Efficient Theta-subsumption of Sets of Patterns

Jan Ramon and Jan Struyf
Katholieke Universiteit Leuven, Dept. of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{jan.ramon, jan.struyf}@cs.kuleuven.ac.be

Abstract

Frequent pattern discovery is an important data mining task. Recently, there was an increasing amount of interest in *relational* frequent pattern mining. In previous work we discussed how one can speed up the matching of relational patterns against the examples. In this paper we present new optimizations that speed up the pattern generation phase. These new optimizations are based on a representation of the patterns in a tree structure and exploit this structure in the various tests that are necessary during pattern generation. We discuss some theoretical properties and present a number of experiments.

1 Introduction

Frequent pattern discovery is an important data mining task. The killer application is basket analysis, where one is interested in sets of items that are frequently bought together in, for example, a supermarket. The APRIORI algorithm (Agrawal et al., 1996) can be used to mine such frequent item sets.

Recently, there was an increasing amount of interest in the more general setting of *relational* frequent pattern mining. Here, the goal is to mine relational patterns directly from a relational database.

A relational pattern can be represented in first order logic as a conjunction of first order literals. Consider the following example pattern[1].

person(X) $\wedge$ buys(X, pizza) $\wedge$ friend(X, Y) $\wedge$ buys(Y, wine).

The frequency of this pattern is equal to the proportion of persons X in the database that buy pizza and have a friend Y that buys wine. If this frequency is above a given threshold *minsup* (minimal support), the pattern is called a frequent pattern.

Relational pattern mining has been applied for natural language processing (Dehaspe and De Raedt, 1997), user modeling (Jacobs and Blockeel, 2001), biochemical applications (King et al., 2001) and for association rule mining in relational databases in general.

Relational frequent patterns can be mined from a database with an algorithm such as WARMR (Dehaspe and Toivonen, 1999). WARMR is an upgrade of APRIORI to the relational (first order) setting. Pattern mining in the relational setting is computationally much more demanding than in the propositional "item set" setting. This explains the high runtimes that are typical for WARMR (runs that take several days are no exception).

The two most time consuming steps of the WARMR algorithm (Section 2) are the candidate generation and the candidate evaluation step. Both steps can be made more efficient by exploiting the structure of the pattern space.

In (Blockeel et al., 2002) we show how the candidate evaluation step can be made more efficient. In this step the frequency of each candidate pattern is computed by executing it on the database. Due to the structure of the pattern space, several candidate patterns typically share a common prefix (e.g. literals $l_1 \wedge \ldots \wedge l_m$). By organizing the candidate patterns in a tree structure called a query-pack it is possible to speed up the evaluation step because computations can be shared for the common parts.

In this work we focus on the candidate generation step. This step encompasses a number of tests that can also be made more efficient by organizing the candidate patterns in a tree

[1] In accordance with Prolog notation, we represent variables with capital letters. We will assume the reader is familiar with basic logic programming concepts (Bratko, 1986).

structure. We present alternative algorithms for these tests that exploit this structure and perform a number of experiments that measure the obtainable speed-up.

This paper is organized as follows. In Section 2 we describe the WARMR algorithm and show how the patterns can be organized in a tree structure. In Section 3 we show how this tree structure can be exploited for improving the efficiency of a number of tests that are performed during the candidate generation step. Section 4 presents a number of experiments that measure the speed-up that can be obtained by using the more efficient tests. In Section 5 we state the conclusions.

2 WARMR

WARMR (Figure 1) is a level-wise algorithm for mining relational frequent patterns. WARMR takes as input a relational database D, a language root R, a refinement operator ρ and a parameter *minsup* and outputs all patterns in the language defined by R and ρ that have a frequency in D above *minsup*.

The language root R is the literal that introduces the key variable in the pattern. Continuing the example from the introduction, $R =$ person(X), with X the key variable. The key variable determines what WARMR will count. In the example, WARMR counts persons.

The refinement operator ρ is an operator that maps a pattern to a set of patterns. Each pattern in the set should be more specific than the original pattern. Recursively applying ρ starting from R should generate all patterns in the hypothesis language. A pattern language can be specified by the user by means of mode declarations. A mode declaration describes how a pattern can be extended by ρ to obtain a more specific pattern.

Consider the following example mode declarations.

`rmode(buys(+`X`, #)).`
`rmode(friend(+`X`,-`Y`)).`

The first mode describes that the buys/2 literal can be added to a pattern, the second one that friend/2 can be added. The '+' states that X should already occur in the pattern, the '-' indicates a new variable and the hash sign introduces a constant symbol.

algorithm WARMR(D,R,ρ,*minsup*)
1 $k = 0$; $C_0 = \{R\}$; $I = \emptyset$
2 **while** $C_k \neq \emptyset$
3 **for each** $c \in C_k$
4 count(c) = 0
5 **for each** $e \in D$, $c \in C_k$
6 **if** c succeeds for e **then**
7 count(c) = count(c) + 1
8 $F_k = \{c \in C_k \mid \text{count}(c) \geq minsup \cdot |D|\}$
9 $I = I \cup (C_k - F_k)$; $C_{k+1} = \emptyset$
10 **for each** $c \in F_k$, $c' \in \rho(c)$
11 **if not** $c \sim_\theta c'$
12 **or** $\exists c'' \in C_{k+1}$, $c'' \sim_\theta c'$
13 **or** $\exists i \in I$, $i \succeq_\theta c'$
14 **then** $C_{k+1} = C_{k+1} \cup \{c'\}$
15 $k = k + 1$
16 **return** $\bigcup F_k$

Figure 1: The WARMR algorithm.

Applying ρ to R yields the following.

$$\rho(\text{person}(X)) = \{\text{person}(X) \wedge \text{buys}(X, \text{pizza}),\ \text{person}(X) \wedge \text{buys}(X, \text{wine}),\ \text{person}(X) \wedge \text{friend}(X, Y)\}$$

More information about refinement operators can be found in (Nienhuys-Cheng and De Wolf, 1997).

WARMR starts by initializing the set of candidate patterns C_0 to the language root. Then it enters the main loop. Each iteration of this loop corresponds to a certain level k. Lines 3-7 implement the candidate evaluation phase. In this step, the frequency for each candidate pattern is computed by counting the number of instances in D (answer substitutions for the key variable) for which the pattern succeeds. Patterns that are found frequent are added to the set of frequent patterns F_k and infrequent patterns are added to the infrequent pattern set I. Lines 10-14 implement the candidate generation phase. Each refinement of a frequent pattern is a candidate pattern unless certain tests (lines 11-13) fail. After all candidates have been generated, the main loop starts the next level. The algorithm terminates if the candidate set is empty. The return value is the set of frequent patterns for each level.

The three tests in the pattern generation phase ensure that each pattern is generated only once and that patterns that can be proved to be infrequent are not generated.

The tests use the notion of θ-subsumption (Plotkin, 1970) and equivalence under θ-subsumption which are defined as follows.

Definition 1 (θ-subsumption) *Let g and s be two patterns, then $g \succeq_\theta s$ iff $\exists\theta : g\theta \subseteq s$.*

Definition 2 (Equivalence) *Let p_1 and p_2 be two patterns, then $p_1 \sim_\theta p_2$ iff $p_1 \succeq_\theta p_2$ and $p_2 \succeq_\theta p_1$.*

The first test checks if the new candidate is equivalent to its parent pattern. The second one checks if it is equivalent to another candidate ("equivalence test"). The last one checks if there is an infrequent pattern that is more general than the new one ("monotonicity test"). If this is the case then the new pattern will also be infrequent.

The operations $\succeq_\theta$ and $\sim_\theta$ have exponential time complexity in the length of the patterns (Kietz and Lübbe, 1994). This explains why a significant proportion of WARMR's execution time is spent on θ-subsumption checks, especially for small databases. In the next section we introduce efficient θ-subsumption algorithms for sets of patterns. These new algorithms can be used to make the tests in the candidate generation phase faster.

3 Efficient Candidate Validation

3.1 Data Representation

As said before, a refinement operator takes as input a pattern and outputs a set of more specific patterns. To construct each output pattern, the refinement operator adds one or more literals to the input pattern, which implies that the input pattern occurs as common prefix in all the output patterns. For example, person(X) occurs as common prefix in each pattern in ρ(person(X)). Because of this property, the candidate patterns can be organized in a tree structure.

Suppose we refine all the patterns in the set ρ(person(X)) a second time. In that case, the result can be organized in a tree structure with two levels, as shown in Figure 2. Each individual pattern corresponds to a path from the tree root to one of the leaves.

It is possible to define a refinement operator ρ^T that works directly on the tree representation. Given an input tree, ρ^T will refine each leaf l_i by adding a number of child nodes below it that correspond to all possible extensions of the pattern path(l_i). path(l_i) is the conjunction of the literals on the path from the root to l_i. The tree generated by $\rho^T(\rho^T(\text{person}(X)))$ is the one shown in Figure 2.

The tree representation has several advantages over a flat representation as a set of patterns. First, it is more compact and consumes less memory. Second, it induces a structure on the set of candidate patterns. And as we will see shortly, this structure allows for several optimizations.

What happens if WARMR generates the refinements from Figure 2? Patterns 1, 5 and 11 will be discarded immediately since they are equivalent to their parent. Next, WARMR will check the remaining patterns against previously generated candidates for equivalence. In this way, it will discover that pattern 4 is equivalent to pattern 2.

person(X) $\wedge$ buys(X, wine) $\wedge$ buys(X, pizza)
person(X) $\wedge$ buys(X, pizza) $\wedge$ buys(X, wine)

The only difference between patterns 4 and 2 is that the order of the literals is different. In a similar way, WARMR will discover that pattern 7 is equivalent to pattern 3 and that pattern 8 is equivalent to pattern 6. Finally, patterns 2, 3, 6, 9, 10 and 12 remain as candidates before the monotonicity test.

3.2 The Monotonicity Test

If some candidate pattern c' is generated and there is already an infrequent pattern $i \in I$ that is a generalization of c', then we can conclude that c' cannot be frequent either. One can check whether such an infrequent generalization exists by storing all infrequent patterns encountered so far and by checking them one by one against c' (line 13 in Figure 1).

Storing all infrequent patterns may be expensive. There is a tradeoff between the time spent in the monotonicity test during candidate generation and the time spent for counting during candidate evaluation. If the candidate evaluation phase is expensive because there are many instances in the database, the gain of storing infrequent patterns will be high because less candidates have to be evaluated (some are generalisations of patterns in I) and because the memory requirements for storing I will be low

			Hash
person(X) -+- buys(X, pizza)	-+- buys(X, pizza)	(1) $\sim_\theta$ parent	2041
|	+- buys(X, wine)	(2)	2582
|	+- friend(X, Y)	(3)	2796
+- buys(X, wine)	-+- buys(X, pizza)	(4) $\sim_\theta$ (2)	2582 = (2)
|	+- buys(X, wine)	(5) $\sim_\theta$ parent	1901
|	+- friend(X, Y)	(6)	2656
+- friend(X, Y)	-+- buys(X, pizza)	(7) $\sim_\theta$ (3)	2796 = (3)
	+- buys(X, wine)	(8) $\sim_\theta$ (6)	2656 = (6)
	+- buys(Y, pizza)	(9)	2796 = (3,7)
	+- buys(Y, wine)	(10)	2656 = (6,8)
	+- friend(X, Z)	(11) $\sim_\theta$ parent	1541
	+- friend(Y, Z)	(12)	1541 = (11)

Figure 2: The pattern tree $\rho^T(\rho^T(\text{person}(X)))$.

compared to the size of D. An intermediate approach is also possible: store infrequent patterns only up to a certain level.

Suppose that the following two patterns turned out to be infrequent (at level 1)

i_1: person(X) $\wedge$ buys(X, pizza)
i_2: person(X) $\wedge$ buys(X, wine)

and that WARMR considers the following candidate pattern c' (at level 2).

person(X) $\wedge$ buys(X, pizza) $\wedge$ friend(X,Y)

WARMR has to test if $i_1 \succeq_\theta c' \vee i_2 \succeq_\theta c'$. Consider first $i_1 \succeq_\theta c'$. This θ-subsumption test can be implemented as suggested by Definition 1. First all variables in c' are replaced by unique constants σ_i, then c' is converted to a set of facts and finally i_1 is "executed" on this set (e.g. with regular Prolog execution). The obtained answer substitution is equal to the substitution θ from Definition 1 after converting the so called skolem constants σ_i back to variables. If no answer substitution can be found, $i_1 \succeq_\theta c'$ does not hold. Continuing our example we obtain the following set of facts.

{person(σ_1), buys(σ_1, pizza), friend(σ_1,σ_2)}

The matching of i_1 on this set succeeds with answer substitution $\{X/\sigma_1\}$, so c' is a generalization of i_1 and c' will also be infrequent (it has an infrequent sub-pattern).

If we have to perform the subsumption test for a set of patterns ($i \in I$) it is possible to exploit the tree structure defined on this set. Each $i \in I$ has to be executed on the skolemized version of c'. If execution succeeds for one $i \in I$, c' is infrequent. The optimization we propose is to convert I to a tree shaped disjunction as follows.

person(X) $\wedge$ (buys(X, pizza) $\vee$
buys(X, wine))

Note that this is a subtree of the tree shown in Figure 2. Instead of executing the patterns one by one, it is possible to execute this disjunction at once on the skolemized version of c'. If one of the branches in the disjunction succeeds[2], c' is infrequent. In our example, the first branch succeeds so c' is infrequent.

Using such a disjunctive structure avoids that the common literals of the patterns (in this case person(X)) have to be executed several times. The higher the branching factor of the disjunction, the higher the speed-up obtained with this optimization will be. Also, the disjunction grows with one level after each level of the WARMR algorithm. Hence, the possible speed-up will also increase with each level.

3.3 The Equivalence Test

For the equivalence test, the regular approach is to test each new candidate for equivalence against all preceding candidates one by one. However, this too can be improved by exploiting the tree structure.

[2]This can be accomplished in Prolog by placing a cut after the call to the disjunction.

Testing equivalence involves a bidirectional θ-subsumption test (Definition 2). We will also consider the two directions separately in the optimized version that we propose. The first direction corresponds to finding the set of all generalisations of c', i.e., $G = \{c'' \in C_{k+1} \mid c'' \succeq_\theta c'\}$. The second direction corresponds to finding a pattern in G that is a specialization of c'. If such a pattern can be found, c' must be equivalent to this pattern and can be discarded.

We start with the problem of efficiently finding G. To construct G, each pattern in C_{k+1} must be executed on a skolemized version of c'. This situation is very similar to what was necessary for the efficient monotonicity test from Section 3.2. The only difference is that it was then sufficient to find *one* generalization, and now we need to compute the entire set of generalisations. We can accomplish this efficiently again by using the tree representation of C_{k+1}. In this case, each branch that succeeds is added to G. Again, because of the tree representation of C_{k+1}, literals can be shared between several patterns, which will result in a significant speedup.

If a certain branch of the disjunction has succeeded, we do not need backtracking over this branch for alternative solutions. Finding one answer substitution is sufficient. To avoid this unnecessary backtracking, branches have to be removed from the disjunction once they have succeeded. This type of execution mechanism is exactly the query-pack execution mechanism introduced in (Blockeel et al., 2002). To summarize, we can compute G efficiently by using the query-pack representation of C_{k+1} and by executing this query-pack on the skolemized version of c'. The set of succeeding paths in the pack is equal G.

The second direction of the equivalence test corresponds to finding a pattern in G that is a specialization of c'. This implies that c' must be executed on the skolemized versions of the patterns in G. If this succeeds for one $c'' \in G$, it holds that $c' \sim_\theta c''$ and c' can be discarded. This procedure can also be made more efficient by exploiting the tree structure of G.

The optimization we propose here is to represent the skolemized version of G as a tree structure and to mach the new candidate pattern c' along a path from G's root to one of its leaves. If such a path can be found, the pattern represented by this path is equivalent to the new candidate.

Matching a pattern against a tree can be implemented by iterating over the literals in the pattern while trying to match each literal in turn on the tree (allowing backtracking to previous literals if a certain match fails). If a literal matches with a certain node n of the tree, all sibling subtrees from the root to that node do not have to be considered anymore for matching subsequent literals and may be removed. The reason for this is that the literals in these subtrees are not on the same path as n is. For the interested reader, a pure Prolog implementation for this procedure is shown in Figure 3. In practice such a Prolog implementation has some overhead: the tree has to be reconstructed after matching each literal.

3.4 Using a Hashtable

The efficiency of the equivalence tests with previously generated candidates can be improved further by computing a specific hash value for each candidate pattern. The hash function must be constructed in such a way that the following implication holds: if two patterns have a different hash value then they can not be equivalent. A hash function that satisfies this property can be used to label most preceding candidates (those with a different hash value as the new candidate) as being not equivalent to the current candidate. For the remaining patterns (those with the same hash value), the more expensive bidirectional subsumption test should then be performed. The hash value for each pattern is shown on the right in Figure 2.

If one uses a hashtable, the labeling phase is not necessary. Each time a new valid candidate is generated it is added to the hashtable (using its hash value as key). To perform the set of equivalence tests for a given new candidate, one computes its hash value and retrieves the set of preceding candidates with equal hash key from the table. This step can be done in constant time. The bidirectional subsumption test has to be performed only between the new candidate and the candidates retrieved from the hashtable (typically only a few if the hash function works well).

The hash function can only use features that are invariant in an equivalence class under θ-

```
match_pattern_tree([], _Tree).
match_pattern_tree([Lit|R], Tree) :-
      match_literal(Tree, Lit, NTree),
      match_pattern_tree(R, NTree).

match_literal((A,B), Lit, NTree) :- !,
      (
            A = Lit, NTree = (A,B)
      ;
            NTree = (A, NB),
            match_literal(B, Lit, NB)
      ).
match_literal((A;B), Lit, NTree) :- !,
      match_literal_disj((A;B), Lit, NTree).
match_literal(A, A, A).

match_literal_disj((A;B), Lit, NTree) :- !,
      (
            match_literal(A, Lit, NTree)
      ;
            match_literal_disj(B, Lit, NTree)
      ).
match_literal_disj(A, Lit, NTree) :-
      match_literal(A, Lit, NTree).
```

Figure 3: Prolog code to test whether a pattern is a generalization of a pattern in a given tree. The pattern is represented as a list, the tree is represented with conjunctions and disjunctions.

subsumption. This includes the set of predicates and the set of constants occurring in the patterns. More specifically, one can collect all constant and predicate symbols from the pattern in a set (removing duplicates) and compute the hash value of the pattern as a combination of the hash values of the individual symbols. If object identity (Esposito et al., 2001) is used instead of regular θ-subsumption, other properties such as the length of the pattern can also be used.

To summarize, an efficient WARMR implementation can use a hashtable to identify possible equivalent patterns (Section 3.4) and after that apply the tree-based equivalence test (Section 3.3). The hash value can also be used to make the equivalence test with the parent pattern more efficient. If the new pattern is not equivalent to its parent, nor to any of the preceding candidates, the efficient WARMR implementation can use the tree-based monotonicity test (Section 3.2) to test if the new candidate is guaranteed to be infrequent or not.

4 Experiments

The aim of our experiments is to measure the efficiency gain that can be obtained by using our efficient subsumption algorithms for sets of patterns in the candidate generation phase of WARMR.

We have implemented the new subsumption tests in WARMR version 3. WARMR is integrated in the relational data mining system ACE[3] (Blockeel et al., 2002). The experiments are performed on the Mutagenesis data set (Srinivasan et al., 1996), which stores the structural description of 230 molecules.

Table 1 presents the results. We have performed WARMR runs with different subsets of the proposed optimizations enabled. Each row of the table corresponds to such a subset. The columns correspond to runs with different values of the *maxdepth* setting (if *maxdepth* = 2, WARMR stops after completing level 2). T_e is the time spent for candidate evaluation, T_g the time for candidate generation (including the various subsumption tests). The first row is the most efficient WARMR version: all the optimizations are enabled (*mon* = the efficient monotonicity test, *eq* = equivalence test that exploits the tree structure and H = hashtables are used for the equivalence test). The last row the least efficient one: all tests are performed one pattern at a time and no hashing is used.

A first thing that can be noted is that the candidate evaluation phase takes much less time for this application than the candidate generation phase. The reason for this is that the number of instances is rather low (230) and that the generated patterns are rather general because the minimal frequency threshold *minsup* was set at 10% (general patterns execute faster).

The time spent on the subsumption tests increases with *maxdepth* because more candid-

[3]ACE is available for academic use; see: `http://www.cs.kuleuven.ac.be/~dtai/ACE/`.

Optimizations	$k=2$		$k=4$		$k=6$		$k=8$		$k=10$		$k=12$	
	T_e	T_g	T_e	T_g	T_e	T_g	T_e	T_g	T_e	T_g	T_e	T_g
H,eq,mon	0.1	0.2	0.5	16	0.7	63	0.9	120	0.9	160	1.0	170
H,mon	0.1	0.2	0.5	21	0.7	73	0.9	140	0.9	180	0.9	190
H,eq	0.1	0.2	0.5	17	0.7	74	0.9	160	0.9	210	1.0	230
eq,mon	0.1	1.9	0.5	110	0.7	260	0.8	390	0.9	450	0.9	470
mon	0.1	2.5	0.5	250	0.7	540	0.8	770	0.9	840	1.0	850
	0.1	2.5	0.5	250	0.7	550	0.9	810	0.9	900	1.0	920

Table 1: Timings (in seconds) of WARMR for different subsets of the proposed optimizations. For each subset, the time spent during the candidate evaluation T_e and generation T_g is reported for different values of the "*maxdepth*" setting.

ates are generated at higher levels and because the subsumption tests have exponentional time complexity in the pattern length.

The speed-up factor of the fully optimized WARMR versus WARMR with no optimizations is 5.4, which is significant. If we would decrease the *minsup* parameter, more patterns would be generated, which may result in even higher gains. Further experiments are necessary to confirm this.

5 Conclusions

Relational frequent pattern mining is an important data mining task. A significant proportion of the execution time of a relation pattern mining algorithm such as WARMR is spent on a number of θ-subsumption tests that occur during the pattern generation phase.

In this work we have proposed a number of optimizations for these subsumption tests. Our optimizations are designed for subsumption tests between a single pattern and a set of patterns and exploit tree structure defined on this set. The optimizations can be used in WARMR in the test for equivalence to a preceding candidate and in the monotonicity test. The equivalence test can be accelerated even more by adding the candidate patterns to a hashtable.

Our experiments show that a significant speed-up can be obtained by using the proposed optimization techniques.

In further work, we would like to investigate what the effect of the *minsup* parameter is on the performance of our optimizations. Lower values for *minsup* would result in more patterns being generated, which may result in higher efficiency gains.

We would also like to investigate other optimizations for the subsumption tests. An example of an extra optimization is to exploit the order of the modes in the language. Such an optimization may prevent many equivalent candidate patterns from being generated.

Another possibility is to try to combine other optimizations proposed in the literature for testing subsumption between two patterns, such as (Kietz and Lübbe, 1994; Santos Costa et al., 2003; Struyf and Blockeel, 2003), with our approach. This is not trivial since these may have a negative influence (because they reorder literals) on the tree structure that our optimizations exploit.

Finally we are planning to compare our efficient WARMR implementation with other fast relational pattern mining algorithms, such as the one proposed by Nijssen and Kok (2001).

Acknowledgments

Jan Ramon is a post-doctoral fellow and Jan Struyf a research assistant of the Fund for Scientific Research (FWO) of Flanders. The authors would like to thank Bart Demoen, Gerda Janssens and Siegfried Nijssen for the fruitful discussions.

References

R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. 1996. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. The MIT Press.

H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. 2002. Improving the efficiency of inductive logic programming through the use of query packs.

Journal of Artificial Intelligence Research, 16:135–166.

I. Bratko. 1986. *Prolog Programming for Artificial Intelligence*. Addison-Wesley.

L. Dehaspe and L. De Raedt. 1997. Mining association rules in multiple relations. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 125–132, Berlin. Springer-Verlag.

L. Dehaspe and H. Toivonen. 1999. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36.

F. Esposito, N. Fanizzi, S. Ferilli, and G. Semeraro. 2001. A generalization model based on oi-implication for ideal theory refinement. *Fundamenta Informaticae*, 47(1-2):15–33.

N. Jacobs and H. Blockeel. 2001. From shell logs to shell scripts. In *Proceedings of ILP2001 - Eleventh International Workshop on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 80–90.

J.U. Kietz and M. Lübbe. 1994. An efficient subsumption algorithm for inductive logic programming. In *Proceedings of the 11th International Conference on Machine Learning*, pages 130–138. Morgan Kaufmann.

R. D. King, A. Srinivasan, and L. Dehaspe. 2001. Warmr: a data mining tool for chemical data. *Journal of Computer-Aided Molecular Design*, 15(2):173–181, feb.

S.-H. Nienhuys-Cheng and R. De Wolf. 1997. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence*. Springer-Verlag, New York, NY, USA.

S. Nijssen and J. N. Kok. 2001. Faster association rules for multiple relations. In *IJCAI01*, pages 891–896.

G. Plotkin. 1970. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press.

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

A. Srinivasan, S.H. Muggleton, M.J.E. Sternberg, and R.D. King. 1996. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2):277–299.

J. Struyf and H. Blockeel. 2003. Query optimization in inductive logic programming by reordering literals. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 329–346. Springer-Verlag.