Faster Association Rules for Multiple Relations

Siegfried Nijssen and **Joost Kok**
snijssen@liacs.nl and joost@liacs.nl
Leiden Institute of Advanced Computer Science, Leiden University
Niels Bohrweg 1, 2333CA Leiden, The Netherlands

Abstract

Several algorithms have already been implemented which combine association rules with first order logic formulas. Although this resulted in several usable algorithms, little attention was payed until recently to the efficiency of these algorithms. In this paper we present some new ideas to turn one important intermediate step in the process of discovering such rules, i.e. the discovery of frequent item sets, more efficient. Using an implementation that we coined FARMER, we show that indeed a speed-up is obtained and that, using these ideas, the performance is much more comparable to original association rule algorithms.

1 Introduction

The formalism of association rules was introduced by Agrawal [1996] for the purpose of basket analysis. An important step in the discovery of such rules is the construction of frequent item sets. These are, for instance, sets of items that are frequently bought together in one supermarket transaction. As this discovery step is time critical, it is obligatory that it is performed reasonably fast. Much research has been done in order to develop efficient algorithms. A well-known algorithm resulting from this research is APRIORI, of which many variants have been developed, such as APRIORI-TID [Agrawal *et al.*, 1996] and a breadth-first algorithm introduced by Pijls and Bioch [1999].

On the other hand, efforts have been done to extend the usability of association rules beyond the basic case of basket analysis. Dehaspe and De Raedt [1997] use the notion of atom sets as a first order logic extension of item sets. The incorporation of techniques from Inductive Logic Programming allows for more complex rules to be found which also take into account background knowledge. Consequently, this also allows data mining of data which is spread over tables which can not reasonably be merged into one table. An algorithm was implemented based on this notion, which was called WARMR. The usefulness of this algorithm was demonstrated in several real-world situations (see, for example, [Dehaspe *et al.*, 1998]). These experiments, however, also showed the major shortcoming of the algorithm: its efficiency proved to be very low, some experiments even taking several days.

We propose to obtain a gain in efficiency by tackling two properties of the WARMR algorithm:

- while still using the first order logic notation, we remove the need for PROLOG;
- by using a more sophisticated datastructure borrowed from an implementation of APRIORI, our algorithm does not depend on a time consuming test for equivalence.

The algorithm that we introduce has some ressemblance with the algorithm that was developped in [Blockeel *et al.*, 2000]. That algorithm however did not tackle one of the most time consuming steps of WARMR: a test for equivalence under θ-subsumption. Our algorithm pays special attention to this step and offers an alternative solution. Under some restrictions we will show that our algorithm is equivalent to WARMR. Experiments with our algorithm then show a considerable speed-up compared to WARMR.

The paper is organized as follows. In the second section we summarize the association rule algorithm on which our work is based. In the third section we discuss some important notions introduced in the WARMR algorithm. The fourth section introduces our modifications, which are verified by giving results of experiments in the fifth section. The sixth section concludes.

2 Breadth-first APRIORI

Our algorithm is based on a variation of APRIORI that was introduced by Pijls and Bioch [1999]. The algorithm performs the same task as APRIORI. Given a database D which contains subsets T of a set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ the algorithm discovers all *frequent item sets*, which are the subsets $I \subseteq \mathcal{I}$ for which $support(I) = |\{T \in D \,|\, I \subseteq T\}|$ exceeds a predefined threshold. An item set of size k is called a $k-$item set. An important property of $support(I)$ is:

$$I_1 \subseteq I_2 \Rightarrow support(I_1) \geq support(I_2), \quad (1)$$

as every subset I_1 of I_2 occurs in every transaction that contains I_2. This property turns an efficient bottom-up levelwise search possible: it can *apriori* be determined that a $k+1$-item set is not frequent if a k-subset is infrequent.

The breadth first-algorithm is such a bottom-up levelwise algorithm. It starts with candidates of size one, after which a process is repeated of counting $k-$candidate item sets and of using them to obtain candidate $k+1$ item sets. All these

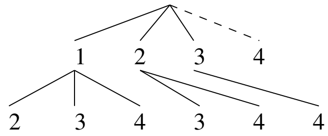


Figure 1: A small example of a trie datastructure

steps are performed on a *trie* datastructure, of which Figure 1 displays an example. Every path from the root to a node corresponds to an item set; all leafs at the deepest level correspond to candidate item sets. Paths which do not reach until the deepest level are maintained only when they correspond to frequent item sets and are displayed by dotted lines for the sake of clarity. The trie is used in the following fashion:

- In the step of candidate counting, a tree traversal is performed for every transaction, as follows: if an item occurs in a transaction, all its children are checked recursively. If a leaf is reached, the support count of the corresponding item set is increased.
- In the step of candidate generation, for every frequent item set new children are generated, consisting of all frequent right brothers. In the example $\{2\}$ is expanded by its frequent right brothers 3 and 4. This copying mechanism in combination with the order of the items takes care of generating every item set at most once.

In both steps, this mechanism distinguishes itself from the original APRIORI algorithm. Instead of building a new tree for each round, this procedure efficiently constructs a new set of candidates by merely copying nodes. Furthermore, during the counting phase, it passes through the tree and checks for the existence of candidates in the current transaction. This in contrast to the original algorithm, where for a given subset in the transaction, a search in a hash node is performed to check whether there is a candidate to be counted. It will appear that these both characteristics make this variant of APRIORI suitable for our purposes.

3 WARMR

As a first order extension of item sets, Dehaspe and De Raedt [1997] use sets of atoms, which they also refer to as *queries* when nearly all variables are existentially quantified and the set is ordered. The free variables are bound by a special purpose *key* predicate. The relation of the key and the query is illustrated in the following Horn clause:

$$\underbrace{k(X)}_{\text{key}} \leftarrow \underbrace{buys(X, Y, cardbonus), property(X, loyal)}_{\text{query}} \quad (2)$$

In this example the predicate $buys$ can be thought of as a table which describes the products that clients are buying, while the $property$ predicate refers to a table containing properties of clients. In the sequel we will use abbreviations such as b for $buys$, p for $property$ and c for $cardbonus$. This example shows how several tables can be combined more elegantly in a query than in an item set.

The support of the query is formalized using the key and is defined to be the number of variable bindings for which the key predicate can be proved. In the given example the support of $\{b(X, Y, c), p(X, l)\}$ is the number of variable bindings of X for which $k(X)$ can be proved given the Horn clause in Formula (2) and a knowledge base defined in PROLOG.

While for item sets the definition of the search space is straightforward, this is not the case for atom sets. Apart from the choice of predicate, there are also many possibilities for the usage of variables in the query. To define the *bias* of the search space WARMR uses a refinement operator based on *mode declarations*. Every mode declaration prescribes the way in which a predicate can be added to a query. The following is an example of a mode:

$$b(+, -, c). \quad (3)$$

It states that predicate b may be added to a query when the first parameter is bound to an existing variable, the second parameter introduces a new variable and the last parameter is bound to the constant c. The parameters are called *mode constraints*; here, we will call $+$ parameters and constant parameters *input parameters* and $-$ parameters *output parameters*. Often an integer is associated with every mode to indicate how many times at most the mode may be applied in the same query.

The usage of atoms instead of items turns it more difficult to create an efficient APRIORI-like algorithm: it is no longer reasonable to use the subset relation to express relations between atom sets. As replacement for the subset relation, and as approximation of logical implication, WARMR uses $\theta-$*subsumption*. An atom set C subsumes an atom set D, denoted by $C \succeq D$, if there is a substitution θ such that $C\theta \subseteq D$. The θ-subsumption relation induces an equivalence relation $\sim$, that is defined as follows: $C \sim D$ iff $C \succeq D$ and $D \succeq C$. It can be shown that a property similar to Formula (1) also holds for θ-subsumption on atom sets:

$$I_1 \succeq I_2 \Rightarrow support(I_1) \geq support(I_2). \quad (4)$$

For a set of frequent queries of size k (denoted by L_k) and a set of infrequent queries of size k (denoted by I_k), WARMR uses this algorithm to generate a new set of candidate queries C_{k+1}:

warmr-gen
$C_{k+1} = \emptyset$;
for all $c \in L_k$ **do**
  for all refinements c' of c **do**
    Add c' to C_{k+1} unless:
      (a) there is a $e \in \bigcup_{i \leq k} I_i : e \succeq c'$, or
      (b) there is a $e \in \bigcup_{i \leq k} L_i \cup C_{k+1} : e \sim c'$.

Restriction (a) removes queries which are *apriori* determined to be infrequent. Restriction (b) removes queries which have the same meaning as previously considered frequent queries or candidates. We will illustrate this on a small example. Consider the following set of mode declarations:

$$\{b(+, juice, -), t(+, c), t(+, electronicpurse)\}$$

This may lead to these two queries:

$$b(X, j, Y_1), t(Y_1, c), b(X, j, Y_2), t(Y_2, e)$$
$$b(X, j, Y_1), t(Y_1, e), b(X, j, Y_2), t(Y_2, c)$$

These clauses however have the same meaning and logically imply eachother.

A major problems of WARMR is that it heavily depends on a good implementation of $\theta-$subsumption. This is prohibitive as θ-subsumption is an NP-complete problem [Kietz and Lübbe, 1994].

4 FARMER

We propose two modifications of WARMR in order to make this algorithm more efficient. Each of the following two subsections will discuss one of them.

4.1 Knowledge base

When taking a closer look at the mode declarations, it can easily be seen that they can be mapped to procedures. As an example, consider the following facts: $b(1, ananas, c)$, $b(1, juice, e)$. Given mode $b(+, \#, +)$, this mode could be associated with a procedure which returns *true* for $(1, j, e)$ and returns *false* for $(1, a, c)$. Furthermore, a mode $b(+, -, -)$ could be associated with a procedure which for input 1 returns $\{(a, c), (j, e)\}$. In FARMER this idea is incorporated by binding all mode declarations to a procedure of one of these two types:

- boolean procedures, which for an input vector return true or false;
- outputting procedures, which for an input vector return a set of output vectors. Of course, this set may be empty and need not be computed entirely before all elements are used.

The first kind of procedures should be used in modes which do not have output parameters. The second kind is used in outputting modes.

A data structure for a knowledge base of PROLOG facts is created and accessed by procedures, as follows: for every mode declaration a multidimensional matrix is allocated; every element in the matrix corresponds to a set of input values and contains a truth value or a list of output values. When a fact for a predicate is read, all corresponding mode matrices are updated accordingly. The advantage of this mechanism is that it takes a constant amount of time to determine the truth of an atom, especially in our current implementation which stores the matrices in core memory[1]. Knowledge can however only be specified using facts or ad-hoc procedures.

4.2 Search

The search which FARMER performs differs in two aspects from the original WARMR algorithm:

- it does not use θ-subsumption;
- it manipulates a trie datastructure to generate the queries which are defined by the bias.

The trie datastructure is a tree which contains all candidate queries as a path from the root to a leaf. An example of such a trie is given in Figure 2. The tree is used for both counting and generating candidates.

[1]For large datasets, this could be a disadvantage. However, as our algorithm fits within the learning from interpretations approach, similar arguments hold when part of a database is on disk.

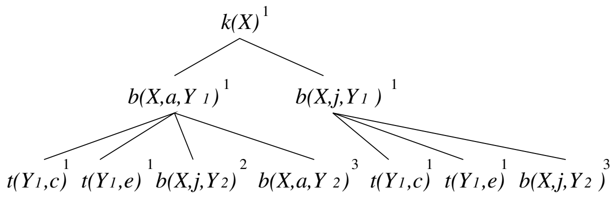


Figure 2: A trie in FARMER

Counting

We will describe the algorithm by giving pseudo-code. In this pseudo-code, we use the following notation:

- a capital A refers to an atom in the tree;
- a capital B refers to a set of variable assignments, which is a set containing a mapping from variables to values;
- $I(A, B)$ is an interpretation function, which returns true if an atom A can be proved using assignment B. In this function, one of the aforementioned procedures is used;
- $\mathcal{B}(A, B)$ is an assignment function. The assignment set B gives a value to some variables occuring in A. For the remaining unbound variables, this function returns all sets of possible assignments. The aforementioned outputting procedure is used here.

For all values of the key variables, the tree is traversed recursively, as follows:

```
Count(A,B)
if I(A,B) = true then
  if A is a leaf then
    disable A
    increase support A
    return true
  else
    for all B' ∈ 𝓑(A,B) do
      d := true
      for all A' ∈ children do
        if A' not disabled then
          d := Count(A',B ∪ B') ∧d
      if d = true then
        disable A
        return true
return false
```

Initially all nodes are enabled. For a given node all values for the outputs are checked as long as there are children which have not been satisfied. A similar idea is also applied in [Blockeel *et al.*, 2000]. We show the integration of the procedures and mode declarations here.

Candidate generation algorithm

We will first introduce the mechanism of candidate generation by giving the algorithm. Afterwards we will compare this generation mechanism with the θ-subsumption based method of WARMR.

The generation mechanism is based on the following idea: when an atom with an input variable is moved to the beginning of a query, that variable could become an output variable, hereby violating the mode declarations. However, for every atom in a query there is one first position at which it can occur without violation. All atoms that can be added to the end of a query can be subdivided in the following three classes:

1. atoms that could not have been added at an earlier position, as they use at least one new variable of the last atom in the query; we call these *dependent atoms*;
3. atoms that are a copy of the last atom in the query, except for the names of the output variables;
2. other atoms that could have been added at an earlier position.

Examples of these classes are given by the superscripts in Figure 2.

During the construction of the tree this subdivision is used. Given a trie and an ordered set of mode declarations, the trie is expanded as follows:

Expand(A)
if A is internal **then**
  for all $A' \in$ children **do**
    Expand (A')
else if A is frequent **then**
  add as child from left to right:
    1. all dependent atoms of A
    2. all frequent right brothers of A
    3. a copy of A with new output variables, if allowed
else
  remove A

The tree in Figure 2 is obtained using this mechanism when it is assumed that all queries of the following (typed) bias are frequent:

$$\{b(+A, a, -B), p(+A, j, -B), t(+B, c), t(+B, e)\} \quad (5)$$

The superscripts also in this case denote the mechanism that was used to create a node.

The first mechanism serves the purpose of introducing atoms which could not be added previously. The atoms are introduced in the same order as the corresponding mode declarations and a deterministic mechanism is used to go through all the input variables.

The dependent atoms are brothers of eachother; the second mechanism takes care that all subsets are generated afterwards – if not infrequent. By keeping the children in order, every subset is generated only once, or, equivalently, only one permutation out of a set of dependent atoms is considered. If necessary, the second mechanism gives new names to output variables to make sure they remain outputs.

The third mechanism is intentionally separated from the other two. Generation of repeating nodes is not desirable in many situations and should in any case be bound to a maximum. In our settings, the bias should explicitely state whether duplication of an atom is allowed.

Atoms of the third kind do not fit very well in the distinction that was introduced. A repeating node could in any case be exchanged with its parent. It would however not be efficient to introduce a set of identical nodes to overcome this problem. Later on, we will also see some additional disadvantages of these atoms.

Candidate generation discussion

Due to the absence of θ-subsumption, it can easily be seen that FARMER does not prune as many queries as WARMR does. In this section we will show which restrictions should be applied to the bias in order to make sure that FARMER will generate the same output.

In WARMR θ-subsumption is used for two purposes:

- to prune infrequent queries before counting;
- to remove queries which "mean the same" as other queries.

Only the θ-subsumption relation that is used for the latter purpose will be considered here, as only this relation influences the set of queries that is found. Infrequent queries will not occur in the results even if they are not pruned.

The θ-subsumption equivalence relation is only one method for determining that queries mean the same. A less strict relation is the equality relation under substitution, which we will denote with $\simeq$ here and is defined for two (unordered) sets of atoms as follows: $C \simeq D$ iff there exist substitutions θ_1 and θ_2 such that $C\theta_1 = D$ and $D\theta_2 = C$. The correspondence between these relations can be expressed using Plotkin's reduced clauses.

Definition 4.1 *A clause D is called reduced iff $C \subseteq D$ and $C \sim D$ imply $C = D$. [Plotkin, 1969]*

Theorem 4.1 *Let C and D be reduced sets of atoms. Then $C \simeq D$ iff $C \sim D$.*

Proof "$\Rightarrow$": this is clear as $C\theta_1 \subseteq D$ and $D\theta_2 \subseteq C$.
"$\Leftarrow$": as $C \succeq D$, $C\theta_1 \subseteq D$, and as $D \succeq C$, $C\theta_1\theta_2 \subseteq C$. Let $C' = C\theta_1\theta_2$. Because $C' \subseteq C$ and $C\theta_1\theta_2 \subseteq C'$, also $C \sim C'$ holds (by definition), and then $C = C'$ because C is reduced. Thus $C\theta_1\theta_2 = C$. In the same way, $D\theta'_1\theta'_2 = D$. As $|C\theta_1| = |C|$ and $C\theta_1 \subseteq D$, $|D| \geq |C|$. As $|D\theta'_1| = |D|$, $|C| \geq |D|$, and finally $|D| = |C|$. By combining $|C\theta_1| = |D|$ and $C\theta_1 \subseteq D$, $C\theta_1 = D$ is shown. This proves that $D \simeq C$. □

From this theorem it also follows that if atom sets are reduced, they can never be subsumption equivalent when they differ in length.

We will show that for a restricted bias, FARMER will always generate reduced atom sets. Then we will show that FARMER does not generate two different atom sets that are substitution equivalent. From this we conclude that, given a restricted bias, FARMER will not generate queries that subsume eachother.

Definition 4.2 *A redundancy restricted bias should obey the following rules:*

1. *no functions may be used;*
2. *repetition of an atom by an attom which differs only in the name of the output variables is not allowed;*
3. *no two modes for the same predicate may exist for which the constraint parameters differ, unless the corresponding parameters are both constant parameters.*

The second rule prevents queries such as $b(A, a, B_1)$, $b(A, a, B_2)$ from being generated. The third rule disallows the biases $(b(+, a, -), b(+, -, -))$ and $(b(+, a, -),$

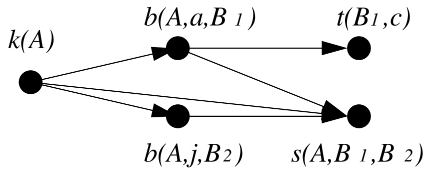


Figure 3: A partial order for a query

$b(+, a, +), t(+, -))$, and consequently queries $(b(A, a, B_1), b(A, C_1, B_2))$ and $(b(A, a, B_1), t(B_1, B_2), b(A, a, B_2)))$. Query $b(A, a, B_1), t(B_1, c), b(A, j, B_2)$ remains possible.

Theorem 4.2 *For a redundancy restricted bias* FARMER *will always generate reduced atom sets.*

Proof Assume that D is a query in the trie, obtained by using a redundancy restricted bias. We will show that for any subset $C \subset D$, $D\theta = C$ can never be true. In order for this, there must at least be two atoms A_1 and A_2 in D which are mapped to the same atom in C: $A_1\theta = A_2\theta$. For any pair we will try to construct such a substitution. By definition of the bias, both atoms must have inputs at the same positions (restriction 2), while the input variables must be different (restriction 1 in combination with the tree building procedure, where such atoms could only be generated as brothers). Construct a substitution which unifies A_1 and A_2. This substitution will always map variables to variables, as no functions are allowed and no modes with constants and variables at the same parameters. Apply θ to the whole query. Consider the set of atoms that introduced the variables used in A_1 and A_2, then there are two possibilities:

1. This set contains one atom which has two outputting parameters. By θ these are bound to the same variable. Such an atom can never be generated according to the mode mechanism used by FARMER;
2. This set has at least two different atoms. Of both atoms an output is bound to the same variable by θ. In whatever order these atoms are placed, one of them has now an input at a position where an output occured. This would require another mode, which is not allowed in this restricted bias.

Thus there can not exist redundant queries. □

Theorem 4.3 *Given a redundancy restricted bias,* FARMER *will never generate two queries that are substitution equivalent.*

Proof We first remark that for ordered atom sets, such as queries, a deterministic variable numbering can be used. Furthermore we note that two queries must be of equal size and that the substition can only map from variables to variables. Thus, to determine whether two queries substitution equal eachother, it suffices to find a permutation of atoms, followed by a variable renumbering, that makes two queries equal. We will show that FARMER generates one permutation.

The restricted bias is such that for every atom in an (unordered) atom set, there is only one possible mode declaration. The usage of input and output parameters determines a partial order on the atoms, which can be depicted in a graph such as in Figure 3 for the atom set $\{b(A, a, B_1), b(A, j, B_2), t(B_1, c), s(A, B_1, B_2)\}$ and the bias $(b(+A, a, -B), b(+A, j, -B), t(+B, c), s(+A, +B, +B))$. Use this strategy to order the nodes in a query Q:

> **order(A)**
> add A to the end of Q
> S := nodes with incoming arrow from A and
>   no incoming arrow from outside Q
> order S according to mode declarations and
>   a deterministic input variable numbering strategy
> **for all** $A' \in S$ in order **do**
>   order(A')

The order obtained by this strategy corresponds to the order of FARMER: the set S corresponds to the set of dependent nodes; the tree building mechanism which places new nodes before copied nodes takes care of the recursion by acting as a sort of LIFO queue. □

Corollary 4.1 *Given a redundancy restricted bias* FARMER *will never generate queries that θ-subsume eachother.*

5 Experimental results

We have compared FARMER and WARMR on two datasets. We should remark that in our experiments we used an implementation of WARMR that did not yet use the tree datastructure discussed in [Blockeel *et al.*, 2000]; a comparison of both algorithms is therefore not completely fair. Experiments in [Blockeel *et al.*, 2000] revealed speed-ups of 20 for WARMR in some situations.

Bongard

The Bongard dataset [Bongard, 1970] contains descriptions of artificial images. The task is to discover patterns in the images. Every image consists of several figures that can be included into eachother. No redundancy restricted bias can be used.

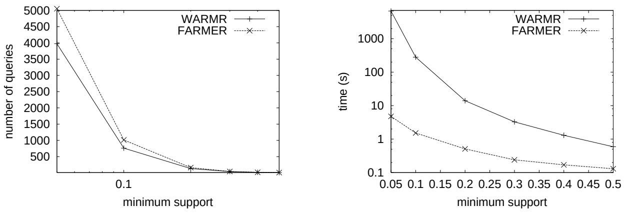
In Figure 4 the results of the experiments are depicted. Figure 4(a) shows the number of queries that each algorithm finds. The number of FARMER is higher in all cases, which is also expected for this bias. In Figure 4(b) the execution times of the algorithms are compared[2]. Paying attention to the fact that the scale is logarithmic, the speed-ups are considerable for this dataset.

Frequent itemsets

In this experiment we compare the performance of FARMER to a special purpose algorithm. As test case a binary digit dataset is used which contains 1000 binary coded numbers. The special purpose algorithm which is used as comparison is the breadth-first implementation of APRIORI by [Pijls and Bioch, 1999]. FARMER uses many of the mechanisms introduced in that algorithm and should perform comparable to that algorithm.

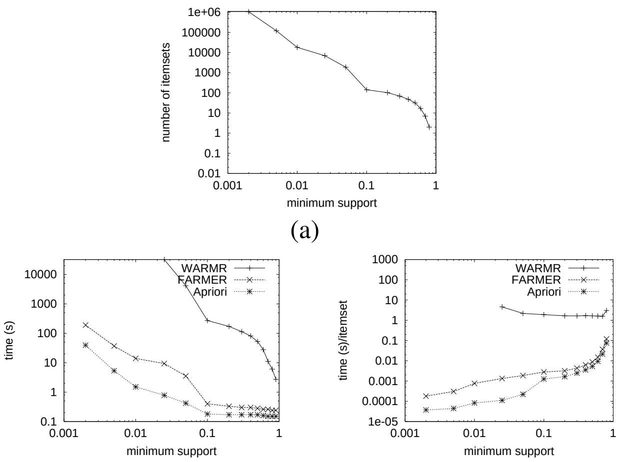
In Figure 5 the results of the experiments are depicted. A characteristic of the dataset is given in Figure 5(a). The number of frequent itemsets appears to increase rapidly when the

[2]Experiments were carried out on a Sun Enterprise 450 4x400MHz UltraSPARC2 CPU w/4MB E-cache 4GB RAM.



(a) (b)

Figure 4: A comparison of FARMER and WARMR on the Bongard dataset. (a) The number of queries in the output. (b) Execution times in seconds. Note that the scales are different.



(a)

(b) (c)

Figure 5: Comparison of results for the digit dataset. (a) The number of frequent itemsets for each minimum support. (b) The execution times of the algorithms. (c) The execution times consumed for each itemset. Note that the scale is logarithmic on both axis.

minimum support is beneath 0.1. In Figure 5(b) the execution times of the algorithms are given. The time graph of FARMER is comparable to that of APRIORI, although still exponentially larger. WARMR has a completely different behaviour than both other algorithms and had such high execution times that no experiments were carried out for low supports.

In Figure 5(c) both previous graphs are combined and the execution time for each itemset is shown. It makes clear how the algorithms react when the amount of solutions they have to find increases. While the execution times of WARMR increase, the times of the other algorithms decrease. Although the overhead for each itemset is larger in FARMER, which could be explained by the additional mechanisms that are hooked in, the difference is acceptable. The decreasing trend can be explained by the increasing number of overlapping evaluations when the number of itemsets increases.

6 Conclusions and further work

We introduced an efficient algorithm for discovering queries. It uses a tree datastructure both to count queries as to generate queries. We showed that for a restricted type of bias, this algorithm is equivalent to a previous algorithm, WARMR, and performs much better.

Although we believe that our restricted bias already adds considerable expressive power to propositional association rules, we are looking at some possibilities to overcome these restrictions. It appears that in case the second restriction is lifted, the range of possible rules already increases considerably. We are investigating the possibility of using the order of the tree in combination with a more sophisticated default order of queries.

Furthermore, we plan to perform more experiments. We successfully performed some experiments on a database with one million records, but more experiments are necessary to find out the behaviour of FARMER on datasets of this size.

References

[Agrawal *et al.*, 1996] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining.*, pages 307–328. AAAI/MIT Press, 1996.

[Blockeel *et al.*, 2000] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ilp proceedings of ilp2000 - 10th international conference on inductive logic programming, 2000.

[Bongard, 1970] M. Bongard. *Pattern Recognition*. Hayden Book Company (Spartan Books), 1970.

[Dehaspe and De Raedt, 1997] L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 125–132. Springer-Verlag, 1997.

[Dehaspe *et al.*, 1998] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press., 1998.

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

[Nijssen, 2000] S. Nijssen. Data mining using logic, master's thesis, Leiden University, 2000.

[Pijls and Bioch, 1999] W. Pijls and J. C. Bioch. Mining frequent itemsets in memory-resident databases. In E. Postma, editor, *Proceedings Eleventh Belgium/Netherlands Artificial Intelligence Conference*, pages 75–82, 1999.

[Plotkin, 1969] G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, Edinburgh, 1969. Edinburgh University Press.