Efficient Frequent Query Discovery in FARMER

Siegfried Nijssen and Joost N. Kok

Leiden Institute of Advanced Computer Science
Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands
snijssen@liacs.nl

Abstract. The upgrade of frequent item set mining to a setup with multiple relations – frequent query mining – poses many efficiency problems. Taking Object Identity as starting point, we present several optimization techniques for frequent query mining algorithms. The resulting algorithm has a better performance than a previous ILP algorithm and competes with more specialized graph mining algorithms in performance.

1 Introduction

Recently, multi-relational or structured data mining has gained much interest. Especially frequent structure mining similar to APRIORI [1] was discussed in a number of recent publications, such as the gSpan algorithm by Yan et al. [10] and FSG by Kuramochi et al. [7]. Given a database of complex structures —in the case of gSpan and FSG a collection of graphs— the task of these algorithms is to find those substructures that occur in many of the complex structures. Already several years ago, Dehaspe et al. [3] introduced an algorithm called WARMR for frequent pattern mining in relational databases. WARMR was built on the solid theoretical foundations of Inductive Logic Programming (ILP). It accomplished similar tasks as the more recent algorithms. When comparing WARMR to graph mining algorithms such as gSpan, we note the following points:

- the greater expressiveness of WARMR: specialized mining algorithms often concentrate on one type of database, for example databases of labeled undirected graphs. For different kinds of structures, modified algorithms are required. In ILP algorithms, such as WARMR, any structure can be expressed easily. The incorporation of background knowledge is also straightforward.
- the choice for traditional clause based query evaluation in WARMR: in this case, two variables may have the same value during evaluation. In the subgraph mining algorithms two nodes in a subgraph cannot be mapped to one node in a database graph.
- in publications of subgraph mining algorithms [6,7,10], much attention is given to efficiency issues. The WARMR algorithm can be considered as a proof-of-concept of a framework; efficiency issues have not been given too much attention.

In this paper, we will introduce a new algorithm for frequent query mining. While it is largely comparable to WARMR from an expressive point of view,

N. Lavrač et al. (Eds.): PKDD 2003, LNAI 2838, pp. 350–362, 2003.

it uses techniques introduced by subgraph mining algorithms as well as new techniques. The main contributions of the paper are:

- We show how the query discovery task can be changed to query discovery under *Object Identity*. Although the focus of this paper is not on the semantic consequences of this choice, we will argument that this approach closely matches that of subgraph mining, is very natural and does not pose restrictions in many data mining situations.
- Building upon this evaluation under Object Identity, and using a tree data structure, we will define an order on queries that allows for more efficient search space traversals than the approach used by WARMR. To some extent this order is equivalent to that of gSpan; it is however more flexible and allows for some new optimizations.
- We will show how this order can be exploited in both breadth-first and depth-first algorithms. For the latter case we will introduce optimizations that are allowed by the query ordering, including hash structures and sorting to reduce the cost of query evaluations that result in false.
- We will present experimental results showing large speed-ups in comparison with a recent implementation of WARMR. We will also compare our results to those obtained by gSpan and FSG. In some cases, our algorithm obtains similar run times as FSG, but it does not equal the efficiency of gSpan. We will give some arguments for this difference in performance.

Our aim is to use ILP formalisms that are very close to WARMR and to reach the efficiency of algorithms like gSpan and FSG.

Our depth-first and breadth-first algorithms are major revisions of our previous FARMER algorithm for mining multiple relations [8]. The algorithm in [8] was restricted to some variants of labeled, unordered trees and did not use Object Identity for query evaluation. In the breadth-first algorithm presented here, only the tree-like notation is reused. Restrictions that were present in the previous version of FARMER, do no longer exist in our new algorithm. We will however still use the name FARMER to denote our class of algorithms.

2 Search Space Specification and Object Identity

We will introduce some notation. Any capital A denotes an atom. An *atom set* S is an unordered set of atoms. An ordered atom set is called a *query* and is denoted by a capital Q. With (Q, A) we denote the query Q to which atom A is concatenated. With $last(Q)$ we denote the last atom of Q. The variables in $A = last(Q)$ that do not occur in $Q \backslash A$ are called the *new variables* of A in Q.

Every predicate p is considered to be *typed*: each argument has a type. A variable or constant that is used as an argument of a predicate, has the same type as the argument. Types are frequently used in ILP systems to allow the definition of more narrow search spaces. With $var(S, T)$ (or $var(Q, T)$) we denote the set of all variables of type T in an atom set S.

We will first introduce a mechanism that defines the search space of queries that our algorithm will investigate. It uses a similar mode mechanism as WARMR.

Definition 1 (Bias). *A* mode *declaration $p(c_1, \ldots, c_n)$ consists of a predicate with arguments c_i, each of which is either '+' (input), '-' (output) or '#' (constant). The* bias $\mathcal{B}$ *of a search space consists of: 1) the type definitions of predicates, 2) a set of modes $\mathcal{M}$, 3) an operator $const(T)$ which defines for each type T a set of constants, 4) a function* max *which assigns an integer to each predicate in $\mathcal{M}$, and 5) one atom $k(X)$ (this atom is called the* key *of the search).*

Definition 2 (Search space). *Given a bias $\mathcal{B}$, a query Q and an atom $A = p(t_1, \ldots, t_n)$, atom A is a (mode)* refinement *of Q iff there is a mode $M = p(c_1, \ldots, c_n) \in \mathcal{M}$ such that for every $1 \leq i \leq n$ either:*

- *t_i is a variable in $var(Q, T_i)$ and c_i ='+', or*
- *t_i is a variable not in $\cup_j var(Q, T_j)$ and c_i='-', or*
- *t_i is a constant in $const(T_i)$ and c_i='#'.*

Here, T_i is the type of argument position i. The search space $\mathcal{S}(\mathcal{B})$ defined by a bias $\mathcal{B}$ consists of all queries Q that can be built iteratively starting from the key atom $k(X)$ using valid *refinements. Each atom A that is added to a query Q should satisfy the following restrictions to be a valid refinement: 1) A is a mode refinement; 2) A does not already occur literally in Q; 3) the predicate p used in A does not occur more than* $\max(p)$ *times in the new query.*

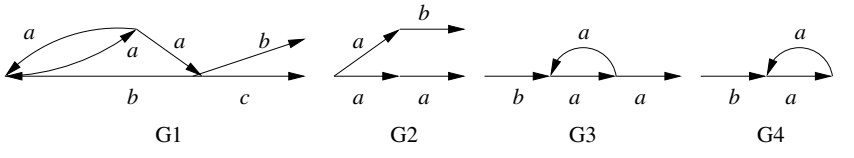

Fig. 1. Directed, edge labeled graphs.

Example 1. As an example we will use the representation of a directed, edge labeled graph using a predicate $e(G, N, N, L)$. Graph G1 in Fig. 1 can be represented using the following facts:

$$K = \{k(g_1), e(g_1, n_1, n_2, a), e(g_1, n_2, n_1, a), e(g_1, n_2, n_3, a), e(g_1, n_3, n_1, b), e(g_1, n_3, n_4, b), e(g_1, n_3, n_5, c)\}.$$

Of course, the choice of constants n_i is arbitrary here. Using the set of modes $\mathcal{M} = \{e(+,-,-,\#), e(+,+,-,\#), e(+,+,+,\#)\}$ the following queries can be constructed:

$$Q2 = k(G), e(G, N_1, N_2, a), e(G, N_2, N_3, a), e(G, N_1, N_4, a), e(G, N_4, N_5, b),$$
$$Q3 = k(G), e(G, N_1, N_2, b), e(G, N_2, N_3, a), e(G, N_3, N_2, a), e(G, N_3, N_4, a),$$
$$Q4 = k(G), e(G, N_1, N_2, b), e(G, N_2, N_3, a), e(G, N_3, N_2, a);$$

they correspond to graphs G2, G3 and G4 in Fig. 1.

Given a knowledge base K the support of a query Q can be defined as:

$$support_K(Q) = \#\{\theta \mid K \models Q\theta\},$$

where θ is a substitution to constants of all variables in the key of Q; $Q\theta$ denotes the application of this substitution to Q.

In WARMR, to compute the $\models$ relation, a Prolog engine based on θ-subsumption is used. For a knowledge base containing only facts, this evaluation comes down to the discovery of a substitution such that $Q\theta \subseteq K$. On the other hand, we will use an evaluation technique based on subsumption under *Object Identity (OI-subsumption)*. Under Object Identity, the satisfying substitution θ is constrained in two ways: no two variables in Q may be mapped to the same constant, and no variable may be mapped to a constant already occurring in Q.

We will briefly illustrate some consequences of this choice. Under usual θ-subsumption, example query Q_2 is a consequence of the knowledge base K, as Q_2 can be satisfied by mapping $N_1 \to n_2, N_3 \to n_2, N_2 \to n_1, N_5 \to n_1, N_4 \to n_3$. In the graph notation of Fig. 1, some nodes in G2 are mapped to the same nodes in G1. Under Object Identity, this is not allowed: the mapping must be injective. Such an injective mapping is also used in gSpan for labeled undirected graphs. Similar arguments show that G3 is not included in G1 under Object Identity, while it would be included under traditional θ-subsumption.

An important issue is that of query *equivalency*. In general, two queries Q_1 and Q_2 are equivalent iff for every possible knowledge base K: $K \models Q_1 \Leftrightarrow K \models Q_2$. For evaluation without Object Identity, one can prove that Q_1 and Q_2 can only be equivalent when Q_1 and Q_2 mutually subsume each other. Without OI, G3 and G4 in our example are equivalent. Every graph which contains G4 also contains G3, as node N_4 can always be mapped to the same node as N_2. The first reason for choosing Object Identity is that these counterintuitive situations are prevented under OI. Under OI queries are equivalent iff they are *alphabetic variants* [4,5]. We will define this equivalency relation more precisely. Given a query Q, let $vars(Q)$ denote the set of all variables occurring in Q and let $varlf(Q)$ denote the list of all variables Q in order of first occurrence.

Definition 3 (Equivalency of queries). *Given a query Q, the normally named query $n(Q)$ is the query Q to which the following renaming substitution is applied: $\theta = \{V/V_i | V \in vars(Q), i = ord(V, Q)\}$. Here $ord(V, Q)$ is the position of V in $varlf(Q)$. Two queries Q_1 and Q_2 are equivalent (denoted by $Q_1 \equiv Q_2$) if there exists a permutation π of the atoms in Q_1 such that $n(\pi(Q_1)) = n(Q_2)$.*

To determine whether two queries are equivalent, is therefore 'only' a problem of finding a permutation which transforms the one query into the other. This is still a difficult problem; it can be shown that to compute whether two queries are equivalent, one has to solve a graph isomorphism problem, and vice versa. The complexity of graph isomorphism is currently unknown: no polynomial algorithm is known, and a proof of NP completeness does not exist either. In comparison with full θ-subsumption, however, OI makes the computation of equivalency slightly easier. This property is the second reason for choosing OI.

We will now present our pattern mining task.

Definition 4. *Given a bias $\mathcal{B}$, a knowledge base K and a threshold minsup,* FARMER *should discover a set of queries $\mathcal{Q}$ such that for every $Q \in \mathcal{S}(\mathcal{B})$ with $support_K(Q) \geq minsup$, there is exactly one $Q' \in \mathcal{Q}$ such that $Q' \equiv Q$.*

A query for which $support_K(Q) \geq minsup$ is said to be *frequent*. The single query in $\mathcal{Q}$ to which a query Q is equivalent is considered to be its *normal form* or its *canonical label*.

The third advantage of OI can be understood by considering Q_2 in conjunction with the following modes, which define a search space of edge labeled *trees*: $\{e(+,-,-,\#), e(+,+,-,\#)\}$. Query Q_2 is not equivalent with any smaller query. Every subquery $Q_2' \in \mathcal{S}(\mathcal{B})$ of Q_2 with $|Q_2| = |Q_2'| + 1$ is however equivalent with a query smaller than $|Q_2'|$. An algorithm which relies on refinement with building blocks of one atom, will not construct Q_2 if it removes equivalent queries immediately. Such difficulties with refinement are avoided under OI.

The choice for Object Identity has many consequences on the types of patterns that can be discovered. As an illustration consider a situation in which one also allows *wildcards* as labels. A possible query in this case would be:

$$k(G), e(G, N_1, N_2, L_1), L_1 = a, e(G, N_2, N_3, L_2), e(G, N_4, N_5, L_3), L_3 = b.$$

Under full OI, all labels L_1, L_2 and L_3 must be different. Although for clear objects (such as nodes), an inequality constraint is a natural choice, for properties (such as the label of an edge) inequality can be undesirable. An elegant solution could be to use a variant of Object Identity which does not force OI on variables for such properties; in this weaker OI, one can sometimes (and also in this example) still guarantee the three properties of Object Identity that we exploit. Due to lack of space, we refer to [9] for more details about OI related issues.

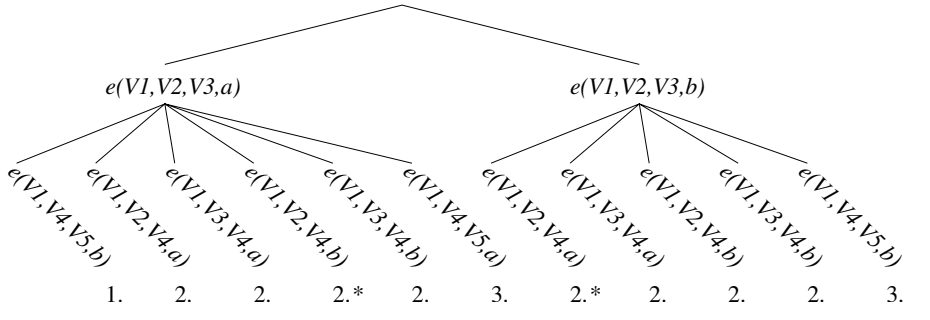

Fig. 2. A query tree.

3 A Tree Based Normal Form

In our algorithm, all queries are stored in an ordered tree as given in Fig. 2. Every node in this tree is labeled with an atom. Every path starting starting in the root represents a query. Every node has therefore an associated query. Once

a query is counted, its support is stored in the associated node. The query tree is similar to the *query pack* tree used by WARMR for efficient query evaluation [2]. By introducing an order on nodes in the tree, FARMER adds as main application of the tree the efficient determination of candidate queries. The order of queries is determined by their order in the tree:

Definition 5 (Order of queries). *Let* $Q_1 = (Q_p, A_p, A_1, Q'_1)$ *and* $Q_2 = (Q_p, A_p, A_2, Q'_2)$, $A_1 \neq A_2$ *be two queries (where* Q_p, Q'_1 *and* Q'_2 *may be empty), then* $Q_1 <_T Q_2$ *iff* $A_1 < A_2$ *in the child list of* (Q_p, A_p).

If $Q_1 <_T Q_2$, then Q_1 is called an *earlier* query than Q_2 or Q_2 is called a *later* query than Q_1.

An outline of the FARMER algorithm is given in Algorithm 1 and 2. In line (5) of Algorithm 1, the order in which nodes are expanded is intentionally left unspecified. The order is only restricted by the precondition of FARMER-EXPAND. In line (5) of FARMER, and line (3) of Algorithm 2 this observation is used: $\forall Q_2 : (\exists Q_1 \subseteq Q_2 : support(Q_1) < minsup) \Rightarrow support(Q_2) < minsup$.

Algorithm 1: FARMER
Input: A bias $\mathcal{B}$, a knowledge base K and a threshold *minsup*.
Output: A tree T with all queries according to Definition 4.
(1) Read K and determine $const(T)$ for each type T
(2) $T :=$ a tree with only the key atom in the root
(3) **repeat**
(4) Count the frequency of all uncounted queries.
(5) **for** one or more uncounted, unmarked, frequent leafs **do**
(6) Expand that leaf
(7) **until** T contains no uncounted queries
(8) Remove all marked nodes

Algorithm 2: FARMER-EXPAND
Input: A query Q in a tree T with counts for (1) all ancestor queries of Q, (2) all earlier queries Q', $|Q'| \leq |Q|$; (3) all later queries Q' which are a brother of an ancestor of Q.
Output: A query tree with uncounted expansions Q' of Q, $|Q'| = |Q| + 1$.
(1) Let A be $last(Q)$; let A_p be the parent of A and Q_p the query associated with A_p.
(2) Add as child of A all valid refinments $A'' = last(n(Q, A'))$, where A' is either:
(3) 1. a frequent atom occurring after A in A_p's child list, where new variables in A' are renamed such that they are also new in (Q, A').
(4) 2. a *dependent* atom, which is any atom that uses at least one variable that was new in A.
(5) 3. a copy of A if A has new variables; those new variables are given new names in the copy.
(6) Remove the new child query A'' if it is equivalent to an earlier query, unless the child is only equivalent to a brother. In that case A is marked but kept in the tree.

It is this property that has led to the popularity of APRIORI-like algorithms: this property restricts the search space in such a way that is possible to compute all frequent patterns if the threshold is not too low.

We will first consider the resulting tree T when all queries are frequent and FARMER-EXPAND line (6) is absent. We will show that for every query in the search space, at least one equivalent query can be found in this tree.

Example 2. Under these assumptions, and given modes $e(+,-,-,\#)$ and $e(+,+,-,\#)$, Fig. 2 shows for each query of length 2 how they are obtained from queries of length 1 by applying FARMER-EXPAND. Each number indicates which of the three possibilities is applied to generate a new atom.

Lemma 1. *Given is a query Q which occurs in a Query Tree T generated by* FARMER*, and an atom $A \notin Q$ which is a valid refinement of Q. Then a query $Q' = n(Q_1, A, Q_2)$ exists in the tree T, for some subdivision of Q into Q_1 and Q_2, $Q = (Q_1, Q_2)$. Furthermore, Q is either a prefix of Q' or $Q' <_T Q$.*

Proof. As A is a valid refinement of Q, there is a prefix (Q_p, A_p) of Q such that the normalized atom $A' = last(n(Q_p, A_p, A))$ is a dependent atom of A_p. This dependent atom is generated in line 4 of the FARMER-EXPAND algorithm. If A_p is the last atom of Q, our statement is clear. Therefore assume that A_p has a different successor A_{p+1} in Q. This atom A_{p+1} is also a child of A_p in T. Consider the order of A' and A_{p+1} in the list of children:

- if A' occurs before A_{p+1}, A_{p+1} is a right-hand child of A'. The copying mechanism in line 3 will copy A_{p+1} as a child of A'; all steps which created Q are applicable subsequently and result in a query Q'.
- if A' equals A_{p+1}, both have output variables. In line 5 a self-duplicate A_{p+1} of A' is generated. All steps which created Q are applicable subsequently.
- if A' occurs after A_{p+1}, A' is copied as a child of A_{p+1}. This child of A_{p+1} may be left or right from A_{p+2} (the next atom in the original query). We can recursively apply our arguments on the situation for $p+1$ until one of the above conditions holds.

Also the order of the old and new query follows from these arguments. □

Theorem 1 (Completeness of search). *For every query Q_1 in the search space, there is at least one equivalent query Q_2 in the tree T.*

Proof (Sketch). This can be shown by induction on the length of the query. A query with only the key occurs in T. By inductive assumption, an equivalent query for $Q_1 \backslash last(Q_1)$ exists in the tree, and a corresponding variable renaming. When $last(Q_1)$ is renamed accordingly, this renamed atom is a valid refinement of the equivalent query, and one can apply Lemma 1. □

Two equivalent queries that still coexist without line 6 in Algorithm 2 are indicated with a (*) in Fig. 2. We will now consider the algorithm with this line

added. We have to prove that by removing an atom from the tree, we do not remove an atom that otherwise would have been used to create a query for which no equivalent query exists.

Lemma 2. *Let T be the tree obtained after iterative application of Algorithm 2 without line 6. Assume that a query Q_2 is equivalent with a query $Q_1 <_T Q_2$. Then every query Q_2' which has Q_2 as prefix must have an equivalent query more left in the tree.*

Proof. As Q_2 is equivalent with Q_1, there is a permutation of atoms of Q_2 followed by a renaming θ that makes Q_2 equal to Q_1. This substitution θ can be applied to all atoms in $Q_s = Q_2' \backslash Q_2$. Some of these atoms are now valid refinements of Q_1. According to Lemma 1 one by one these atoms can be added to Q_1, yielding queries Q_1' that are either extensions of Q_1 or occur $Q_1' < Q_1$. □

Theorem 2. *For every query defined by the bias, Algorithm 1 generates exactly one normal form if all queries are frequent.*

Proof. It is clear that no two normal forms can occur: in line 6 of Algorithm 2 and line 8 of Algorithm 1, any query which has an equivalent lower query is removed. Theorem 1 showed that if equivalents were not removed, the search is complete. According to Lemma 2, if a query Q is equivalent to an earlier query, all of its descendents must also be equivalent to an earlier query. Q should therefore not be expanded further. The only remaining function of atom $last(Q)$ is its function as an expansion for earlier brothers in line 3 of Algorithm 2. In case Q is equivalent to an earlier query Q' which is not a brother, $last(Q)$ is not required as a building block for earlier brothers: the brother atom can be added to Q to yield a query $Q'' < Q'$ and every expansion of Q' can also be added to Q'' (similar to the construction of Lemma 2). By the marking mechanism only those atoms are kept as building block that are equivalent to an earlier brother. □

A consequence of the monotonicity constraint is that every building block of a query Q must also be frequent. From this observation it follows that our algorithm performs exactly the task that was defined in Definition 4.

4 Depth First and Breadth First Algorithms

In the algorithm discussed in the previous section, many elements have been kept unspecified. In this section, we give an overview of some details.

Equivalency Check. To determine whether an earlier equivalent query exists, we essentially use an exhaustive search algorithm. Given a query Q, the mode mechanism is used recursively to build queries Q' that contain atoms in Q. After an atom is added, the tree T is consulted to determine whether Q' is later (in which case Q' is not further expanded) or infrequent (in which case Q cannot be frequent and is pruned). Once a query $Q' < Q$ is found which contains all

atoms of Q, Q is pruned. Especially the combination of frequency pruning with equivalency pruning is a distinctive feature of our algorithm. Although it requires infrequent nodes to be stored in the tree, it could give the exhaustive exponential search an additional value and could reduce the number of queries that should be counted later significantly.

Order of Query Expansion. We distinguish two query expansion orders: *breadth first* and *depth first*. In the breadth first approach, all nodes at the lowest level of the tree are expanded. This yields a tree in which all nodes at the new lowest level are uncounted. The nodes are counted next, and the process is repeated until no new level can be added.

In the depth first approach, only one node is expanded; the new children are counted immediately. Starting with the first child, the process is recursively repeated. Only after the complete subtree of the first child has been constructed, the next child is recursively expanded.

In both approaches, the precondition of Algorithm 2 is satisfied. Breadth first is the traditional approach and corresponds to the evaluation order of APRIORI [1], WARMR [3] and FSG [6]. The depth first order matches that of gSpan.

Query Counting. To determine whether a query is OI-subsumed by a knowledge base of facts, an exponential search is required (one can easily see that this problem is equivalent to the subgraph isomorphism problem, which is known to be NP hard). Especially those queries which can *not* be satisfied for a given key substitution are computationally very expensive as many variable assignments have to be checked before this can be concluded. The task of the algorithm is to reduce the number of key substitutions which result in *false* as much as possible, and to reduce the cost of such an evaluation if the computation is required.

One strategy to reduce the computational cost, is to overlap the computation of queries. Consider a query Q with several child expansions. One can backtrack over all possible assignments of Q as long as one of the child expansions is not satisfied. This is more efficient than to evaluate each child expansion separately.

The advantage of the breadth-first approach is that the number of queries that should be evaluated at a certain level is maximal. For a given substitution of key variables, the evaluation of many queries can be combined. Our breadth first implementation uses this evaluation technique, which is similar to *query packs* as discussed in [2] for WARMR and [8] for our previous FARMER algorithm.

To reduce the number of false evaluations, a substitution ID list approach can be used. For each query that is evaluated, one can store the list of all key substitutions for which the query can be satisfied. One can easily see that a query which is constructed from a query Q (either by copying $last(Q)$ or by expanding Q) can never be *true* for key substitutions for which Q is *false*. Therefore only substitutions in Q's SID list need to be evaluated.

To reduce the cost of evaluation, with each key substitution θ one can also store the variable assignments that satisfy each query Q. If the backtracking over variables is performed in a deterministic order from left to right, one can continue the evaluation of each expansion of Q starting from the assignment that

satisfied Q without having to recompute that assignment. Some assignments are skipped in this way, but one can show that this can safely be done. To reduce the memory demand of the approach, for each query Q and key substitution θ we only store the difference $\Delta(\theta, Q)$ between the first variable assignment that satisfies Q and the first assignment that satisfies the parent in T of Q.

Order of Children. There are many possible child orders:

- The order in which children are generated in Algorithm 2. This is the order that we used in [8] and yields queries that are very well readable.
- A lexicographical order. To determine query equivalency, one repeatedly has to search for a given atom in a set of children. With a lexicographical order, in combination with binary search and hashing, we speed up this search.
- Sorted by support. Atoms with a lower support occur earlier in a query in this case, which results in a quicker evaluation of queries that cannot be satisfied (the most selective atoms occur earliest).
- Sorted by *backtracking progression*. Consider a query Q, a key substitution θ and a set $\Delta(\theta, Q)$ of variable assignment changes. The position of the leftmost variable affected by $\Delta(\theta, Q)$ in Q is the backtracking progression of Q for θ. By averaging $\Delta(\theta, Q)$ over all θ one can compute the average backtracking progression of each query. When a candidate query (Q, A) is generated by copying an atom A below a query Q, both $\Delta(\theta, Q)$ and $\Delta(\theta, A)$ could be used as starting point for the evaluation of (Q, A); best would be to always use the assignment which has backtracked most. However, when the evaluation of several queries is overlapped, much additional bookkeeping would be required. As tradeoff we always use $\Delta(\theta, Q)$ as starting point, but sort to make sure that the parent has backtracked most on average.

Note that in the last two orders, some special care has to be taken in the equivalency procedure, as the order of children is only known *after* they are counted.

5 Experimental Results

From the possibilities discussed in the previous section, we implemented and tested several (see [9]). We implemented a breadth-first algorithm with naive sorting order and evaluation without substitution ID lists as a reference algorithm. Furthermore we implemented a depth-first algorithm which incorporated overlapping evaluation and a complex sorting order: given a query Q that is going to be expanded, all children of nodes that are not an ancestor of Q are stored in lexicographical order to allow for quick equivalency checks; nodes on the path corresponding to Q are also sorted first on backtracking cost, then on support and finally lexicographically. These two orders can be combined in an efficient way. From our experiments, we concluded that it is most beneficial.

Bongard Dataset[1]**.** The Bongard dataset [2] was used to compare WARMR, depth-first and breadth-first FARMER (Fig. 3). In the experiments, FARMER was

[1] Experiments were performed on a Linux Pentium II 350Mhz with 192MB RAM, using the GNU C++ compiler, version 2.96 with O3 code optimization setting.

clearly several orders of magnitude faster than WARMR. One should however realize that in these experiments, WARMR was provided with a bias that forced Object Identity by adding inequality atoms. WARMR was not optimized for this. Part of the efficiency difference may also be due to the different programming language that was used (Prolog).

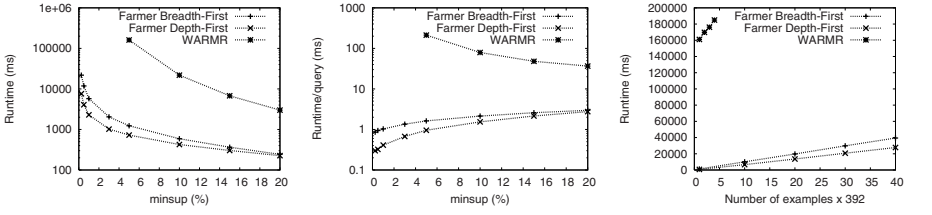

Fig. 3. Results on the Bongard dataset. Default dataset size is 392, $minsup = 5\%$.

Predictive Toxicology Evaluation Challenge (PTE). Execution times for PTE were published in [10], [6] and [7]; in these publications, labeled, *undirected* graphs were constructed from the atom and bond information; one searches for *connected* frequent subgraphs. To emulate the injective setup of gSpan and FSG, Object Identity is a necessity. To deal efficiently with connected, undirected graphs, the mode mechanism that was described in this article is not powerful enough. Therefore, we incorporated a more powerful declarative formalism based on *mode trees* in FARMER. Due to space limitations, we omit the details.

Table 1 and Fig. 4 display some execution times. We also show some execution times of other publications to set these into a perspective. Note that our algorithm runs on computers with relatively few memory, even though the ID lists augmented with variable assignments have to be stored in main memory.

Table 1. Comparison of execution times on the PTE dataset for $minsup \in \{6\%, 7\%\}$.

Machine	Algorithm	6% (s)	7% (s)
Intel Pentium III 500Mhz 448MB	gSpan [10]	5s	
AMD Dual Athlon MP1800+ 2GB	FSG Iterative Partitioning [7]	11s	7s
AMD Athlon XP1600+ 265MB	FARMER	72s	48s
Intel Pentium II 350Mhz 192MB	FARMER	224s	148s
Intel Pentium III 500Mhz 448MB	FSG [10]	248s	
AMD Dual Athlon MP1800+ 2GB	FSG Inverted index [7]	675s	23s
Intel Pentium III 650Mhz 2GB	FSG [6]		600s

We may conclude that our algorithm does not reach the state-of-the-art performance of gSpan. Compared to other graph mining algorithms, its performance is reasonable. We could easily compute all frequent subgraphs down to a support of 3%. The performance of gSpan is hard to obtain with the more general setup

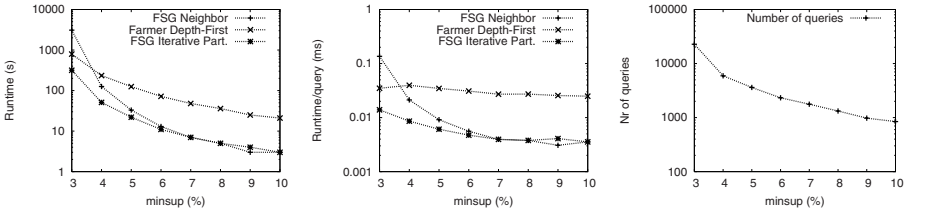

Fig. 4. Results on the PTE dataset. FARMER was run on an AMD Athlon XP1600+. For FSG results published [7] for an AMD Dual Athlon MP1800+ are used.

that we are dealing with. For example, gSpan orders the labels on the vertices first and performs a depth-first search to discover the graphs with the first label first. Next, all vertices with the first label are removed from the database, and the process is repeated for the remaining graphs. In general, this optimization is harder to apply. Therefore, one could better use gSpan if one is exactly searching for the kind of patterns that gSpan is optimized for.

Mutagenesis. The Mutagenesis dataset is very similar to the PTE dataset and was also used in [2]. We use it to compare FARMER with WARMR without Object Identity. Using a minimum support of 20%, WARMR discovers 91 frequent queries in 207s (of which 205s are spent while generating candidates). On the same Intel Pentium II FARMER discovers 1075 frequent queries in 73s. The different number of queries is due to the fact that WARMR does not discover graphs like $C-C-C$, as these are equivalent to $C-C$ without Object Identity. The set of queries found by FARMER is a proper superset of those found by WARMR.

6 Conclusion

In this article we presented an efficient algorithm for discovering frequent queries. We used Object Identity and a tree data structure to introduce several optimizations. Experiments showed that the algorithm outperforms WARMR and is comparable with some more specialized algorithms, but is not as efficient as recently published graph mining algorithms.

Acknowledgements

We are grateful to the Artificial Intelligence research group of the KU Leuven and to Xifeng Yan for their help with some experiments.

References

1. Agrawal, R., Manilla, H., Srikant, R., Toivonen, H., Verkamo, A.: Fast Discovery of Association Rules. In: U.M. Fayyad et al. (eds). Advances in Knowledge Discovery and Datamining. AAAI/MIT Press (1996) 307–328.

2. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research 16. (2002) 135–166.
3. Dehaspe, L., Toivonen, H.: Discovery of frequent Datalog patterns. In: Data Mining and Knowledge Discovery 3, no. 1. (1999) 7–36.
4. Esposito, F., Laterza, A., Malerba, D., Semeraro, G.: Refinement of Datalog Programs. Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programming. (1996) 73–94
5. Ferilli, S., Fanizzi, N., Mauro, N., Basile, T.: Efficient θ-subsumption under Object Identity. In: Proceedings of the ICML'02. (2002)
6. Kuramochi, M., Karypsis, G.: Frequent Subgraph Discovery. In: Proceedings of the ICDM'01. (2001) 313–320
7. Kuramochi, M., Karypsis, G.: An Efficient Algorithm for Discovering Frequent Subgraphs. Technical Report 02-026, University of Minesota. (2002).
8. Nijssen, S., Kok, J.N.: Faster Association Rules for Multiple Relations. Proceedings of the IJCAI'01. (2001) 891–897
9. `http://www.liacs.nl/home/snijssen/farmer`
10. Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In: Proceedings of the ICDM'02. (2002)