the DJANGO algorithm (Maloberti and 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 data sets 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 and 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.

The 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.

1.4 Summary and Organization

The main contribution of the paper is introducing the LOGAN-H system with its learning algorithms and implementation heuristics. The system is a bottom up ILP system using ideas from learning theory in its algorithms. The paper demonstrates through various experiments that the system and its algorithms provide an interesting alternative to top down learning which is the common approach in ILP. The paper also makes a contribution to the study of efficient subsumption algorithms. We develop some new subsumption algorithms and evaluate them both in the context of machine learning and independently. One of the new methods, called DJANGOPRIMAL below, seems particularly promising when hard subsumption problems need to be solved.

The paper is organized as follows. Section 2 formalizes the learning setting and problem. Section 3 describes our learning algorithm and its different variants. Section 4 describes three subsumption engines that our system uses. Section 5 describes extensive experiments that demonstrate the validity of our method and of the subsumption procedures. Finally, in the appendix the reader can find multiple details on several implementation improvements and heuristics used to optimize the system.

2. Notation and Problem Settings

We start with some basic notation from logic and logic programming as used in the paper. For a general introduction to these topics see Chang and Keisler (1990) and Lloyd (1987). We assume that the learner is given its "vocabulary" in advance. Formally this is a fixed *signature*, a set of predicates each with its associated arity. The expressions we consider do not allow any constants or other function symbols so only variables can be used as arguments to predicates. An atom is a predicate with an appropriate list of arguments. A Horn clause (sometimes called a Horn rule or just a rule) is an expression $C = (\wedge_{n \in \mathrm{N}} n) \rightarrow P$, where N is a set of atoms and P is an atom. In the examples that follow we assume a signature with 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. In this paper all variables in the clause are universally quantified and we often omit the quantifiers in expressions. The conjunction on the left of the implication is called the condition (also antecedent) of the rule and P is called the conclusion (also consequent). Clauses may have empty consequents. A universally quantified function-free Horn expression is a conjunction of Horn clauses. The goal of our learning algorithms is to learn such expressions.

An expression is given a truth value relative to an interpretation of the symbols in the signature. An *interpretation* lists a domain of elements and the truth values of predicates over them. For example, the interpretation

$$e_1 = ([1,2,3],[p(1,2),p(2,3),p(3,1),q(1,3)])$$

has domain $[1,2,3]$; by convention, the four atoms listed are true in the interpretation and other atoms are false. The *size* of an interpretation is the number of atoms true in it, so that $size(e_1) = 4$. The interpretation e_1 *falsifies* (we sometimes also say *violates*) the clause above. To see this, substitute $\{1/x_1, 2/x_2, 3/x_3\}$). On the other hand

$$e_2 = ([a,b,c,d],[p(a,b),p(b,c),p(a,c),p(a,d),q(a,c)])$$

satisfies the clause. We use standard notation $e_1 \not\models c_1$ and $e_2 \models c_1$ for these facts.

Throughout the paper we use the following notation. The *clause set* $[s,c]$ (where $c \neq \emptyset$) represents the conjunction of clauses $\bigwedge_{b \in c}(s \rightarrow b)$. If c is empty then no clause is represented and $[s,\emptyset]$ is in this sense "illegal". As mentioned above, Horn clauses in general do allow for "empty consequents". For example, a disjunction $(\overline{A} \vee \overline{B})$ is often described as $(A \wedge B \rightarrow \Box)$. Our system can handle such consequents but we do so by explicitly representing a predicate `false` with arity 0 which is false in all interpretations. Thus, the clause above will be represented in our system as $[[A,B],[\mathtt{false}]]$.

2.1 The Model: Learning from Interpretations

Learning from interpretations has seen growing interest in recent years (De Raedt and Dzeroski, 1994; De Raedt and Van Laer, 1995; Blockeel and De Raedt, 1998; Blockeel et al., 1999; Stolle et al., 2005). Unlike the standard ILP setting, where examples are atoms, examples in this framework are interpretations. As mentioned above we consider two settings for the learner.

The *batch learning algorithm* performs the standard supervised learning task: given a set of positive and negative examples it produces a Horn expression as its output. In this paper we measure algorithm performance by testing the hypothesis on an unseen test set, but we also mention theoretical guarantees for the algorithms implied by previous work.

The *interactive learning algorithm* requires an interface capturing Angluin's (1988) model of learning from equivalence queries (EQ) and membership queries (MQ). In this model 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".

3. The Learning Algorithms

We first describe the main procedures used by the learning algorithms and then describe the interactive and batch algorithms.

3.1 Basic Operations

Candidate clauses: For an interpretation I, the "relational candidates" (Khardon, 1999b) $\textit{rel-cands}(I)$ is the set of potential clauses all sharing the same antecedent and violated by I. The set $\textit{rel-cands}(I)$ is calculated in two steps. The first step results in a set of clauses $[s,c]$ such that s (the antecedent) 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. This 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

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

where $\textit{variabilize}()$ replaces every domain elements with a distinct variable. For example, the clause set $\textit{rel-cands}(e_3)$ 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 one to one 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 the clause

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

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).

Clearly, the order by which objects are considered for removing can affect the resulting hypothesis. The theoretical guarantees for the interactive algorithm discussed below hold regardless of the order but the performance of the batch algorithm can be affected dramatically. The current system does not try to optimize this and simply drops objects in the order they are encountered in example descriptions.

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 $[s_p,c_p]$. It is closely related to the LGG operation of Plotkin (1970) but it avoids the exponential growth in size of the resulting clause after several applications of the operation. When pairing, we use 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 $\{1/a,2/b,3/c\}$.
- $J = [p(1,2),p(2,3),q(1,3)]$.
- $[s_p,c_p] = [[p(1,2),p(2,3),q(1,3)],[q(3,1),p(3,1)]$.

Note that the pairing operation is not symmetric. 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) and it thus subsumes both s_a and s_b. 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 an injective 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 consequent 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 antecedent 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.

3.2 The Interactive Algorithm

The interactive algorithm is basically the algorithm *A2* from Khardon (1999b). 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.

Note that the algorithm enumerates matchings and corresponding pairings of clauses. As in the case of minimization the order of pairings considered can make a difference to the resulting hypothesis but the current system does not attempt to optimize this and uses some arbitrary order that is easy to calculate.

The interactive algorithm was previously analyzed and it has the following guarantees:

```
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 $S$.
```

Table 1: The Interactive Algorithm.

Theorem 1 (see Corollary 2 in Khardon (1999b)) *Assume there is a target expression T with m clauses each having at most k variables and that equivalence and membership queries are answered correctly according to T. Let p be the number of predicates in the signature, a the maximum arity and n the maximum number of objects in any negative counter-example provided to equivalence queries in the process of learning. Then the interactive algorithm will stop and produce a hypothesis equivalent to T with $O(mpk^ak^k)$ clauses after $O(mpk^ak^k)$ equivalence queries and $O((n+m^2)pk^ak^{3k})$ membership queries.*

3.3 The Batch Algorithm

The batch algorithm is based on the observation[2] 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:

2. 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).

Lemma 2 (See Lemma 11 and 12 in Khardon 1999b) *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-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, 1999b). 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 we know that no atom is implied by s, that is, there are no correct consequents. Therefore the evaluation process is stopped and the procedure returns $[s,\emptyset]$ to indicate that this is the case.[3] At the end of *one-pass*, each consequent is correct w.r.t. the data set.

This operation is at the heart of the algorithm since the hypothesis and candidate clause sets are repeatedly evaluated against the data set. 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.

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 $[s',c']$ and 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.

3.4 Discussion of Batch Algorithm

The analysis of the interactive algorithm can be used to give some guarantees for the batch algorithm as well. In particular it is clear that all counter-examples used are correct since they are in the data set. If we can guarantee in addition that all membership queries implicit in *one-pass* are answered correctly then the batch algorithm can be seen as performing some run of the interactive algorithm and bounds on queries and hypothesis size translate as well.

3. Recall from above that the "empty consequent" is explicitly represented by `false` and if this is a correct consequent it will not be removed.

```
1. Initialize $S$ to be the empty sequence.

2. Repeat until $H$ is correct on all examples in $E$.

   (a) Let $H = variabilize(S)$.
   (b) If $H$ misclassifies $I$ ($I$ is negative but $I \models H$):
       i. $[s,c] = one\text{-}pass(rel\text{-}cands(I))$.
       ii. $[s,c] = minimize\text{-}objects([s,c])$.
       iii. For $i = 1$ to $m$ (where $S = ([s_1,c_1],\ldots,[s_m,c_m])$)
              For every pairing $[J,(c_i \cap c) \cup (s_i \setminus J)]$ of $[s_i,c_i]$ and $[s,c]$
              If $J$'s size is smaller than $s_i$'s size then
                 let $[s',c'] = one\text{-}pass([J,(c_i \cap c) \cup (s_i \setminus J)])$.
                 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 $S$.
```

Table 2: The Batch Algorithm.

Recall that all of the membership queries asked by the interactive algorithm are sub-structures of negative examples. Consider a "complete" data set in the sense that every sub-structure of a negative example in the data set is also included in the data set. In this case the answer to a membership query on interpretation J exists explicitly in the data set. Similarly, the call to *one-pass* returns at least one conclusion (c' is not empty) if and only if a correct conclusion exists. In fact, a slightly weaker condition is also sufficient. We say that a data set is *relational complete for target T* if sub-structures of negative examples that are positive for T have a "representative" in the data where I' is a representative of I if it is isomorphic to I. Clearly such a data set provides exactly the same answer as a corresponding complete data set. We therefore have the following result:

Theorem 3 *Consider any data set which is relational complete for target T where T has m clauses each having at most k variables. Let p be the number of predicates in the signature, a the maximum arity and n the maximum number of objects in any negative example in the data set. Then the batch algorithm will stop and produce a hypothesis consistent with the data with $O(mpk^a k^k)$ clauses.*

Notice that the bound on hypothesis size does not depend on the number of examples but only on parameters of the target. If the data set is not complete 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 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 sub-structures of examples in other examples and indeed this led to good performance in these experiments.

The use of a data set to simulate queries raises a subtle aspect concerning time complexity. Assume the target expression T exists and that it uses at most k variables in each clause. It is shown in Khardon (1999b) 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 data sets 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.

At this point it is interesting to compare the batch algorithm to the one used by the GOLEM system (Muggleton and Feng, 1992). The appendix discusses the relation between the so-called "normal ILP setting" used in GOLEM and the setting of learning from interpretations used by LOGAN-H. It suffices here to say that some manipulation allows us to work on the same problems. GOLEM uses bottom up learning by calculating the relative least general generalization (RLGG) of clauses. In our setting this corresponds to taking cross products of interpretations instead of pairings. This is incorporated in a greedy search that iteratively generalizes one of the clauses in its hypothesis by combining a new example with the clause. Thus the general structure of the algorithms is pretty similar. The main differences are that (1) we use pairings in order to get small clauses in the hypothesis whereas GOLEM uses the RLGG, that (2) pairing is aimed at learning multiple conclusions simultaneously thus moving atoms from antecedent to consequent, that (3) LOGAN-H uses the minimization procedure, and that (4) GOLEM greedily chooses the clause with maximum cover for combining clauses whereas LOGAN-H's choice is based on ordering the hypothesis clauses as suggested by the theoretical analysis.

3.5 Other Practical Considerations

The batch algorithm as described above includes all the main ideas but it may be hard to apply in some cases. The appendix gives details of heuristics implemented in the system that extend its applicability. In particular the system can handle noisy data (where no consistent hypothesis exists), handle numerical attributes through discretization, and use rule pruning. In addition the appendix describes several techniques that considerably improve run time. Aside from these, the most crucial run time issue is the subsumption test which we discuss next.

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

Table 3: Example of Table Based Subsumption.

4. Efficient Subsumption Tests

The *one-pass* procedure must enumerate all substitutions that embed a clause in an example.[4] This problem is NP-Hard and hence we cannot expect a solution that is always efficient. Here we describe three different procedures that are shown to perform well in different settings; our system can work with any of these. This may be useful in other contexts as well.

4.1 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. This approach is similar to the matching procedure of Di Mauro et al. (2003) and was developed independently; we discuss the similarities and differences in more detail below. To illustrate the method 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 joins from left to right we get partial substitution tables given in Table 3 (from left to right). 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

4. It may be worth clarifying here that this is the standard subsumption problem and we do not require different variables to be matched to different objects. The one to one restriction appears in the analysis of Khardon (1999b) but is not part of the algorithm or its hypothesis.

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).

4.2 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. In this way we are not performing a completely random choice over possible substitutions. Instead we are informing the choice by our intermediate table. The absolute limit to the size of intermediate tables is TH$\times$16. If a join would produce a larger table we arbitrarily trim the table when it reaches this capacity, and then follow with the random selection. 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.

4.3 Subsumption Based on Constraint Satisfaction Algorithms

The idea of using constraint satisfaction algorithms to solve subsumption problems has been investigated in Maloberti and 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_V associates a set of admissible values to a variable $V \in \mathcal{V}$. The function $dom(V)$ returns the domain associated to the variable V.

$\mathcal{C}$ is a set of constraints, each constraint $C \in \mathcal{C}$ involves a tuple $\langle S, R \rangle$ such that :

- S is a subset of $\mathcal{V}$, denoted *scope* of a constraint. The function $vars(C)$ returns the variables in the scope of C, that is, $vars(C) = S$.
- R is a set of tuples, denoted *relations* of a constraint, each tuple is a set of simultaneously admissible values for each variable in S. The function $rel(C)$ returns the relations associated to the constraint C.

A constraint C is satisfied by an assignment of values to variables if and only if the corresponding tuple of assignments to $vars(C)$ is in $rel(C)$. A CSP is *satisfiable* iff it admits a solution, assigning to each variable $V \in \mathcal{V}$ a value $v \in dom(V)$ such that all constraints are satisfied. The arity of a CSP is the maximum number of variables in any constraint.

To illustrate how constraint satisfaction can be used consider the following subsumption problem where the clause is Cl and the example is Ex:

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

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

- The set of variables $\mathcal{V}$ is equivalent to the set of variables of Cl; $\{X_0,X_1, X_2, X_3\}$;
- All domains are the same and are equivalent to the set of objects in Ex, $D = \{a_0,a_1,a_2, a_3\}$;
- Each literal l_i in Cl 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 relation R_i of C_i includes all variable tuples from instances of l_i's predicate in Ex.

  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 an *n*-ary CSP can always be transformed into a binary CSP, that is, 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. To avoid confusion we call variables in the dual representation Δ-variables. Dualization encapsulates each constraint of the primal CSP into a Δ-variable, and a constraint is added in the dual CSP between each pair of Δ-variables which shares 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 has the same predicate symbol p, Δ-variables are denoted $\{\Delta_{p.1},\Delta_{p.2},\ldots,\Delta_{p.n}\}$.
- 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 $\mathcal{C}$. The relation R_i contains all pairs of literals in Ex in which the corresponding primal variables are given the same values (so they are compatible). 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 the same literal, a unary constraint is created in order to check that all literals in the domain also have the same term in corresponding positions.

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 (Chen, 2000) that polynomially solvable instances of Primal CSPs may require exponential time in the Dual representation, and vice versa. The system of Maloberti and Sebag (2004) uses the dual representation and we therefore refer to it as 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, using arc-consistency and following with depth first search with dynamic variable ordering. However, due to the fact that the primal representation has non binary constraints, simpler heuristics are used in DJANGOPRIMAL. Some implementation details and the differences between the two methods are described in the appendix.

4.4 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 table 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, 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.

Di Mauro et al. (2003) have previously introduced a table based method for subsumption tests (although our system was developed independently). The main difference between the two approaches is that their method tries to optimize space by compressing table entries. Essentially, their tables try to compress multiple substitutions into a single line. This can be done whenever variables have independent constraints. Therefore, one may expect their method to do better in terms of space when tables are large, but possibly at the cost of run time to compress and decompress the substitutions. In contrast, our randomized method attempts to save both time and space but this is at the cost of completeness.

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.

Other systems have developed different methods to reduce the computational cost of subsumption tests. Blockeel et al. (2000) and Blockeel et al. (2002) propose the use of *query packs*. Query packs make the evaluation of a set of hypotheses against a clause more efficient. A query-pack structures a set of hypotheses in a tree where each node is a set of literals, hypotheses with common parts have a common path to the root. Therefore, common parts are evaluated only once for all the hypotheses instead of once for each hypothesis. The query-packs can drastically improve performance, particularly with top-down algorithms, since they usually create new hypotheses by adding new literals to a candidate hypothesis, which is therefore common to all new hypotheses. Santos Costa et al. (2003) propose the combined use of several transformations based on additional information on the variable types to speed up subsumption. This additional information allows to recursively decompose the clause into independent parts up to the instantiation of "grounded variables". These techniques are orthogonal to ours so that it may be possible to incorporate them for further performance improvement.

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 and Khardon (2004), was done using the C language and implements only the batch algorithm but otherwise includes all the techniques described above.

In this section we describe several experiments using the system and its subsumption engines. The experiments are designed to test and demonstrate several issues. First, they exemplify the scope and type of applications that may be appropriate. Second, they demonstrate that LOGAN-H can give state of the art performance on relational learning problems, and that it results in very strong hypotheses when the data has some similarity to being "complete" in the sense discussed above. Third, the experiments evaluate the contribution of different speedup techniques in the system. Fourth, the experiments evaluate the performance of different subsumption engines both within the learning system and independently.

The experiments are as follows. We first illustrate the scope and limitation of interactive algorithm on the task of learning complex list manipulation programs. We then discuss a toy grammar learning problem showing that the batch algorithm can be applied to this potentially hard problem using carefully selected data. We then describe a set of quantitative experiments in three benchmark ILP domains: the Bongard domain (De Raedt and 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 to the experiments performed by Maloberti and 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 Experiments with LOGAN-H

This section describes our experiments with LOGAN-H in the 5 domains as outlined above.

5.1.1 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 (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 (1999b) 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. 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.[5] 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.

5.1.2 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 one 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),
```

5. This formalization is not range-restricted so we did not use the restriction to live pairings given in the appendix.

```
                 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, that is, all atoms that are true for this sentence. The base relations identifying properties of words, $det(), n(), pn(), iv(), tv(), relw(), number()$, 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 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. 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.

5.1.3 BONGARD PROBLEMS

The Bongard domain is an artificial domain that was introduced with the ICL system (De Raedt and 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 uniformly at random, and each object is inside some other object with probability 0.5 where the target was chosen uniformly at random 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

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

Table 4: Complexity of Targets.

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

Table 5: Performance summary in average percentage accuracy.

some similarity to the “closure” property of data sets 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

$$\begin{array}{c} circle(X)\ in(X,Y)\ in(Y,Z)\ color(Y,B) \\ color(Z,W)\ black(B)\ white(W)\ in(Z,U) \quad \rightarrow \quad triangle(Y) \end{array}$$

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.

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 and Van Laer, 1995) and Tilde (Blockeel and 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.

5.1.4 ILLEGAL POSITIONS IN CHESS

Our next experiment is in the domain of the chess endgame White King and Rook vs. 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 data set 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 $(\mathtt{a},\mathtt{b})$ on the chess board, the White Rook is in position $(\mathtt{c},\mathtt{d})$, and the Black King in position $(\mathtt{e},\mathtt{f})$. 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)]-
```

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

	25	50	75	100	200	500	1000	2000	3000
Relevant background 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
All background 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
FOIL (Quinlan, 1990)				92.50			99.40		

Table 6: Performance summary for KRK-illegal data set in average percentage accuracy

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

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

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 (as discussed in the appendix). 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 data set especially for small training sets.

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 "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). The caching mechanism described in the appendix 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

Method	Avg. accuracy
Equal Frequency best split `atom-charge`=25 `lumo`=8 `logp`=30	89.41%
Equal Frequency automatic selection	83.62%
Information Gain	82.25%

Table 8: Accuracy results on the Mutagenesis domain.

due do the large number of rules induced. On the other hand with larger table sizes (TH=1000) the randomized method does very well and reproduces the deterministic results.

5.1.5 MUTAGENESIS

The Mutagenesis data set is a structure-activity prediction task for molecules introduced by Srinivasan et al. (1994). The data set 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 data set has been partitioned into 10 subsets for 10-fold cross validation estimates and has been used in this form in many studies (Srinivasan et al., 1994; Sebag and Rouveirol, 2000; De Raedt and 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 relations 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 the interpretations setting as explained in the appendix. In addition, since constants are meaningful in this data set, 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` and hence discretization is needed. We report on experiments with the equal frequency method (Dougherty et al., 1995) and the information gain (IG) method (Fayyad and Irani, 1993) which are discussed in more detail in the appendix. The IG method decides on the number of bins automatically. The equal frequency method requires the number of bins as input. To get reliable results we used double cross validation performing automatic parameter selection over the following range of value: for `atom-charge` $\{5, 15, 25, 35, 45\}$ for `lumo` $\{4, 6, 8, 10, 20, 30, 80\}$ and for `logp` $\{4, 6, 8, 10, 30, 50\}$. For reference we also report the best result that can be obtained in hindsight by searching for good partitions using the test set.

Table 8 summarizes some of the accuracies obtained. As can be seen the overoptimistic "best" results give pretty high accuracy. The automatic selection methods perform quite well giving 83.62% and the IG method comes close with 82.25%.

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 and Rouveirol, 2000) reports results in the range 85%–88% on B3 depending on the values of various tuning parameters, ICL (De Raedt and Van Laer, 1995) reports an accuracy of 84% and finally Laer et al. (1996) report that FOIL (Quinlan, 1990) achieves an accuracy of 83%.

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%

Table 9: Runtime comparison for subsumption tests on Mutagenesis data set.

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 DJANGODUAL and that even with small parameters the randomized methods do very well. An inspection of the hypothesis to the deterministic runs shows that they are very similar.

5.2 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 and 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 and 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 and Saitta, 2000; Giordana et al., 2003; Maloberti and 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, that is, the size of each domain in dual representation, which is set to 50;

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

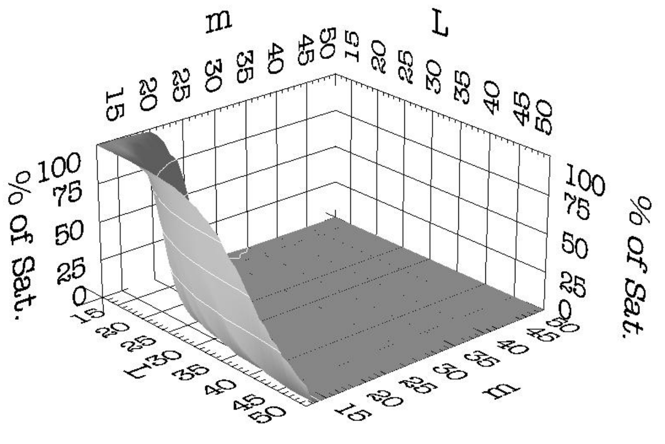


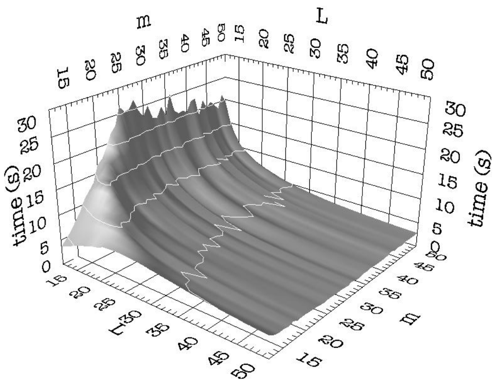
Figure 2: Percentage of subsumption tests satisfied on 50×50 pairs $(\mathcal{C}, Ex)$, where $\mathcal{C}$ is a clause uniformly generated with $n = 12$ variables and m literals (m in [12,50]), and Ex is an example uniformly generated with $N = 50$ literals built on each one of the m predicate symbols 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, that is, 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, N, is constant, thus all domains of the literals of $\mathcal{C}$ have the same size, and each example contains $N \times m$ 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.[6]

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 DJANGODUAL. Figure 3 (B) shows the average execution times for DJANGOPRIMAL. The Phase

6. The experiments performed here are slightly different from the ones in Maloberti and 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 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 and Sebag (2004), 100 clauses and 100 examples were generated, each clause was tested against one example.



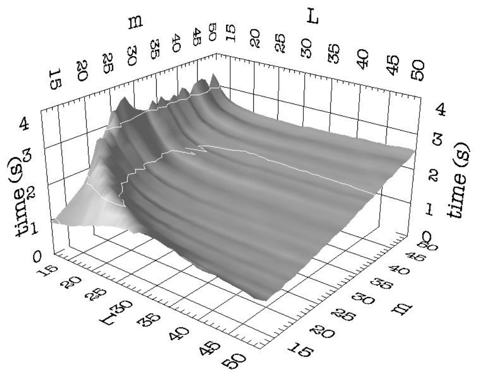


Figure 3: Subsumption cost(m,L) in seconds for (A) DJANGODUAL and (B) DJANGOPRIMAL averaged over 50×50 pairs $(\mathcal{C},Ex)$. A reduction in running time of a factor of 7 is observed in DJANGOPRIMAL compared to Meta-DJANGO (notice difference in scale of the time axis).

Transition phenomenon is also apparent, however DJANGOPRIMAL is 7 times faster than DJANGODUAL on average, despite the simpler versions of algorithms used in DJANGOPRIMAL.

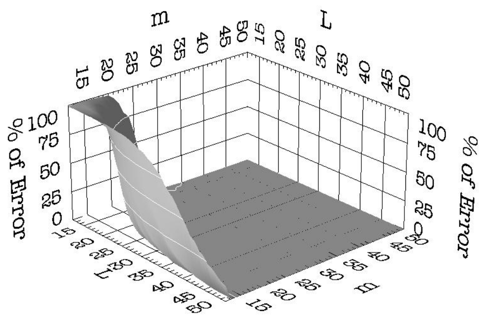
The table based methods are less well suited for the problems in this test suite. The deterministic table methods ran out of memory and could not be used. The randomized methods give a tradeoff between run time and accuracy but they perform worse than DJANGO.

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.

Finally, it is worth recalling the table method of Di Mauro et al. (2003) and the experimental results reported for it in Di Mauro et al. (2003) and Maloberti and Suzuki (2004) for simple subsumption as well as for enumerating all solutions. Both papers report that the table method does solve the challenge problems, so indeed it avoids some of the space problems incurred with our deterministic version. But both papers indicate that DJANGO is at least an order of magnitude faster and often much more, where the relation in run times depend on whether the problems are in the phase transition, over constrained region or under constrained region. These observations seem to agree with our comparisons of the table based and DJANGO methods.

To summarize, the subsumption experiments suggest that in general the DJANGO methods are more robust than the table based methods and that DJANGOPRIMAL is a promising alternative. However, these results must be interpreted with caution, since in these experiments the tables meth-



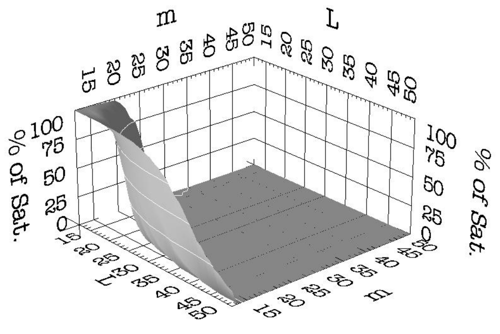
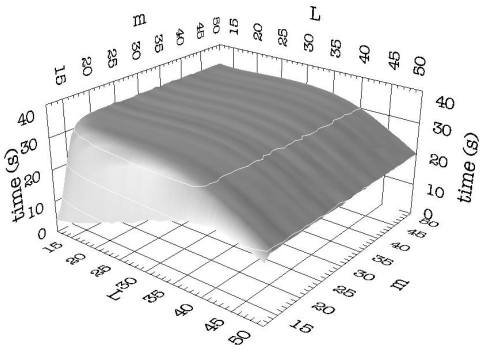


Figure 4: (A) Percentage of wrong subsumption tests for randomized tables method on 50×50 pairs $(\mathcal{C}, Ex)$, with TH= 1 and R= 1. (B) Percentage of satisfiable subsumption tests. Notice that the table based method with such low valued parameters is not able to discover any subsumption substitution,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



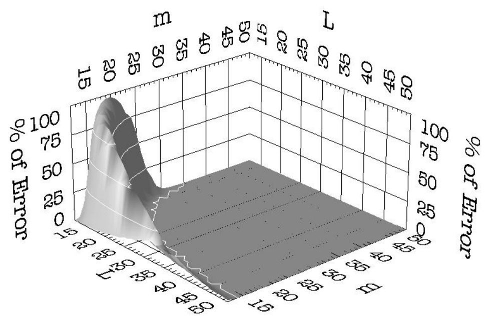


Figure 5: (A) Subsumption cost(m, L) and (B) error percentage for randomized tables method with TH= 10 and R= 10 averaged over 50×50 pairs $(\mathcal{C}, Ex)$.

ods look for all solutions while DJANGOPRIMAL and DJANGO only search for a single 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 over constrained region and moves toward the phase transition and then possibly the under constrained region. Since tables perform well in the over constrained 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 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 (1999b) 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 systems 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 within 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 sub-structures of examples in the data set are likely to be in the data set 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. 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 data sets 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 assays 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.

Acknowledgments

This work was partly supported by EPSRC grant GR/M21409 in the UK and by NSF Grant IIS-0099446 in the USA. In addition, Roni Khardon was supported by a Research Semester Fellowship Award from Tufts University. 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.

Appendix A. Further Details on Implementation and Applicability

The appendix gives additional details on important implementation issues leading to speedup and wider applicability of the system. In addition we give a discussion of using our learning from interpretations system on data given in the normal ILP setting.

A.1 Noisy Data Sets

Many real life data sets are "noisy", in the sense that there is no Horn expression hypothesis that is consistent with all the data. In this case the batch algorithm described above is not well defined since it assumes in step 2(b) that at least one potential consequent is correct and is therefore not removed in *one-pass*. However, if the data set is inconsistent it is possible that all potential consequents are removed. If this happens our system simply marks the counter-example as inconsistent and ignores it in future tests.

A related problem occurs when we use randomized subsumption tests. Here since the subsumption test is incomplete we may not notice that a rule in the hypothesis is violated by a negative example. As a result the algorithm may see the same negative counter-example multiple times. Note that since pairings move atoms from antecedent to conclusion the same counter-example may be encountered more than once even under normal conditions. To handle this, the system uses a small constant (5 in the experiments) to bound the number of times an example may be used as a counter-example. If this bound is exceeded the negative example is ignored in future tests.

A.2 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 (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 cache hits. Caching can reduce or increase run time of the system, depending on the data set, the cost for subsumption for examples in the data set, and the rate of cache hits.

A.3 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 antecedent of the pairing. One can show that if the target expression is range restricted (that is, 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 antecedent 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.

A.4 Discretization of Real-Valued Arguments

The system includes a capability for handling numerical data by means of discretization. This is standard in machine learning systems but several aspects are peculiar to the relational setting and to bottom up learning.

Notice first that unlike attribute-value data there may be more than one occurrence of the same type of value in the same example, in different arguments of a predicate, different instances of the same predicate or different predicates. To capture this we first divide the numerical attributes into "logical groups" by annotating the data set. For example the rows of a chess board will belong to the same group regardless of the predicate and argument in which they appear. The system can then determine the threshold values and possibly the number of bins to divide values into. 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. Also, we use predicate names that explicitly encode thresholds numbers, for example, `logp_val>00` or `logp_val<02`.

Several approaches to discretization choosing the number of bins and the boundaries have been proposed in the literature (Fayyad and Irani, 1993; Dougherty et al., 1995; Blockeel and De Raedt, 1997). Our experiments use the following two approaches. 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. To select the number of bins automatically one must use some version of cross validation. The *Information Gain* (IG) approach introduced by Fayyad and 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. 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 and De Raedt (1998) where values were weighted according to the number of times they appear in the examples. However, Blockeel and De Raedt (1998) set the number of bins by hand since IG provided too few regions. In our experiments we use the original criterion.

An interesting aspect arises when using discretization which highlights the way our system works and potential limitations. 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 point is illustrated by the following experiment discretizing values in the KRK domain. In this domain 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. 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. As can be seen in Table 10 good accuracy is maintained with discretization but the learning curve is much slower than the non discretized version presented earlier. Another 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 discretization. With "relevant background mode" the discretization threshold predicates and the adjacent predicates are different in every example. Since 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.

A.5 Pruning Rules

The system performs bottom up search and may stop with relatively long rules if the data is not sufficiently rich (that is, 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 data set. For each atom in the rule the system evaluates whether the removal of the atom increases the error on the validation set. If not,

	25	50	75	100	200	500	1000	2000	3000
Relevant background mode	43.32	43.70	45.05	44.60	52.39	72.26	84.80	90.30	92.17
All background mode	67.27	72.69	75.15	78.00	82.68	88.60	91.03	91.81	92.01

Table 10: Performance summary for KRK-illegal data set with discretization in average percentage accuracy.

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.

A.6 Restricting Legal Conclusions

Our algorithm allows any predicate and instantiation to serve as a conclusion in a clause. However, when applying to a real data set we may know (or want to set up) that some predicates serve as rule conclusions and others do not. This is simply implemented in the *rel-cands* operation. This feature is useful both for applicability and in reducing run time that would be needed to generate, evaluate and remove rules with wrong conclusions.

A.7 Applicability to Data from Normal ILP Setting

In the normal ILP setting (Muggleton and 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, 1999b). 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 $I' = ([V], [BK(b) \cup \{b\}])$ 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 $BK(b)$ only implies b.

If b is a negative example in the standard ILP setting, we construct an interpretation $I = ([V],[BK(b)])$, where V is the set of variables appearing in $BK(b)$, and label it positive. Notice that if the system ever considers the clause $BK(b) \rightarrow b$ as a candidate, *one-pass* will find the positive interpretation I and will drop b, as desired. In fact, *one-pass* will return $[BK(b),\emptyset]$ for any input clause set $[BK(b),c]$ since it will drop all consequents in c that are not included in $BK(b)$ (including `false`), thus precluding clause $BK(b) \rightarrow b$ from ever being included in any hypothesis. This again assumes that no consequent can be implied by $BK(b)$.

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 do not need example identifiers and 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 $I' = ([V],[BK(b) \cup \{b\}])$ 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 data set 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.

A.8 Algorithmic Details on DJANGO

This section provides some of the details on the algorithms used in DJANGOPRIMAL and DJANGODUAL and the difference between them. Both versions 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 the 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 built 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 are used to propagate the effects of the last assignment:

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 and Saitta, 2000). Instances in the under-constrained region use FC, while instances in the phase transition and the over constrained 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 and Sebag (2004).

References

D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

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

M. Arias and R. Khardon. Bottom-up ILP using large refinement steps. In *Proceeding of the International conference on Inductive Logic Programming*, pages 26–43. Springer, 2004. LNAI 3194.

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

H. Arimura. Learning acyclic first-order Horn sentences from entailment. In *Proceedings of the International Conference on Algorithmic Learning Theory*, pages 432–445. Springer, 1997. LNAI 1316.

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

H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 77–84. Springer, 1997. LNAI 1297.

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

H Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.

H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In *Proc. Tenth International Conference on Inductive Logic Programming*, LNAI 1866, pages 60–77. Springer-Verlag, Berlin, 2000.

H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, April 1987.

I. Bratko. Refining complete hypotheses in ILP. In *International Workshop on Inductive Logic Programming*, pages 44–55. Springer, 1999. LNAI 1634.

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

C. Chang and J. Keisler. *Model Theory*. Elsevier, Amsterdam, Holland, 1990.

X. Chen. *A Theoretical Comparison of Selected CSP Solving and Modeling Techniques*. PhD thesis, University of Alberta, Canada, 2000.

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

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

L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 6th Conference on Algorithmic Learning Theory*. Springer, 1995. LNAI 997.

R. Dechter and J. Pearl. Structure identification in relational data. *Artificial Intelligence*, 58:237–270, 1992.

N. Di Mauro, T.M.A. Basile, S. Ferilli, F. Esposito, and N. Fanizzi. An exhaustive matching procedure for the improvement of learning efficiency. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, pages 112–129. Springer, 2003. LNAI 2835.

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

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

I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 246–252, 1996.

A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 2:217–251, 2000.

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

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

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

R. Khardon. Learning range-restricted Horn expressions. In *Proceedings of the Fourth European Conference on Computational Learning Theory*, pages 111–125. Springer, 1999a. LNAI 1572.

R. Khardon. Learning function free Horn expressions. *Machine Learning*, 37(3):249–275, 1999b.

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

M. Krishna Rao and A. Sattar. Learning from entailment of logic programs with local variables. In *Proceedings of the International Conference on Algorithmic Learning Theory*. Springer, 1998. LNAI 1501.

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

J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987. Second Edition.

A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

J. Maloberti and M. Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.

J. Maloberti and E. Suzuki. An efficient algorithm for reducing clauses based on constraint satisfaction techniques. In *Proceedings of the 14th International Conference on Inductive Logic Programming*, pages 234–251. Springer Verlag LNAI 3194, 2004.

T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

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

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

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

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

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

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

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

G. D. Plotkin. A further note on inductive generalization. *Machine Intelligence*, 6:101–124, 1971.

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

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

C. Reddy and P. Tadepalli. Learning first order acyclic Horn programs from entailment. In *International Conference on Inductive Logic Programming*, pages 23–37. Springer, 1998. LNAI 1446.

V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.

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

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

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

C. Stolle, A. Karwath, and L. De Raedt. CLASSIC'CL: An integrated ILP system. In *Discovery Science*, pages 354–362, 2005.

E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.