Executing Query Packs in ILP

Hendrik Blockeel
Luc Dehaspe
Bart Demoen
Gerda Janssens
Jan Ramon
Henk Vandecasteele

Report CW 287, May 2000

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Executing Query Packs in ILP

Hendrik Blockeel
Luc Dehaspe
Bart Demoen
Gerda Janssens
Jan Ramon
Henk Vandecasteele

Report CW 287, May 2000

Department of Computer Science, K.U.Leuven

Abstract

Inductive logic programming systems usually send large numbers of queries to a database. The lattice structure from which these queries are typically selected causes many of these queries to be highly similar. As a consequence, independent execution of all queries may involve a lot of redundant computation. We propose a mechanism for executing a hierarchically structured set of queries (a "query pack") through which a lot of redundancy in the computation is removed. We have incorporated our query pack execution mechanism in the ILP systems Tilde and Warmr by implementing a new Prolog engine ilProlog which provides support for pack execution at a lower level. Experimental results demonstrate significant efficiency gains. Our query pack execution mechanism is very general in nature and could be incorporated in most other ILP systems, with similar efficiency improvements to be expected.

Keywords : Inductive logic programming, machine learning, data mining.
AMS(MOS) Classification : Primary : I.2.6, Secondary : I.2.3.

1 Introduction

Many data mining algorithms, including ILP algorithms, employ an approach that is basically a generate-and-test approach: large numbers of hypotheses are generated and tested against the database in order to check whether they are valid. Even though their search through a hypothesis space is seldom exhaustive in practical situations, and clever branch-and-bound or greedy search strategies are employed, the number of hypotheses generated and tested by these approaches may still be huge. This is especially true when a complex hypothesis space is used, which is often the case in ILP.

Very often the hypothesis space is structured as a lattice, and the search through the space makes use of this lattice. Because hypotheses close to one another in the lattice are similar, the computations involved in testing their validity will be similar too. In other words, many of the computations that are performed when executing one query will have to be performed again when executing the next query. Storing certain intermediate results during the computation for later use could be a solution (e.g., tabling as in the XSB Prolog engine [7]), but may be infeasible in practice because of its memory requirements. It becomes more feasible if the search is reorganised so that intermediate results are always used shortly after they have been computed; this can be achieved to some extent by rearranging the computations. The best way of removing the redundancy, however, is to re-implement the execution strategy of the queries so that as much computation as possible is effectively shared.

In this paper we discuss a strategy for executing sets of queries, organised in so-called query packs, that avoids these redundant computations. The strategy is presented as an adaptation of the standard Prolog execution mechanism. We also report on the use of this technique by several inductive logic programming systems. Experimental results suggest that in some cases a speed-up of an order of magnitude or more can be achieved in this way. This significantly contributes to the applicability of inductive logic programming to real world data mining tasks.

The remainder of this paper is structured as follows. In Section 2 we precisely describe the ILP problem setting in which this work is set. In Section 3 we define the notion of a query pack, indicate how it could be executed by a standard Prolog interpreter and what computational redundancy this causes, and propose an algorithm to execute such query packs that avoids these redundant computations. In Section 4 we describe how the query pack execution strategy can be incorporated in two existing inductive logic programming algorithms (Tilde and Warmr). In Section 5 we present experimental results that give an indication of the speed-up that these systems achieve by using the query pack execution mechanism. In Section 6 related work is mentioned and in Section 7 we conclude.

2 Problem setting

The problem setting we are considering is the following [8]:

Given

- a set of conjunctive queries S
- a deductive database D
- a tuple K of variables that occur in all queries in S

Find

- the set $R = \{(K\theta, Q) | Q \in S \text{ and } Q\theta \text{ succeeds in } D\}$; i.e., find for each query Q in S those instantiations of K for which the query succeeds in D.

We refer to the tuple K as a *key*; the intuition behind K is that it uniquely identifies a single example. In the context of learning a single predicate the key would typically consist of the variables that occur in the head of the clause being learned.

Example 1. Assume an ILP system learning a definition for `father/2` wants to evaluate the following hypotheses:

```
father(X,Y) :- parent(X,Y), male(X).
father(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(Y).
father(X,Y) :- parent(X,Y), female(Y).
```

Examples are of the form `father`(c_1,c_2) with c_1 and c_2 some constants; hence in the above set of clauses each example is uniquely identified by a ground substitution of the tuple (X, Y). This means that, put in the above problem setting, we have a set of Prolog queries S $= \{(parent(X,Y), male(X)), (parent(X,Y), female(X)), (parent(X,Y), male(Y)), (parent(X,Y), female(Y))\}$ and a key $K = (X, Y)$. Given a query $Q \in S$, finding all couples (x, y) for which $((x, y), Q) \in R$ (with R the result set as defined above) is equivalent to finding all `parent`(c_1,c_2) facts predicted by the clause `parent(X,Y) :-` Q.

Our problem setting is very general in the sense that it covers many tasks typically encountered in ILP settings. Indeed, once it is known which queries succeed for which examples, statistics about the queries can be readily obtained from this. A few examples:

- discovery of frequent item-sets: for each query Q the number of keys for which it succeeds just needs to be counted, i.e., $|\{K\theta | (K\theta, Q) \in R\}|$
- induction of Horn clauses: the accuracy of a clause $H : -B$ can be computed as $\frac{|\{K\theta | (K\theta,B) \in R \text{ and } D \models H\theta\}|}{|\{K\theta | (K\theta,B) \in R\}|}$ with R the result set
- induction of classification or regression trees: computing the class entropy or variance of the examples covered (or not) by a query involves simple computations on counts similar to the above ones

Obviously our problem setting returns more information than most systems need; in practice one could avoid computing the result set itself and just update some relevant counters instead. By considering the above setting any efficiency results we obtain are *a fortiori* applicable to systems avoiding the computation of this overhead of information.

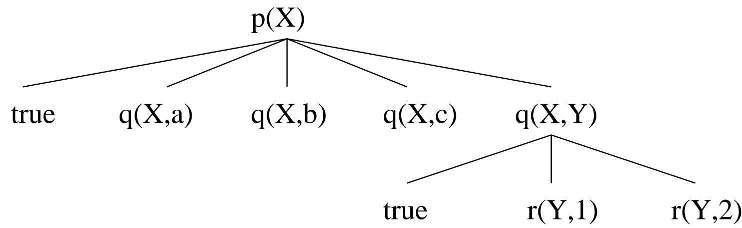

Fig. 1. A query pack

3 Query packs

3.1 Definition

Definition 1. *A query pack is a tree structure with literals or conjunctions of literals in the nodes. Each path from the root to some node (leaf or internal node) represents a conjunctive query. Such a query is said to be a* member *of the query pack.*

Example 2. Given the following set of queries:

```
p(X)
p(X), q(X,a)
p(X), q(X,b)
p(X), q(X,c)
p(X), q(X,Y)
p(X), q(X,Y), r(Y,1)
p(X), q(X,Y), r(Y,2)
```

the query pack shown in Fig. 1 contains these and only these queries as members. We represent the pack as a Prolog term as

p(X), (true **or** q(X,a) **or** q(X,b) **or** q(X,c) **or** q(X,Y), (true **or** r(Y,1) **or** r(Y,2)))

As in the example, we will denote query packs textually by writing an **or** operator between different subtrees of a node. We name the operator **or** because intuitively it closely corresponds to a disjunction. Indeed a query pack could be constructed and executed as a complex Prolog query reflecting the tree structure of the pack (with a ";" separating children of the same node and a "," separating a node from its children): the set of answer substitutions for this query is then exactly the union of the sets of answer substitutions for the members of the pack. Executing the pack in this manner would indeed remove a lot of redundant computation, but is still suboptimal, as the following example shows.

Example 3. Consider the following set of queries, with X as key:

```
p(X), lotsofwork(X), q(X,Y)
p(X), lotsofwork(X), r(X,Y)
```

where `lotsofwork` represents a complex computation. Evaluating the queries separately implies that for each X for which `p(X)` succeeds, `lotsofwork(X)` will be computed twice, whereas for the following query

```
p(X), lotsofwork(X), (q(X,Y); r(X,Y))
```

`lotsofwork(X)` will only be called once for each X. Both queries are equivalent with respect to their answer substitutions.

On the other hand, consider the following query:

```
p(X), q(X,Y), (r(Y,Z); s(Y,Z))
```

Suppose `r(Y,Z)` associates many Z's with each single Y. With the above query, all alternative values for Z will be generated by `r(Y,Z)` before `s(Y,Z)` be called. Given that we are only interested in answer substitutions for X, getting one single answer substitution for Z suffices and the alternatives for `r(Y,Z)` should be cut away. Note that the cut should survive backtracking to some extent: when backtracking to `q(X,Y)` yields a new value for Y, the `r(Y,Z)` branch should still be avoided because we already know that it succeeds for this value of X. Only when the value of the key X changes should it be reconsidered.

The execution mechanism we need can be implemented in Prolog, but only in a very inefficient manner [11, 2]; actually preliminary results in [2] suggested that the overhead involved in this implementation destroys the efficiency gain obtained by redundancy reduction. To fully exploit the sharing of computations changes are needed at the level of the Prolog engine itself.

3.2 Efficient Execution of Query Packs

Several possible solutions for efficient execution of query packs are described in [11]. The most efficient one consists of an extension of the WAM (Warren Abstract Machine), the machine underlying most Prolog implementations. The extended WAM provides the **or** operator as discussed above, and permanently removes branches from the pack that do not need to be investigated anymore. This extended WAM is the basis of a new Prolog engine dedicated to inductive logic programming, called ILPROLOG.

The starting point for the query pack execution mechanism is the usual Prolog execution of a query Q given a Prolog program P. By backtracking Prolog will generate all the solutions for Q by giving the possible instantiations θ such that $Q\theta$ succeeds in P.

In this context, a query pack consists of a conjunction *conj* of literals followed by a set of branches, where each branch is again a query pack. Note that leaves are query packs with an empty set of branches. For each query pack $\mathcal{Q}$, $conj(\mathcal{Q})$ denotes the conjunction and $children(\mathcal{Q})$ denotes the set of branches.

A (root) query pack actually represents a set of queries. Execution of a query pack aims at finding out which queries of the pack succeed. If a query pack is executed as if the **or**'s were usual disjunctions, backtracking occurs over queries that have already succeeded and too many successes are detected. To avoid this,

some part of the query pack should no longer be considered during backtracking as soon as a query succeeds. The algorithm realises this by reporting success of queries (and of query packs) to points higher up in the query pack.

A (non-root) query pack can be safely removed if all the queries that depend on it (namely all the queries that are below it in the query pack) have succeeded once. For a leaf $\mathcal{Q}$ (empty set of children), success of $conj(\mathcal{Q})$ is sufficient to remove it. For a $\mathcal{Q}$ with dependent queries, we wait until all the dependent queries report success or equivalently until all the query packs in $children(\mathcal{Q})$ report success.

At the start of the evaluation of a root query pack, the set of branches for every query pack in it contains all the branches in the given query pack. During the execution, query packs can be removed from sets and the values of the $children(\mathcal{Q})$ changes accordingly. Thus, when due to backtracking a query pack is executed again, it might be the case that fewer branches have to be considered.

The execution of a query pack $\mathcal{Q}\theta$ is defined by the algorithm $execute_qp(\mathcal{Q}, \theta)$ (Fig. 2) which imposes additional control on the usual Prolog execution.

The usual Prolog execution and backtracking behaviour is modelled by the while loop (line 1) which generates all possible solutions σ for the conjunction in the query pack. If no more solutions are found, fail is returned and backtracking will occur at the level of the calling query pack.

The additional control manages the $children(\mathcal{Q})$. For each solution σ, the necessary branches of $\mathcal{Q}$ will be executed. It is important to notice that the initial set of branches of a query pack is changed destructively during the execution of this algorithm. Firstly, when a leaf is reached, success is returned (line 8) and the corresponding branch is removed from the query pack (line 6). Secondly, when a query pack that initially had several branches, finally ends up with an empty set of branches (line 6), also this query pack is removed (line 8). The fact that branches are destructively removed, implies that when due to backtracking the same query pack is executed again for a different σ, not all branches have to be executed any more. Moreover, by returning success the backtracking over the current query pack conjunction $conj(\mathcal{Q})$ is stopped: all branches have reported success.

3.3 Using Query Packs

Fig. 3 shows an algorithm that makes use of the pack execution mechanism to compute the result set R as defined in our problem statement. From a query pack $\mathcal{Q}$ containing all queries in S, a derived pack $\mathcal{Q}'$ is constructed by adding a `report_success/2` literal to each leaf of the pack; the task of `report_success(K,N)` is simply to add (K, Q_N) to R (with Q_N the N-th query in the pack).[1] $\mathcal{Q}'$ will be executed via the predicate `evaluate_pack/1` which is

[1] In our current implementation the result set is implemented as a bit-matrix indexed on queries and examples. This implementation is feasible (on typical computers at this moment) as long as the number of queries in the pack multiplied by the number of examples is less than 10^9, which holds for most current ILP applications.

```
0    execute_qp( pack Q, substitution θ) {
1    while ( σ ← next_solution( conj(Q)θ)
2        {
3          for each Q_child in children(Q) do
4              {
5                if ( execute_qp( Q_child , σ) == success)
6                    children(Q) ← children(Q) \ {Q_child}
7              }
8          if ( children(Q) is an empty set) return(success)
9        }
10   return(fail)
11   }
```

Fig. 2. The query pack execution algorithm

```
1   evaluate(set of examples E, pack Q, key K) {
2       Q' ← Q;
3       q ← 1;
4       for each leaf of Q' do {
5           add report_success(K, q) to the right of the conjunction in the leaf
6           increment q
7       }
8       C ← (evaluate_pack(K) :- Q');
9       compile_and_load(C);
10      for each example e in E do {
11          k ← key(e);
12          evaluate_pack(k);
13      }
14  }
```

Fig. 3. Using query packs to compute the result set R

dynamically defined, compiled, loaded and then executed. Obviously an ILP system not interested in the result set itself could provide its own `report_success/2` predicate and thus avoid the overhead of explicitly building the result set.

Note that this algorithm follows the strategy of running all queries for each single example before moving on to the next example: this could be called the "examples in outer loop" strategy, as opposed to the "queries in outer loop" strategy used by most ILP systems. The "examples in outer loop" strategy has important advantages when processing large data sets; see, e.g., [12, 5].

3.4 Computational Complexity

Lower and upper bounds on the speedup factor that can be achieved by executing a pack instead of separate queries can be obtained as follows. For a

pack containing n queries $q_i = (a, b_i)$, let T_i be the time needed to compute the first answer substitution of q_i if there are any, or to obtain failure otherwise. Let t_i be the part of T_i spent within a and t'_i the part of T_i spent in b_i. Then $T_s = \sum_i (t_i + t'_i)$ and $T_p = \max(t_i) + \sum_i t'_i$ with T_s and T_p representing the total time needed for executing all queries separately, respectively executing the pack. Introducing $c = \sum_i t_i / \sum_i t'_i$, it is possible to show that $T_s/T_p \geq 1$ and $T_s/T_p \leq (c+1)/(c/n+1) < \min(c+1, n)$. Thus the speedup factor is bound by the branching factor n and by the ratio c of computational complexity in the shared part over the computational complexity of the non-shared part. For multi-level packs, under the assumption of a constant branching factor b, if the computation of literals at depth 1 to i in the pack is dominant then the speedup factor approaches b^{d-i} with d the depth of the pack.

4 Use of Query Pack Execution in ILP Systems

In this section we briefly describe two existing ILP algorithms and show how the above execution method can be included in the algorithms to improve efficiency.

4.1 Refinement of a single rule

The first algorithm we discuss is TILDE[4], an algorithm that builds first-order decision trees. In a first-order decision tree, nodes contain literals that together with the conjunction of the literals in the nodes above this node (i.e., in a path from the root to this node) form the query that is to be run for an example to decide which subtree it should be sorted into. When building the tree, the literal (or conjunction of literals) to be put in one node is chosen as follows: given the query corresponding to a path from the root to this node, generate all refinements of this query (a refinement of a query is formed by adding one or more literals to the query); evaluate these refinements on the data set (computing, e.g., the information gain [14] yielded by the refinement), choose the best refinement, and put the literals that were added to the original clause to form this refinement in the node.

At this point it is clear that a lot of computational redundancy exists if each refinement is evaluated separately. Indeed all refinements contain exactly the same literals except those added during this single refinement step. Organising all refinements into one query pack, we obtain a query pack that essentially has only one level (the root immediately branches into leaves). When TILDE's lookahead facility is used [3], refinements form a lattice and the query pack may contain multiple (though usually few) levels.

Note that the root of these packs may consist of a conjunction of many literals, giving the pack a broom-like form. The more literals in the root of the pack, the greater the benefit of query pack execution is expected to be.

Example 4. Assume the node currently being refined has the following query associated with it: `circle(A,C),leftof(A,C,D),above(A,D,E)`, i.e., the node

covers all examples where there is a circle to the left of some other object which is itself above yet another object.

The query pack generated for this refinement could for instance be

```
                                              / triangle(A,F)
                                              | circle(A,H)
                                              | small(A,I)
                                              | large(A,J)
                                              | in(A,E,K)
                                              | in(A,D,L)
circle(A,C), leftof(A,C,D), above(A,D,E),   <  in(A,C,M)
                                              | above(A,E,N)
                                              | above(A,D,O)
                                              | above(A,C,P)
                                              | leftof(A,E,Q)
                                              | leftof(A,D,R)
                                              \ leftof(A,C,S)
```

When evaluating this pack, all backtracking through the root of the pack (the "stick" of the broom) will happen only once, instead of once for each refinement. In other words: when evaluating queries one by one, for each query the Prolog engine needs to search once again for all objects C, D and E fulfilling the constraint `circle(A,C), leftof(A,C,D), above(A,D,E)`; when executing a pack this search is only done once.

Other Systems Besides Tilde Many systems for inductive logic programming use an algorithm that consists of repeatedly refining clauses. Any of these systems could in principle be rewritten to make use of a query pack evaluation mechanism and thus achieve a significant efficiency gain. Consider, e.g., a system performing an A^* search through a refinement lattice, such as PROGOL [13]. Since A^* imposes a certain order in which clauses will be considered for refinement, it is hard to reorganise the computation at this level. However, when taking one node in the list of open nodes and producing all its refinements, the evaluation of the refinements involves executing all of them; this can be replaced by a pack execution. In principle one could also perform several levels of refinement at this stage, adding all of them to A^*'s queue; part of the efficiency of A^* is then lost, but the pack execution mechanism is exploited to a larger extent. Which of these two effects is dominant will depend on the application: if most of the first-level refinements would be further refined anyway at some point during the search, clearly there will be a gain in executing a two-level pack; otherwise there may be a loss of efficiency. In any case, however, with single-level refinement packs a speedup should certainly be achieved.

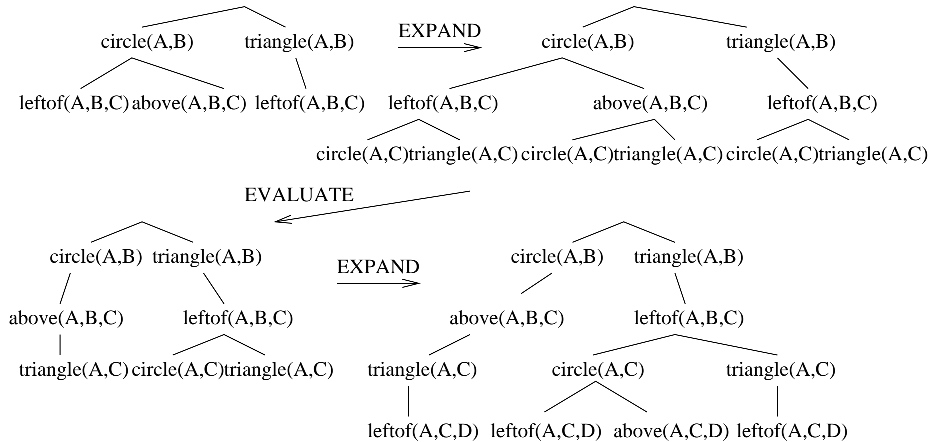

Fig. 4. A sequence of 4 query packs in WARMR. Refinement of the above left query pack results in the 3-level pack above right. Removal of queries found infrequent during pack evaluation results in the bottom left pack. Finally, another level is added in a second query expansion step to produce the bottom right pack. This iteration between expansion and evaluation continues until the pack is empty.

4.2 Level-wise frequent pattern discovery

An alternative family of data mining algorithms scans the refinement lattice in a breadth-first manner for queries whose frequency exceeds some user-defined threshold. The best-known instance of these level-wise algorithms is the APRIORI method for finding frequent item-sets [1]. WARMR [10] is an ILP variant of attribute-value based APRIORI.

Query packs in WARMR correspond to hash-trees of item-sets in APRIORI: both are used to store a subgraph of the total refinement lattice down to level n. The paths from the root down to level $n-1$ in that subgraph correspond to frequent patterns. The paths from root to the leaves at depth n correspond to candidates whose frequency has to be computed. Like hash-trees in APRIORI, query packs in WARMR exploit massive similarity between candidates to make their evaluation more efficient. Essentially the WARMR algorithm starts with an empty query pack and iterates between pack evaluation and pack extension (see Figure 4). The latter is achieved by adding all potentially frequent refinements[2] of all leaves in the pack, i.e., adding another level of the total refinement lattice.

5 Experimental Evaluation

The goal of this experimental evaluation is to empirically investigate the actual speedups that can be obtained by re-implementing ILP systems so that they use

[2] Refinements found to be specialisations of infrequent queries cannot be frequent themselves, and are pruned consequently.

the pack execution mechanism. At this moment such re-implementations exist for the TILDE and WARMR systems, hence we have used these for our experiments. We attempt to quantify both the speedup of packs w.r.t. to separate execution of queries, and the total speedup that this can yield for an ILP system.

5.1 Tilde

Comparison of different implementations of Tilde It is useful to consider four different ways in which TILDE can be run in its ILPROLOG implementation:

1. No packs: the normal implementation of TILDE, where queries are generated one by one and each is evaluated on all relevant examples (original implementation as described in [4]). Since queries are represented as terms, each evaluation of a query involves a meta-call in Prolog.
2. An "Examples in outer loop" implementation of TILDE, where examples are considered one by one and for each example all queries are run one after another (see [5]). Each call of a query, as in the previous setting, involves a meta-call.
3. Disjoint execution of packs: a query pack is executed in which all queries in the pack are put beside one another; i.e., common parts are not shared by the queries. The computational redundancy in executing such a pack is the same as that in executing all queries one after another; the main difference is that in this case all queries are compiled.
4. Packed execution of packs: a query pack is executed where queries share as much as possible.

Of these four settings the second one turned out to be the slowest by far; we attribute this to the fact that queries are not compiled but executed using meta-calls. The most interesting information, however, is obtained by comparing the other three settings:

- The difference between packed execution and disjoint execution accurately indicates the efficiency gain that is obtained by redundancy removal (not blurred by any effects of pre-compilation and other possible implementation differences).
- The difference between the packed execution and the original implementation gives an idea of the net efficiency gain that is obtained, taking into account all implementation changes.

In the remainder of this section we focus on the timings for settings 1,3 and 4, which provide us with the above information.

Next to the total time needed to build a tree, we have also recorded the time needed to compile and load the query packs, and the time needed to execute them (this is excluding the time needed for computing statistics from the result set and retrieving the best query afterwards). To evaluate the net speedup of TILDE due to query pack execution, total times should be compared; when comparing disjoint and packed execution it is better to look at execution times only.

In a first experimental setup we measured times for TILDE on Bongard datasets[3] of different sizes, where no simple classification theory existed for the examples. In a second setup a true classification theory of small size existed for the examples, such that in most cases the same tree is learned; this mostly removes the influence of different tree sizes on induction times. In some cases there is a difference in tree size even for the same settings : different implementations may select different test when multiple tests of the same quality exist, which might give rise to significant differences in tree size (this was observed in earlier implementations of TILDE). We report these tree sizes in our tables. As can be seen in the table, in only one case a small difference in size was observed between trees induced under the same settings.

The size of a pack depends on how many refinements are generated at a certain node; this was varied by setting TILDE's lookahead parameter to 0, 1 or 2 (with 2 generating the largest packs).

Table 1 gives an overview of the experimental results we obtained for the first setup (trees of very different sizes), Table 2 gives a similar overview for the second setup. The total induction time is reported, as well as (for pack-based execution mechanisms) the time needed for pack compilation and pack execution.

Some interesting observations are:

- Comparing total times, net speedup factors of 1 to 4.7 are obtained. The speedup becomes larger when larger packs are generated (higher lookahead setting).
- Comparing execution times of disjoint and packed execution, speedup factors of 1.3 up to 23 are obtained. Again, larger packs yield larger speedups.
- When subtracting the compilation time from the total induction time, it can be seen that disjoint execution is approximately as fast as the original implementation; i.e., the re-implementation from "queries in outer loop" to "examples in outer loop" does not have a net effect. Packed execution, of course, does yield an important gain.
- For each individual setting, the total induction time varies with the number of examples in the same way as for the other settings. This confirms that the re-implementation does not affect the way in which the complexity of the learning algorithm depends on the number of examples.

We have also run experiments on the Mutagenesis data set [15], with somewhat different results, as can be seen in Table 3. Query packs are much larger than for the Bongard data set; with a lookahead of 2 the largest packs had over 12000 queries. For these large packs a significant amount of time is spent compiling the pack, but even then large net speedups are obtained, in one specific case up to a factor of 20. Such high speedups can be obtained in cases where occasionally a computationally complex clause is generated (with many variables and many literals sharing variables) and then refined; a large proportion of the

[3] The "Bongard" data sets contain problems related to those used by M. Bongard [6] for research on pattern recognition and were introduced as an ILP benchmark by [9].

LA	tree size (nodes)	original	disjoint			packed			speedup ratio	
			total	comp	exec	total	comp	exec	*net*	**exec**
592 examples										
0	27	*2.0*	*4.2*	2.26	**0.52**	*1.79*	0.35	**0.12**	*1.1*	**4.3**
1	15	*4.95*	*9.13*	5.06	**1.54**	*3.04*	0.77	**0.21**	*1.6*	**7.3**
2	11	*15.12*	*17.81*	9.12	**4.06**	*6.01*	1.79	**0.18**	*2.5*	**22.6**
1194 examples										
0	58	*6.29*	*11.97*	6.44	**1.92**	*5.1*	0.98	**0.57**	*1.2*	**3.4**
1	54	*24.14*	*43.25*	24.79	**9.41**	*11.92*	3.12	**1.23**	*2.0*	**7.7**
2	22	*56.42*	*99.14*	65.76	**15.44**	*23.46*	8.45	**0.84**	*2.4*	**18.4**
1804 examples										
0	48	*7.89*	*12.97*	6.02	**2.75**	*6.3*	1.13	**1.04**	*1.25*	**2.6**
1	58	*35.67*	*54.57*	27.76	**15.5**	*17.42*	4.46	**3.27**	*2.0*	**4.7**
2	13	*46.17*	*56.69*	27.34	**17.2**	*16.55*	4.87	**1.35**	*2.8*	**12.7**
2986 examples										
0	65	*17.62*	*26.68*	11.39	**7.71**	*13.85*	1.92	**4.40**	*1.27*	**1.8**
1	80	*74.56*	*103.1*	41.49	**41.84**	*33.22*	5.49	**10.6**	*2.2*	**3.9**
2	83	*426.52*	*699.9*	406.6	**201.27**	*140.24*	45.95	**14.43**	*3.0*	**13.9**
6013 examples										
0	129–131	*50.62*	*77.59*	22.52	**34.99**	*50.1*	3.93	**26.69**	*1.01*	**1.3**
1	148	*277.93*	*498.38*	122.15	**319.79**	*177.66*	14.07	**114.49**	*1.56*	**2.8**
2	81	*1600*	*2458*	539.64	**1752**	*386.8*	47.58	**187.66**	*4.1*	**9.3**

Table 1. Statistics of trees built by Tilde in function of the lookahead setting (LA = 0, 1, 2) and the "packs" setting (original = no packs, disjoint execution, packed execution). Reported times are the total time needed to build a tree, the time spent on compilation of packs and the time spent on their execution. Times are measured in seconds, tree sizes in nodes.

total induction time may be spent for this, and it is exactly here that the highest speedup factor can be expected (high c and n, see complexity analysis).

Comparison with other engines Since ILPROLOG is a new Prolog engine, we wanted to compare its performance with existing engines; obviously the efficiency gain achieved through the query pack execution it offers should not be offset by a less efficient implementation of the engine itself.

TILDE was run on the above data using the SICSTUS and MASTERPROLOG engines. The implementation of TILDE for these engines is almost exactly the same as the one for the ILPROLOG engine when the "original" setting (no packs) is used. Table 4 shows some results. It can be seen from the table that at this moment, and on these data, ILPROLOG is less efficient than MASTERPROLOG, however the pack execution mechanism amply compensates for that.

Some remarks are to be made here: a) the Leuven ILP systems were originally implemented in MASTERPROLOG and these implementations make use of some non-standard builtins which for the other Prolog systems have to be simulated, which puts MASTERPROLOG at an advantage when comparing timings; b) ILPROLOG is still under construction and further efficiency improvements in the

LA	tree size (nodes)	original	disjoint total	disjoint comp	disjoint exec	packed total	packed comp	packed exec	speedup ratio *net*	speedup ratio **exec**
521 examples										
0	2	*0.63*	*1.05*	0.43	**0.22**	*0.58*	0.12	**0.06**	*1.09*	**3.7**
1	4	*1.7*	*1.26*	0.24	**0.51**	*0.74*	0.13	**0.07**	*2.30*	**7.3**
2	3	*4.6*	*2.94*	0.49	**1.42**	*1.26*	0.17	**0.12**	*3.65*	**11.8**
1007 examples										
0	7	*1.12*	*1.11*	0.3	**0.32**	*0.73*	0.12	**0.12**	*1.53*	**2.7**
1	6	*4.06*	*3.08*	0.77	**1.31**	*1.59*	0.27	**0.27**	*2.55*	**4.9**
2	4	*12.74*	*7.38*	1.19	**3.86**	*3.05*	0.46	**0.34**	*4.18*	**11.4**
1453 examples										
0	5	*1.44*	*1.28*	0.15	**0.49**	*0.95*	0.06	**0.21**	*1.52*	**2.3**
1	4	*5.25*	*3.48*	0.22	**2.05**	*1.85*	0.1	**0.56**	*2.84*	**3.7**
2	3	*14.33*	*8.83*	0.48	**5.66**	*3.49*	0.23	**0.68**	*4.11*	**8.3**
2473 examples										
0	9	*3.34*	*2.99*	0.38	**1.33**	*2.21*	0.14	**0.75**	*1.51*	**1.8**
1	6	*12.3*	*9.11*	0.72	**6.08**	*4.78*	0.25	**2.14**	*2.57*	**2.8**
2	4	*40.05*	*24.66*	1.36	**17.94**	*8.44*	0.47	**2.64**	*4.75*	**6.8**
4981 examples										
0	13	*8.46*	*7.92*	0.75	**4.18**	*6.2*	0.14	**3.1**	*1.36*	**1.4**
1	6	*35.63*	*29.45*	0.72	**23.76**	*16.63*	0.25	**11.4**	*2.14*	**2.1**
2	4	*116.93*	*84.14*	1.36	**71.41**	*25.43*	0.52	**13.64**	*4.60*	**5.2**

Table 2. Statistics of trees built by Tilde in function of the lookahead setting (LA = 0, 1, 2) and the "packs" setting (no packs, disjoint execution, packed execution). Reported times are the total time needed to build a tree, the time spent on compilation of packs and the time spent on their execution. Times are measured in seconds, tree sizes in nodes.

engine are expected (for instance, its compiler does not produce native code yet, which the other compilers do); c) our experience with these experiments suggests that timings are relatively unstable w.r.t. certain implementation changes, especially with respect to dynamically asserted information; therefore these results (in particular for SICSTUS) have to be taken with a grain of salt.

5.2 Warmr

Used implementations and engines For WARMR we consider the following implementations:

1. No packs: the normal implementation of WARMR, with "queries in outer loop", where queries are generated and evaluated one by one.
2. With packs: An "examples in outer loop" implementation where first all queries for one level are generated and put into a pack, and then this pack is evaluated on each example.

For the "no packs" implementation we also give the execution times for MASTERPROLOG.

LA tree size (nodes)		original	disjoint total	disjoint comp	disjoint exec	packed total	packed comp	packed exec	speedup ratio *net*	speedup ratio **exec**
Regression, 230 examples										
0	12	*5.45*	*7.84*	2.71	**3.04**	*4.35*	0.71	**1.68**	*1.25*	**1.81**
1	12	*22.71*	*36.33*	15.73	**16.6**	*12.18*	4.0	**4.85**	*1.86*	**3.42**
2	13	*239.84*	–	(99.87)	(**124.55**)	*72.06*	35.8	**19.86**	*3.33*	>**6.27**
Classification, 230 examples										
0	18	*6.03*	*9.04*	2.98	**4.05**	*4.51*	0.48	**2.32**	*1.34*	**1.75**
1	17	*28.38*	*42.4*	15.67	**21.67**	*13.6*	2.36	**7.38**	*2.09*	**2.94**
2	24	*2453.46*	–	(160.5)	(**397.41**)	*124.15*	39.29	**39.8**	*19.8*	>**9.99**

Table 3. Timings for Mutagenesis. A – in the table indicates that that run ended prematurely; timings between parentheses indicate times already consumed at that moment.

	SICSTUS	MASTERPROLOG	ILPROLOG(original)	ILPROLOG(packs)
Bongard-592	52.3	10.05	14.87	6.13
Bongard-1194	206	37.5	56.98	24.78
Bongard-1804	217	27.62	46.57	15.06
Bongard-2986	1979	244	429	129
Bongard-6013	8600	651	1586	371
Bongard-521	16.7	3.14	4.49	1.37
Bongard-1007	58.6	8.23	12.7	3.5
Bongard-1453	105	8.91	14.2	3.37
Bongard-2473	228	20.45	39.2	9.33
Bongard-4981	714	43.45	115.4	27.7

Table 4. ILPROLOG compared to other engines (times in seconds)

Datasets We used the Mutagenesis dataset, with a language bias that was chosen so as to generate a limited number of refinements (i.e., a relatively small branching factor in the search lattice); this allows us to generate query packs that are relatively deep but narrow. Table 5 summarises the number of queries for each level.

Results In Table 6 the execution times of WARMR on Mutagenesis are given, with maximal search depth varying from 2 to 5 levels.

These results raised our interest because they show a behaviour that is quite different from the one observed with TILDE. Speedup factors barely increase with increasing depth of the packs, in contrast to TILDE where larger packs yielded higher speedups. At first sight we found this surprising; however it becomes less so when the following observation is made. When refining a pack into a new pack by adding a level, WARMR prunes away branches that lead only to infrequent queries. There are thus two effects when adding a level to a pack: one is the widening of the pack at the lowest level (at least on the first few levels, a new pack typically has more leaves than the previous one), the second is the narrowing of the pack as a whole (because of pruning). Since the speedup obtained by using

Level	Queries	Frequent queries
1	1	1
2	9	6
3	27	20
4	90	72
5	970	897

Mutagenesis

Table 5. Number of queries for the Mutagenesis experiment with WARMR.

Level	With packs		No packs ILPROLOG		No packs MASTERPROLOG		speedup ratio	
	exec	total	exec	total	exec	total	*net*	**exec**
2	0.65	0.93	1.17	1.26	0.37	0.62	*1.35*	**1.80**
3	4.63	6.69	8.24	9.2	1.62	2.77	*1.38*	**1.78**
4	40.14	65.22	94.41	107.04	10.46	24.25	*1.64*	**2.35**
5	345.75	673.75	861.77	1078.09	101.52	311	*1.60*	**2.49**

Table 6. Results on Mutagenesis

packs largely depends on the branching factor of the pack, speedup factors can be expected to decrease when the narrowing effect is stronger than the widening-at-the-bottom effect. We suspect both effects are approximately equally strong here. We had not anticipated this behaviour and think it deserves further investigation.

5.3 Summary of Experimental Results

Our experiments confirm that a) query pack execution in itself is much more efficient than executing many highly similar queries separately; b) existing ILP systems (we use TILDE and WARMR as examples) can use this mechanism to their advantage, achieving significant speedups (up to a factor 20 in our experiments); and c) although a new Prolog engine is needed to achieve this, the current state of development of this engine is already such that more advanced engines that do not have the query pack mechanism cannot compete with it.

In addition, differences between the effect of pack execution on TILDE and WARMR have come to light, prompting further investigation of, e.g., the influence of size and shape of packs on the efficiency gains obtained with them.

6 Related work

The re-implementation of TILDE is related to the work by [12] who were the first to describe the "examples in outer loop" strategy for decision tree induction. The query pack execution mechanism, here described from the Prolog execution point of view, can be seen as a first-order counterpart of APRIORI's mechanism for counting item-sets [1].

The idea of optimising sets of queries instead of individual queries has existed for a while in the database community; Tsur et al. [16] describe an algorithm

for efficient execution of so-called query *flocks* in this context. Like our query pack execution mechanism, the query flock execution mechanism is inspired by Apriori and is set in a deductive database context. The main difference between our query packs and the query flocks described in [16] is that query packs are more hierarchically structured and the queries in a pack are much less similar than the queries in a flock. (A flock is represented by a single query with placeholders for constants, and is equal to the set of all queries that can be obtained by instantiating the placeholders to constants. Flocks could not be used for the applications we consider here.)

7 Conclusions

There is a lot of redundancy in the computations performed by most ILP systems. In this paper we have identified a source of redundancy and proposed a method for avoiding it: execution of query packs, and we have discussed how query pack execution can be incorporated in ILP systems. The query pack execution mechanism itself has been implemented in a new Prolog system called ilProlog dedicated to data mining tasks, and two ILP systems have been re-implemented to make use of the mechanism. We have experimentally evaluated these re-implementations, and the results of these experiments confirm that large speedups may be obtained in this way. We conjecture that the query pack execution mechanism can be incorporated in other ILP systems and that similar speedups can be expected.

The problem setting in which query pack execution was introduced is very general, and allows the technique to be used for any kind of task where many queries are to be executed on the same data, as long as the queries can be organised in a hierarchy.

Future work includes further improvements to the ilProlog engine and the implementation of techniques that will increase the suitability of the engine to handle large data sets. Another interesting issue is how to port the proposed query pack execution mechanism to the context of relational databases. A lot of work has been done in the database community concerning optimisation of single queries, but optimisation of sets of queries is relatively new. In the best case one might hope to combine the efficient bottom-up computations usually performed in databases with a computation sharing mechanism such as the one we here propose. Such a result would significantly increase the efficiency with which large databases can be mined.

Acknowledgements

Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research (FWO) of Flanders. Luc Dehaspe is a post-doctoral fellow of the K.U.Leuven Research Fund. Jan Ramon is funded by the Flemish Institute for the Promotion of Scientific Research in Industry (IWT). Henk Vandecasteele is supported by the FWO-project G.0246.99 “Query languages for database mining”. The authors

would like to thank Luc De Raedt for his influence on this work, and Ashwin Srinivasan for suggesting the term "query packs".

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. 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, 1996.
2. H. Blockeel. *Top-down induction of first order logical decision trees.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/blockeel98:phd.ps.gz`.
3. H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.
4. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
5. H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.
6. M. Bongard. *Pattern Recognition.* Spartan Books, 1970.
7. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996. See also `http://www.cs.sunysb.edu/~sbprolog`.
8. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
9. L. De Raedt and W. Van Laer. Inductive constraint logic. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *Proceedings of the 6th International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.
10. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
11. Bart Demoen, Gerda Janssens, and Henk Vandecasteele. Executing query flocks for ILP. In Sandro Etalle, editor, *Proceedings of the Eleventh Benelux Workshop on Logic Programming*, Maastricht, The Netherlands, November 1999. 14 pages.
12. M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Extending Database Technology*, 1996.
13. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13, 1995.
14. J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann series in machine learning. Morgan Kaufmann, 1993.
15. A. Srinivasan, S.H. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.
16. Dick Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: A generalization of association-rule mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 1–12, New York, June 1–4 1998. ACM Press.