FOIL-D: Efficiently Scaling FOIL for Multi-relational Data Mining of Large Datasets

Joseph Bockhorst° and Irene Ong°

Department of Computer Sciences
University of Wisconsin, Madison WI 53706
joebock@cs.wisc.edu,ong@cs.wisc.edu

Abstract. Multi-relational rule mining is important for knowledge discovery in relational databases as it allows for discovery of patterns involving multiple relational tables. Inductive logic programming (ILP) techniques have had considerable success on a variety of multi-relational rule mining tasks, however, most ILP systems do not scale to very large datasets. In this paper we present two extensions to a popular ILP system, FOIL, that improve its scalability. (i) We show how to interface FOIL directly to a relational database management system. This enables FOIL to run on data sets that previously had been out of its scope. (ii) We describe estimation methods, based on histograms, that significantly decrease the computational cost of learning a set of rules. We present experimental results that indicate that on a set of standard ILP datasets, the rule sets learned using our extensions are equivalent to those learned with standard FOIL but at considerably less cost.

1 Introduction

Traditional data mining techniques aim to extract patterns from data sets that may be naturally represented in flat files. Conversely, relational databases represent data as a set of interconnected relational tables. Many useful patterns involve multiple tables, and since data in this format often cannot naturally be represented in flat files, traditional propositional data mining methods have difficulties learning such patterns.

Inductive Logic Programming (ILP) [1] algorithms aim to learn a set of first-order logical rules from multi-relational data and thus are well suited to multi-relational data mining tasks. There are however, two practical barriers that must be overcome before ILP systems may be applied to mining of large datasets. (i) ILP systems must deal with a limited amount of physical memory. Most ILP implementations, such as FOIL [2], tacitly assume the whole database fits into main memory. If it does not fit, these programs either crash or grind to an effective halt as they rely on the operating system to manage moving data between physical memory and disk. (ii) ILP systems must be faster. The

[0] Both authors contributed equally to this work

search space of ILP systems is very large and even heuristic methods are slow. Moreover, since the time needed to score an operator typically depends on the database size, direct application of most ILP systems to very large datasets is impractical.

In this paper, we describe extensions to the ILP algorithm FOIL that address both of these barriers. To deal with limited physical memory we leverage off the considerable effort that has been addressed to this issue in the design of relational database management systems (RDBMSs). We show how to succinctly express FOIL's operations in terms of SQL statements that RDBMSs have been optimized to execute. To deal with time cost we describe probabilistic models that we use to estimate the gain of FOIL's operators. We show how to use these models to significantly speed up learning. We present experimental results that indicate that with our estimation method we are able to learn the same rules as standard FOIL on several standard ILP datasets, but in significantly less time.

The interest in scaling up ILP for relational data mining in large datasets has been growing as Dimaio and Shavlik [3], Tang et. al [4] and Mooney et. al [5] have recently published research in this direction. Although the idea of incorporating the ability to learn first order rules from RDBMSs is not new – Stonebraker et. al [6] added this feature to Postgres and Brockhausen and Morik [7] have directly implemented an ILP algorithm (RDT) to a RDBMS – the use of probabilistic models to estimate scores for learning rules has, to the best of our knowledge, not been done before. Similar estimation ideas have been used in the area of query-optimization in databases [8, 9].

2 FOIL

Quinlan's first-order inductive learner [2, 10] (FOIL) is a popular ILP algorithm that learns function-free first order rules for a target relation. FOIL requires as input a set of *extensionally* defined relations. That is, each input relation is defined by a listing of its tuples rather than *intensionally* as set of logical rules. This parallels closely the organization of data in relational databases where we can think of the tuples of a table as defining a relation. One of the input relations, which we refer to as POS, is designated as the target relation. Another input relation, which we refer to as NEG, is of the same arity as POS and contains tuples that do not belong to the target relation. These two relations define FOIL's *initial* positive and negative tuple set. The other relations $B1, ..., BN$ serve as background knowledge.

Given its input, FOIL learns rules of the form

$$POS(X_1, ..., X_m) \leftarrow L_1 \wedge L_2 \wedge ...$$

where each L_i is a (possibly negated) literal and $X_1 \dots X_m$ are distinct variables. The goal of FOIL is to discover a set of rules that entails all of the tuples in POS and none of the tuples in NEG.

Table 1. The FOIL algorithm. Given a set of tuples, POS, in the target relation, a set of tuples, NEG, not in the target relation and sets for tuples $B_1, ..., B_M$ that define M background relations, FOIL returns a set of first-order rules for the target relation. The symbol $\Leftarrow$ indicates variable assignment and the symbol $\leftarrow$ indicates logical implication.

```
 1: procedure FOIL(POS, NEG, B_1, ···, B_N )
 2:   learnedRules ⇐ {}
 3:   POS_unc ⇐ POS                        ▷ start with all positive tuples uncovered
 4:   repeat                                              ▷ begin FOIL's outer loop
 5:     newRule ⇐ learnOneRule()
 6:     update POS_unc            ▷ remove tuples in POS_unc covered by newRule
 7:     add newRule to learnedRules
 8:   until |POS_unc| == 0
 9:   return learnedRules
10: end procedure

11: procedure learnOneRule()
12:   newRuleBody ⇐ {}                        ▷ start with general rule POS ← true
13:   POS_curr ⇐ POS_unc                  ▷ all positive tuples are covered by newrule
14:   NEG_curr ⇐ NEG                      ▷ all negative tuples are covered by newrule
15:   repeat                                              ▷ begin FOIL's inner loop
16:     bestLit ⇐ getBestLiteral(candidateLiterals())
17:     conjoin bestLit to newRuleBody                   ▷ add literal to growing rule
18:     update POS_curr                             ▷ arity of POS_curr may increase
19:     update NEG_curr                             ▷ arity of NEG_curr may increase
20:   until |NEG_curr| == 0
21:   return POS ← newRuleBody
22: end procedure

23: procedure getBestLiteral(C)                       ▷ C is set of candidate literals
24:   maxGain ⇐ −∞
25:   for all candLit ∈ C do
26:     p′ ⇐ |POS_curr| if candLit is added to newrule
27:     n′ ⇐ |NEG_curr| if candLit is added to newrule
28:     p^++ ⇐number of tuples in POS_curr covered by candLit
29:     gain ⇐ gain(p′, n′, p^++)
30:     if gain > maxGain then
31:       bestLiteral ⇐ literal
32:       maxGain ⇐ gain
33:     end if
34:   end for
35:   return bestLiteral
36: end procedure
```

Table 1 presents a high level outline of FOIL. FOIL is a covering algorithm that on each iteration of its outer loop, which starts at Line 4[1], adds a single clause to its learned rule set that logically entails (covers) some of the previously uncovered tuples in POS and none of the tuples in NEG. FOIL's method of building a single rule, shown in the LEARNONERULE procedure of Table 1, begins with the general rule $POS \leftarrow \texttt{true}$, which covers all positive and negative tuples. This rule is then specialized by greedily conjoining literals one at a time to the body until the new rule covers no negative tuples.

One trait of FOIL that distinguishes it from most propositional supervised learning algorithms is the dynamic nature of the positive and negative example training sets. When deciding which literal to append to the body of the new rule, FOIL considers only the *current* positive and negative tuple sets, POS_{curr} and NEG_{curr} in Table 1. From one iteration to the next, these sets may shrink, grow, increase in arity or some combination thereof.

At the start of the LEARNONERULE procedure POS_{curr} is initialized to the tuples of POS that are not covered by any rule learned so far, POS_{unc}, and NEG_{curr} is set to all the tuples in NEG. Following the addition of the chosen literal (Line 17), FOIL updates POS_{curr} and NEG_{curr}. Let $L(N_1, ..., N_a, O_1, ..., O_b)$ be the chosen literal where N_i is a *new* variable that does not appear anywhere else in the new rule and O_i is an *old* variable that appears either in the head or a previously introduced literal. If L is unnegated, the arity of tuples in the updated POS_{curr} and NEG_{curr} sets increases by a, the number of new variables. A tuple t in POS_{curr} (or NEG_{curr}) gives rise to a (possibly expanded) tuple in the updated set for each tuple in the relation associated with the chosen literal that matches t on the arguments indicated by the old variables.

FOIL uses an information theoretic heuristic to determine which literal to append to the body of a growing rule. On each iteration of its inner loop, which starts on Line 15, FOIL chooses the literal that has maximum gain where it defines the gain of literal L_i as

$$gain(L_i) = p^{++} \times \left\{ \log(\frac{p'}{p' + n'}) - \log(\frac{p}{p + n}) \right\}.$$

Here p and n are the sizes of POS_{curr} and NEG_{curr}, p' and n' are the sizes of the updated POS_{curr} and NEG_{curr} if L_i were added and p^{++} is the number of tuples in POS_{curr} that are covered by L_i.

The main computational cost of FOIL comes from the evaluation of all the candidate literals every time a new literal is added to a growing clause. If the input data set cannot fit in main memory, management of the current positive and negative tuple becomes more complicated. In the next section we show how, if our data is stored in relational database tables, these tasks can be concisely expressed by a small number of SQL statements.

[1] This and all subsequent references to line numbers refer those in Table 1.

3 FOIL-D

In this section we present FOIL-D, our implementation of FOIL that interfaces directly with relational databases. FOIL-D assumes operations that involve manipulation of the relational data may not fit in main memory and thus provides database operations, in terms of SQL statements, for them. FOIL-D does however, assume that other operations, such as generating and storing the candidate literals, fit in main memory.

One difference between FOIL and our current implementation of FOIL-D is that for simplicity FOIL-D only considers unnegated literals while FOIL considers both unnegated and negated literals. There is nothing fundamental that prevents FOIL-D from considering negated literals though, and we plan to add support for them in the future.

3.1 Database organization and operations

Let the tuples defining the input relations be in database tables named POS, NEG, B1, ..., BN where the mapping between tables and the relations in Table 1 is the obvious one. If the data is not in such a format, temporary tables can be constructed. We store the uncovered positive tuples and the current positive and negative examples in database tables named POS_UNC, POS_CURR and NEG_CURR respectively. Due to the dynamics of the training sets, the number of columns of POS_CURR and NEG_CURR may change during the course of the algorithm. At any time though there is a one-to-one correspondence between the columns of POS_CURR (and NEG_CURR) and the distinct variables that have been introduced by either the head or a literal in the body of the growing clause.

The left-hand column of Table 2 lists the line numbers from Table 1 where FOIL-D issues database queries. The right-hand column gives the SQL statements[2] for the corresponding line. We now discuss each of these operations in turn.

Line 3 Here we initialize POS_UNC to all the tuples in POS. SQL statements of the from `CREATE` *new-table-name* `LIKE` *existing-table-name* create a new empty table that has the same column names and types as the existing table indicated.

Line 6 Here we remove from POS_UNC those tuples covered by the rule just learned. We first save the tuples of POS_UNC into the temporary table OLD_POS_UNC and recreate an empty POS_UNC. The `INSERT INTO` statement fills POS_UNC with those tuples of OLD_POS_UNC that do not have any matching tuple in POS_CURR, the expanded tuples covered by the new rule. The semantics of `LEFT JOIN` is to include at least one tuple from the "left" table (OLD_POS_UNC in this case) even if none of them are selected

[2] These statements are compatible with version 4.1 of MySQL (http://www.mysql.com)

Table 2. SQL statements used by FOIL-D. The left-hand-column indicates line numbers from Table 1 where FOIL-D issues database queries. The right-hand-column lists the corresponding SQL statements.

Line 3	CREATE TABLE POS_UNC LIKE POS INSERT INTO POS_UNC SELECT * FROM POS
Line 6	ALTER TABLE POS_UNC RENAME OLD_POS_UNC CREATE TABLE POS_UNC LIKE POS INSERT INTO POS_UNC SELECT *cols* FROM OLD_POS_UNC LEFT JOIN POS_CURR ON *cond* WHERE *cond′* DROP TABLE OLD_POS_UNC
Line 13	CREATE TABLE POS_CURR LIKE POS INSERT INTO POS_CURR SELECT * FROM POS_UNC
Line 14	CREATE TABLE NEG_CURR LIKE NEG INSERT INTO NEG_CURR SELECT * FROM NEG
Line 18	ALTER TABLE POS_CURR RENAME OLD_POS_CURR CREATE TABLE POS_CURR (...) INSERT INTO POS_CURR SELECT *cols* FROM OLD_POS_CURR, *rel*(*bestLit*) WHERE *cond* DROP TABLE OLD_POS_CURR
Line 19	ALTER TABLE NEG_CURR RENAME OLD_NEG_CURR CREATE TABLE NEG_CURR (...) INSERT INTO NEG_CURR SELECT *cols* FROM OLD_NEG_CURR, *rel*(*bestLit*) WHERE *cond* DROP TABLE OLD_NEG_CURR
Line 26	SELECT COUNT (*) FROM POS_CURR, *rel*(*candLit*) WHERE *cond*
Line 27	SELECT COUNT (*) FROM NEG_CURR, *rel*(*candLit*) WHERE *cond*
Line 28	SELECT DISTINCT COUNT (*) FROM POS_CURR, *rel*(*candLit*) WHERE *cond*

with the condition in the `ON` clause, provided they pass the condition in the `WHERE` clause. These tuples have `null` values for any columns that come from the "right" table (POS_CURR). The statement FOIL-D issues selects only those tuples of OLD_POS_UNC that do not match any in POS_CURR by setting $cond'$ to $col = \mathsf{null}$ where *col* is a column in POS_CURR. The condition *cond* in the `ON` clause is a set of equality constraints between columns in OLD_POS_UNC and columns in POS_CURR that correspond to variables in the target relation.

Lines 13 and 14 Here we initialize POS_CURR and NEG_CURR, the current positive and negative example set.

Lines 18 and 19 Here we update POS_CURR and NEG_CURR after we add the highest scoring literal *bestLit* to the growing rule. The statements to update POS_CURR are analogous to those to update NEG_CURR. For simplicity, we only describe those for updating POS_CURR. Before getting the tuples with the `INSERT INTO` statement, we save the old tuples in a temporary table OLD_POS_CURR and create a new POS_CURR table that will hold the updated examples. The new POS_CURR table will have a column for each column in OLD_POS_CURR and each new variable in *bestLit*. The `INSERT INTO` statement joins OLD_POS_CURR with the relation of *bestLit*, *rel*(*bestLit*). The condition *cond* in the `WHERE` clause is a conjunction of equalities, one for each old variable in *bestLit*, between columns in OLD_POS_CURR and the corresponding columns in *rel*(*bestLit*). The projection *cols* lists the columns in OLD_POS_CURR along with the columns in *rel*(*bestLit*) that correspond to the new variables in *bestLit*.

Lines 26-27 Here we compute counts p' and n' that, along with p^{++}, we need to compute the gain of the candidate literal *candLit*. The conditions *cond* in the `WHERE` clauses are conjunctions of equalities, one for each new variable in *candLit*, and are the same for both statements. If *candLit* becomes *bestLit*, *cond* will also be used in the `INSERT INTO SELECT` statements issued when updating POS_CURR and NEG_CURR on Lines 18 and 19.

Line 28 Here we compute p^{++}. The `DISTINCT` keyword assures that tuples in POS_CURR are counted at most once. The condition *cond* is the same as the one used on Line 26 to compute p.

3.2 Computational cost of FOIL-D

The primary computational cost of running FOIL-D on large databases comes from the database *join* operations used to execute the six conditional `SELECT` statements (Lines 6, 18, 19, 26, 27 and 28). A join operation ($r \bowtie s$) between tables r and s selects a subset of the tuples in the cross product $r \times s$ that match a specified set of constraints. The implementation of the join operation is a heavily studied topic in database research. See, for example, Ramakrishnan's textbook [11]. The cost of a join depends on a number of properties of the join such as the number and types of constraints (*eg*, $>, <, =$), the presence of any database indexes on the join columns, and the number of tables. In FOIL-D all joins involve only equality constraints (*equi-joins*) between two tables. FOIL-D does

perform joins with $k > 1$ equality constraints (k-column joins). In this paper, we are agnostic about the implementation of join but measure the computational cost by the total number of joins performed to learn a theory.

Inspection of Tables 1 and 2 reveals that the number of joins needed to learn a rule set of R rules with L total literals where C total candidates are considered is

$$\# \text{ of join operations} = R + 2L + 3C$$

The number of rules in a learned theory R is typically small, often less than five, and the number of literals L is also manageable, in the ten's at most. The number of candidates C, however, is often much larger and thus C dominates the others. The number of candidate literals considered at *each* step depends most strongly on the arity of the maximum arity relation and the number of old variables introduced so far [12]. For example, the number of candidate literals considered to append to a rule with 5 old variables and max arity of 3 is 136 [12]. FOIL-D uses type constraints on the arguments of the relations (or the columns in the database) which reduces the total number of candidate literals somewhat but not so much that it does not dominate in the above expression. Next, we describe extensions to FOIL-D that reduce the total number of join operations needed to learn a theory.

4 FOIL-DH

To get the counts p', n' and p^{++} needed to compute the gain of a candidate literal we only need the *number* of tuples in the result sets of the SQL statements listed in Table 2 that we execute on Lines 26-28. Thus, one way to reduce the total number of joins is to compute p', n' and p^{++} by more efficient means. The approach we consider here is, using histograms, to construct probabilistic models of the tuples in each table and to use these models to estimate the counts. We call this system "FOIL-D with histograms" or simply FOIL-DH.

We maintain a histogram for each column of POS_UNC, NEG, B1, ..., BN, POS_CURR and NEG_CURR. Let $h^{r.c}$ be the histogram for the column $r.c$ of table r. The domain of $h^{r.c}$ is the same as the domain of the type of column $r.c$. The count $h^{r.c}(v)$ for value v is the number of tuples in r for which the value of column $r.c = v$.

4.1 Estimating p' and n'

Now we show how, given the column histograms and an independence assumption, to quickly estimate the size of any k-column equi-join, such as the ones to get p' and n'.

The probability $p^{r.c}(v)$ that a randomly selected tuple in table r has value v in column c is

$$p^{r.c}(v) = h^{r.c}(v)/|r|.$$

For an equi-join between columns c of table r and c' of table s the probability that a randomly selected tuple from the cross product $r \times s$ satisfies the equality constraint, and thus is included in the result set, is

$$p(r.c, s.c') = \sum_v p^{r.c}(v) \times p^{s.c'}(v)$$

where the sum is over all values in the type of column $r.c$ (and $s.c$). The number of tuples in the result set is exactly given by

$$size(r \bowtie_1 s) = |r||s| \times p(r.c, s.c')$$

where $\bowtie_1$ indicates a 1-column join between r and s.

This is an exact calculation because the nature of the cross product guarantees that for a randomly selected tuple in $r \times s$, the value in a column from r is statistically independent of the value of a column from s. For multi-column joins the summary statistics in the histograms are not sufficient to exactly compute the size of the result set. If we assume, however, that the values of all columns in the join from the *same* table are statistically independent, we can estimate the size of a $k-$column join as

$$\hat{size}(r \bowtie_k s) = |r||s| \prod_{i=1}^{k} p(r.i, s.i) \tag{1}$$

where here without loss of generality we assume column $r.i$ is joined with $s.i$ for $1 \leq i \leq k$.

Our estimates of p' and n' for candidate literal *candLit* then are

$$\hat{p}' = \hat{size}(\text{POS_CURR} \bowtie_b rel(candLit))$$

and

$$\hat{n}' = \hat{size}(\text{NEG_CURR} \bowtie_b rel(candLit))$$

where here $rel(candLit)$ is the relation of *candLit* and b is the number of old variables in *candLit*.

4.2 Estimating p^{++}

In order to compute the gain of *candLit*, in addition to p' and n', we need p^{++}, the number of tuples in POS_CURR (pre-update) still covered by the new rule if we accept *candLit*. Thus, we need to estimate the number of tuples in $r =$ POS_CURR that give rise to at least one tuple in $r \bowtie_k s$ where s is $rel(candLit)$.

To estimate p^{++}, we first compute q, the probability a randomly selected tuple in $r \times s$ matches on join columns 2 through k given the independence assumptions as

$$q = \prod_{i=2}^{k} p(r.i, s.i)$$

where again we assume column $r.i$ is joined with $s.i$ for $1 \leq i \leq k$. If $k = 1$ we set q to 1.0. If we randomly pick with replacement j tuples from $r \times s$, the probability that at least one matches on join columns 2 through k is

$$m(j) = (1.0 - (1.0 - q)^j).$$

A tuple in r with value v in the first join column has $h^{s.1}(v)$ tuples in the cross product that match in the first column and this many "chances" to match on the remaining $k-1$ columns. So, we estimate the probability that the tuple will have at least one match in the result set as $m(h^{s.1}(v))$. Summing over all values and multiplying by $h^{r.1}(v)$ gives our estimation of p^{++}:

$$\hat{p}^{++} = \sum_v h^{r.1}(v) * m(h^{s.1}(v)).$$

Our choice of doing this final sum over values of the first join column is arbitrary. We could have chosen any i of the k join columns as the one to do the final sum over where then we calculate q as the probability of a match on the $k-1$ columns excluding join column i. Due to errors introduced by the sampling with replacement assumption of $m(j)$, in general the estimate $\hat{p}^{++}$ will be different for each choice of final join column i. In practice however, we have found $\hat{p}^{++}$ to be close for any choice of i.

As with the expression for estimating p' and n', our estimation $\hat{p}^{++}$ is exact for 1-column joins. Thus, we can exactly compute the gain for candidate literals with exactly 1 old variable given the column histograms.

4.3 Estimating the highest gain literal

We speed up FOIL-D by using estimations of p', n' and p^{++} to estimate the highest scoring candidate literal in two ways. The first method simply estimates the gain of every candidate literal directly with $\hat{p}', \hat{n}'$ and $\hat{p}^{++}$ and chooses the one with highest estimated gain to append to the growing clause. This method eliminates the $3C$ joins needed to estimate the gain of the C candidate literals thereby reducing the number of joins performed to learn a theory with R rules and L literals to

$$\#\text{ of join operations} = R + 2L.$$

A problem with this method is that the independence assumption often causes the estimated gains of the highest-scoring candidate literals with 2 or more old variables to be less than their true gains. In cases where the candidate literal with highest gain has multiple old variables, this procedure often erroneously estimates the highest scoring literal to be one with only 1 old variable.

We find, however, the relative ranks of the candidate literals with multiple old variables, especially the highest scoring ones, is relatively stable following the estimation procedure. This leads to our other use of $\hat{p}', \hat{n}'$ and $\hat{p}^{++}$.

We also use the estimates of the candidate literal gains as a filter to reduce the number of candidate literals whose gains are computed exactly. Given filter size

F we pick a candidate literal to conjoin to the growing clause with this method as follows. First, we compute the estimated gains of the entire candidate set using our histograms as described above. Next, we rank the candidate literals with two or more old variables by estimated gain and compute the exact gain for the top F. Finally, we choose that the literal with the highest true gain among these F and the top scoring candidate literal with a single old variable. We call this method FOIL-DH(F).

The number of joins performed to learn a theory with FOIL-DH(F) is

$$\#\text{ of join operations} = R + 2L + 3FL$$

which is still much smaller than $R + 2L + 3C$ for small F.

4.4 Richer Probabilistic Models

As just described, the current implementation of FOIL-DH estimates the size of a multi-column join by assuming that, for a randomly selected tuple from either of the tables in the join, the values for the join columns are independent of one another. If the join columns are highly correlated, this assumption is likely to lead to inaccurate gain estimates and could result in poor rules.

One way to address this pitfall is to explicitly represent the dependencies between columns with a more complex probability model, such as a *Bayesian network*[3][13], for each table's tuples. Using Bayesian networks, the product on right-hand-side of the Equation 1 would be replaced with the probability, as given by the Bayesian networks for the two tables, that a randomly selected tuple from the cross-product would match on the join columns.

5 Experimental results

This section describes experiments conducted and results obtained by FOIL-D, FOIL-DH and FOIL-DH(F) on three learning tasks taken from machine learning literature, which were used by [2] to illustrate the power of FOIL. Descriptions of the domains are taken from [2]. The aim of the experiments performed is to determine whether the cost of obtaining accurate hypotheses, with respect to FOIL, can be significantly reduced by using probabilistic models (FOIL-DH) and filtering (FOIL-DH(F)) to estimate the scores of literals.

5.1 Learning connectivity of a network

Our first learning task involved learning the definition of *can-reach* in the network from [2], shown in Figure 1. Extensional definitions of *can-reach*(N,N), *not-can-reach*(N,N) as well as *linked-to*(N,N) were given as positive example,

[3] The current version of FOIL-DH corresponds to a Bayesian network for each table that has one vertex for each column and no edges.

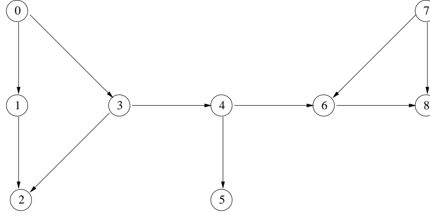

Fig. 1. A small network, where ⟵ indicates *linked-to*, from [2]

Table 3. Rules learned for *can-reach*

FOIL-D, FOIL-DH(1-6)	can-reach(X0, X1) ← linked-to(X0, X1) can-reach(X0, X1) ← linked-to(X0, X2), can-reach(X2, X1)
FOIL-DH(0)	can-reach(X0, X1) ← linked-to(X0, X2), linked-to(X3, X1), linked-to(X2, X4), can-reach(X2, X1) can-reach(X0, X1) ← linked-to(X0, X2), linked-to(X3, X1), linked-to(X2, X4), linked-to(X4, X5), linked-to(X6, X3)

negative example and background knowledge respectively. Variables in this relation consists of one type N (Node). The goal is to learn a general definition for *can-reach*. Table 3 shows the rules learned by FOIL-D and FOIL-DH(0-6), which are using filter sizes set to 0 through 6.

5.2 Learning eastbound trains

The second learning task is from the INDUCE system by [14] and described by [2]. Trains in Figure 2 have different numbers of cars, with various shapes and tops, carrying loads of various number and shape. The task is to distinguish between *eastbound* and *westbound* trains. The target relation *eastbound*(T) is to be defined in terms of the following relations has_car(T,C), not_has_car(T,C), in_front(C,C), behind(C,C), long(C), short(C), open_rectangle(C), not_open_rectangle(C), hexagon(C), not_hexagon(C), bucket(C), not_bucket(C), ellipse(C), not_ellipse(C) rectangle(C), not_rectangle(C), u_shaped(C), not_u_shaped(C), jagged_top(C), not_jagged_top(C), peaked_top(C), not_peaked_top(C), flat_top(C), not_flat_top(C), arc_top(C), not_arc_top(C), open_top(C), closed_top(C), contains_no_load(C), contains_load(C,LS), one_load(C), two_load(C), three_load(C), two_wheels(C), three_wheels(C), double(C), and not_double(C).

The variables are of three types: T (train_type), C (car_type) and LS (load_shape_type). Clauses learned by FOIL-D and FOIL-DH(0-6) are shown in Table 4.

5.3 Learning family relationships

The third and final task we consider is that of learning family relationships, described by [15]. Figure 3 shows two isomorphic families of twelve members each.

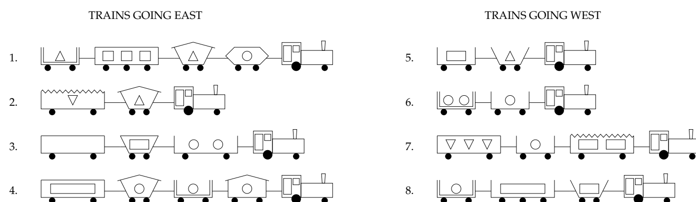

Fig. 2. Eastbound and westbound trains from [14] and [2]

Table 4. Rules learned for *eastbound*

FOIL-D, FOIL-DH(0-6)	eastbound(X0) ← has-car(X0, X1), closed-top(X1), short(X1)

There are twelve relationship types to be learned: *mother*, *father*, *wife*, *husband*, *son*, *daughter*, *sister*, *brother*, *aunt*, *uncle*, *niece* and *nephew*. Each target relation to be learned are to be defined in terms of the following background relations: mother(P,P), father(P,P), wife(P,P), husband(P,P), son(P,P), daughter(P,P), sister(P,P), brother(P,P), aunt(P,P), uncle(P,P), niece(P,P), nephew(P,P), and their negations. The variables are all of type P (People).

The rules learned by FOIL-DH(1) for this task are identical to those learned by FOIL-D on all twelve relations. Table 5 and Table 6 show the definitions learned by FOIL-D and FOIL-DH(0-6) for *uncle* and *mother* respectively. Since in these families, all uncles are married, the second literal in both *uncle* clauses serves the purpose of asserting that person X0 is a man. The rules learned by FOIL-D and FOIL-DH(1-6) for other relations are similar in structure. FOIL-DH(0) fails to learn any rules[4]

Table 5. Rules learned for *uncle*

FOIL-D, FOIL-DH(1-6)	uncle(X0, X1) ← niece(X1, X0), husband(X0, X2) uncle(X0, X1) ← nephew(X1, X0), husband(X0, X2)
FOIL-DH(0)	No rules learned

When learning the relations *can-reach* and the twelve familial relationships, FOIL-D as well as FOIL-DH(1-6) constructs accurate, compact rules. However, for the *can-reach* relation, FOIL-DH(0) generates long, convoluted clauses that consist of only literals with *one* old variable. Furthermore, FOIL-DH(0) does not

[4] FOIL-DH(0) learns no rules in cases where the first rule it “learns” covers zero positive examples and is discarded.

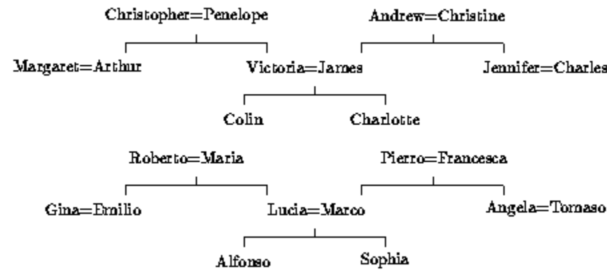

Fig. 3. Two family trees, where = means *married-to*, from [15] and [2]

Table 6. Rules learned for *mother*

FOIL-D, FOIL-DH(1-6)	mother(X0, X1) ← daughter(X1, X0), husband(X2, X0) mother(X0, X1) ← son(X1, X0), husband(X2, X0)
FOIL-DH(0)	No rules learned

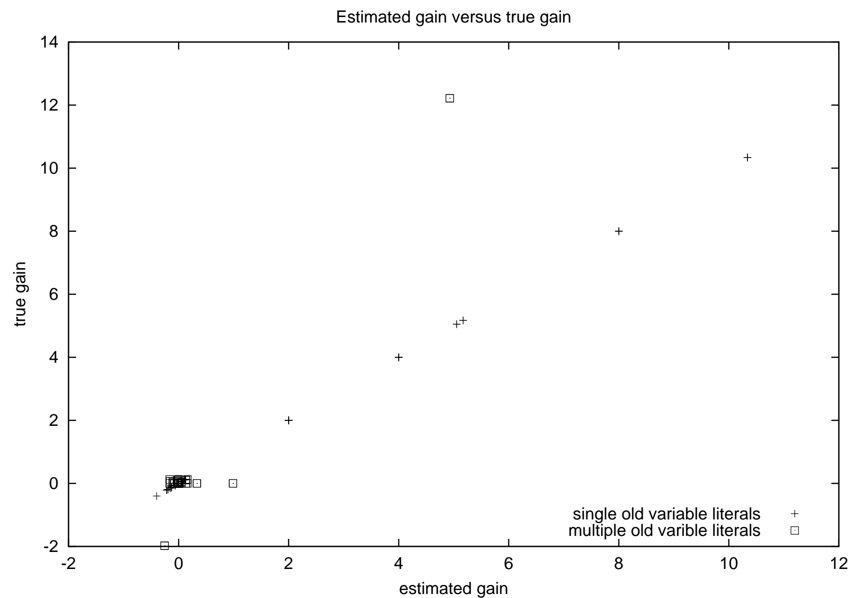

Fig. 4. Estimated gain versus actual gain calculations for adding the first literal in the relation *uncle*. The literal with the highest scoring true gain, *niece(X1,X0)*, has true gain of near 12 and estimated gain of near 5. This literal is not chosen by FOIL-DH(0), but since this literal has the highest estimated score of the multiple old variable literals it is correctly chosen by FOIL-DH(1).

generate any rules for any of the twelve family relationships. The reason for this can be seen in Figure 4. Whenever FOIL-DH(0) makes an estimation of the gain for literals with *multiple* old variables, it is always *under* estimates the actual gain of the highest scoring literals, whereas it always estimates the *exact* actual gain for literals with exactly *one* old variable. Hence, literals with exactly *one* old variable will score higher than literals with *multiple* old variables, resulting in the unique property of the rules learned by FOIL-DH(0) for the relation *can-reach*. This reasoning also supports the results of FOIL-DH(0) on the relationships of the family dataset because the definitions of those relations require literals with *two* old variables to be added first to the body of the clause.

How then was FOIL-DH(1), with a filter size of just 1, able to learn the correct definitions for *can-reach* and the twelve familial relationships? One possible explanation is that even though the estimations for literals with multiple old variables are lower, the order of the estimated gain for these literals is maintained. This would allow FOIL-DH(1), which specifically considers the highest estimated literal with *multiple* old variables, to accurately select the best literal.

Surprisingly, all the different FOIL types in our experiments were able to learn the *exact* rule for *eastbound* trains as shown in Table 4. It is interesting to note that FOIL-DH(0) was able to learn the exact rule for *eastbound* trains, in accordance with our reasoning above, because the rule is comprised of literals with exactly *one* old variable.

Figure 5 shows the total number of JOINs performed for the different FOIL types. FOIL-DH(0) to FOIL-DH(6) grows linearly in the number of JOINs performed and they still provide significantly more savings than FOIL-D.

6 Conclusion

We have presented preliminary methods towards enabling the ILP system FOIL to be applied to multi-relational data mining tasks on large data sets. Our methods address both the space and time hurdles that prohibit standard ILP implementations from being applied to these problems.

To deal with insufficient physical memory we build off of relational database management systems. We have described FOIL-D, a system that mimics the operation of FOIL but that performs the memory intensive FOIL operations using SQL queries to a relational database.

To deal with the slowness of FOIL on very large datasets we have described FOIL-DH, an extension of FOIL-D that uses histograms to quickly estimate the gains of candidate literals without performing expensive database join operations. We also showed how we use the estimation procedure as a filter to select a small number of candidate literals whose gain we compute exactly.

Our experimental results show that while FOIL-DH dramatically reduces the numbers of joins it sometimes fails to learn the correct theories on a set of standard ILP problems because of erroneous estimates. If, however, we use the estimates as a filter we are able to learn the same correct theories as FOIL-D on the datasets we looked at while performing significantly fewer joins.

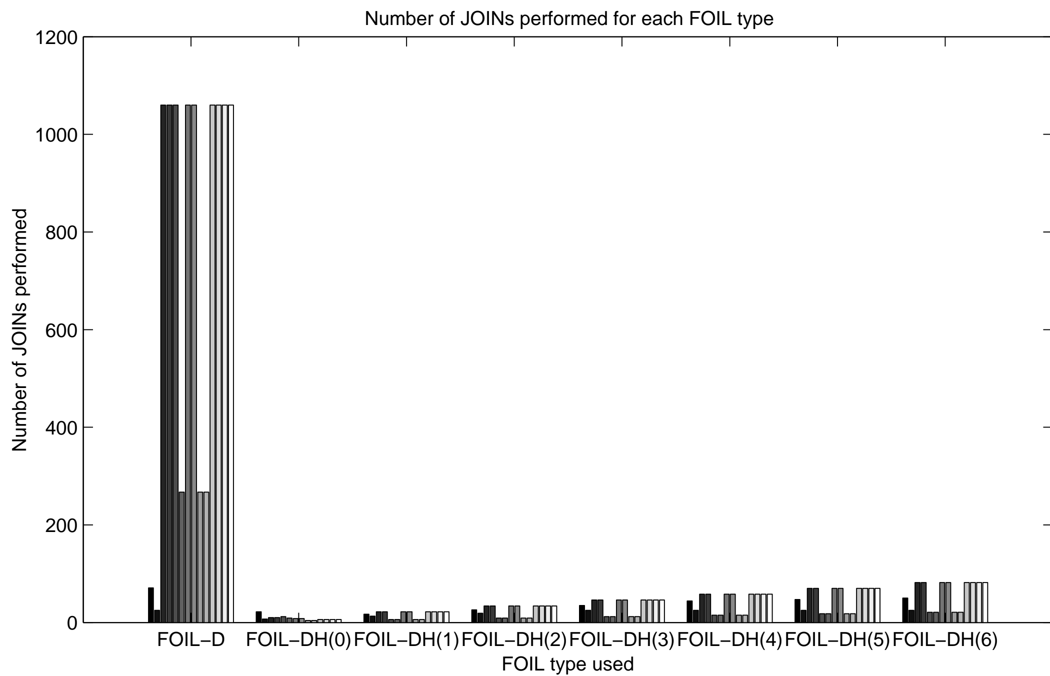

Fig. 5. Total number of JOINs performed for the different FOIL types when learning the following relations (in order from left to right in the histogram): can-reach, east-bound, mother, father, wife, husband, son, daughter, sister, brother, aunt, uncle, niece, nephew

The experiments we present in this paper all involve small datasets on standard but contrived problems. Our plans for the future involve evaluating FOIL-D and FOIL-DH in terms of accuracy and efficiency on a number of large real-world datasets. In addition, we plan on investigating more complex probability models, based on Bayesian networks.

In closing we note that although this paper has dealt exclusively with extending FOIL, some of the concepts developed here are applicable to other ILP systems. For example, systems that score whole clauses (as opposed to FOIL's scoring of literals) as a function of the number of positive and negative examples covered may benefit from similar estimation methods.

7 Acknowledgments

We gratefully acknowledge support for this research from U.S. Air Force grant F30602-01-2-0571 and NIH grant T15-LM07359-01. Thanks to David Page, Vítor Santos Costa and Inês Dutra who helped and encouraged us to present the ideas in this paper. Thanks also to Rich Maclin for help with graph formatting.

References

1. Lavrac, N., Dzeroski, S.: Inductive Logic Programming: Techniques and Applications. Ellis Horwood (1994)
2. Quinlan, J.R.: Learning logical definitions from relations. Machine Learning **5** (1990) 239–2666
3. Dimaio, F., Shavlik, J.: Speeding up relational data mining by learning to estimate candidate hypothesis scores. In: Proceedings of the ICDM Workshop on Foundations and New Directions of Data Mining. (2003)
4. Tang, L.R., Mooney, R.J., Melville, P.: Scaling up ilp to large examples: Results on link discovery for counter-terrorism. In: Proceedings of the KDD-2003 Workshop on Multi-Relational Data Mining, Washington, DC (2003) 107–121
5. Mooney, R.J., Melville, P., Tang, L.R., Shavlik, J., Dutra, I., Page, D., Santos Costa, V.: Relational data mining with inductive logic programming for link discovery. In Kargupta, H., Joshi, A., Sivakumar, K., Yesha, Y., eds.: Data Mining: Next Generation Challenges and Future Directions. Volume To appear. AAAI Press (2004)
6. Stonebraker, M., Kemnitz, G.: The postgres next-generation database management system. Communications of the ACM **34** (1991) 78–92
7. Brockhausen, P., Morik, K.: Directaccess of an ilp algorithm to a database management system. In: Proceedings of the MLnet Familiarization Workshop. (1996) 95–110
8. Ioannidis, Y.E., Poosala, V.: Histogram-based solutions to diverse database estimation problems. IEEE Data Eng. Bull. **18** (1995) 10–18
9. Ioannidis, Y.E., Poosala, V.: Balancing histogram optimality and practicality for query result size estimation. In: SIGMOD Conference. (1995) 233–244
10. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Proceedings of the European Conference on Machine Learning, Vienna, Austria (1993) 3–20

11. Ramakrishnan, R.: Database Management Systems. McGraw-Hill, New York (1998)
12. Pazzani, M., Kibler, D.: The utility of knowledge in inductive learning. Machine Learning **9** (1992) 57–94
13. Pearl, J.: Probabalistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Mateo, CA (1988)
14. Michalski, R.S., Mozetič, I., Hong, J., Lavrač, N.: The multipurpose incremental learning system AQ15 and its testing application to three medical domains. In: Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, Morgan Kaufmann (1986) 1041–1045
15. Hinton, G.E.: Learning distributed representations of concepts. In: Proceedings of the Eighth Annual Conference of the Fifth International Joint Conference on Artificial Intelligence, Amherst, MA, Lawrence Erlbaum (1986) 356–362