Query Optimization in Inductive Logic Programming by Reordering Literals

Jan Struyf and Hendrik Blockeel

Katholieke Universiteit Leuven, Dept. of Computer Science,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Jan.Struyf, Hendrik.Blockeel}@cs.kuleuven.ac.be

Abstract. Query optimization is used frequently in relational database management systems. Most existing techniques are based on reordering the relational operators, where the most selective operators are executed first. In this work we evaluate a similar approach in the context of Inductive Logic Programming (ILP). There are some important differences between relational database management systems and ILP systems. We describe some of these differences and list the resulting requirements for a reordering transformation suitable for ILP. We propose a transformation that meets these requirements and an algorithm for estimating the computational cost of literals, which is required by the transformation. Our transformation yields a significant improvement in execution time on the Carcinogenesis data set.

1 Introduction

Many Inductive Logic Programming (ILP) systems construct a predictive or descriptive model (hypothesis) for a given data set by searching a large space of candidate hypotheses. Different techniques exist to make this search more efficient. Some techniques improve efficiency by reducing the number of hypotheses that are evaluated during the search. Examples of such techniques are language bias specification mechanisms [10], which limit the number of valid hypotheses, and search strategies (e.g. branch-and-bound search [9], heuristic search [12] and stochastic search [16]), which reduce the number of hypotheses to be evaluated by cutting away branches in the search.

Evaluating a candidate hypothesis can also be made more efficient. In ILP each hypothesis is represented by a number of clauses in first order logic. For example, in an ILP rule learner such as Progol [9] or FOIL [12], the condition of each rule r is a first order logic query q. Rules are generated and evaluated one by one. In order to evaluate the performance of a rule, the query q has to be executed on each example in the data set.

Many techniques designed for efficiently executing relational queries (e.g. SQL) in a relational database management system [4] can, after some modifications, also be used for efficiently executing first order queries. An example of this is index structures (e.g., hash-tables), which are used to efficiently retrieve tuples from a relational database given the value of some attribute. Most ILP systems provide a similar kind of indexing [7, 3] on the datalog facts that are commonly used to describe the examples.

Query optimization [4, 5] is another technique that is frequently used in relational database management systems. Query optimizers transform queries into a different form, which is more efficient to execute. Also in ILP, techniques have been introduced for optimizing first order queries (see [13] for an overview). One specific approach that is popular in relational database management systems is to reorder the relational operators such that the most selective ones are executed first. A similar approach has not yet been evaluated in the context of ILP. In this work we introduce a query transformation that optimizes first order queries by reordering literals. The basic idea is that also first order queries become more efficient to execute if “selective” literals are placed first.

This paper is organized as follows. In Section 2 we present a motivating example that illustrates the amount of efficiency that can be gained by a reordering transformation in ILP. In Section 3 we list three important requirements for such a transformation and point out some significant differences with query optimization in relational databases. In Section 4 we introduce a reordering transformation for optimizing first order queries in ILP. We also describe an extension able to handle queries that are the output of the *cut-transform* [13]. Section 5 presents experiments that evaluate the performance of the transformation on the Carcinogenesis data set. Section 6 concludes the paper.

2 A Motivating Example

We start with a motivating example that illustrates the amount of efficiency that can be gained by reordering the literals of a first order query. In this example we use the Carcinogenesis data set [14], which stores structural information about 330 molecules. Each molecule is described by a number of `atom/4`[1] and `bond/4` facts.

Consider the following query:

```
atom(M,A1,h,3),atom(M,A2,c,16),bond(M,A2,A1,1).
```

This query succeeds for a molecule `M` that contains a hydrogen atom `A1` of type 3 that is bound to a carbon atom `A2` of type 16 by a single bond. We execute each possible reordering (i.e., permutation) of the query on each example molecule. Table 1 lists all permutations sorted by average execution time.

As can be seen from Table 1, the most efficient permutation is 6 times faster than the original query q_4 and 9 times faster than the least efficient one. This can be explained by looking at the size of the SLD-tree [8] that is generated when the ILP system executes the permutation on a given example.

Given a query $q = l_1 \ldots l_n$ and example e, the SLD-tree will contain at level i different nodes $v_{i,j}$ that correspond to calls to literal l_i with input substitution $\theta_{i,j}$. The input substitution for the root node $\theta_{1,1}$ binds the key variable of the

[1] We have omitted the atom charge to simplify the presentation.

Table 1. Averaged execution times (ms/example) for the 6 permutations of an example query. The queries were executed using the YAP Prolog system version 4.4.2 on an Intel P4 1.8GHz with 512MB RAM running Linux.

	Time	Permutation
q_1	0.010	`atom(M,A2,c,16),bond(M,A2,A1,1),atom(M,A1,h,3).`
q_2	0.023	`atom(M,A2,c,16),atom(M,A1,h,3),bond(M,A2,A1,1).`
q_3	0.052	`bond(M,A2,A1,1),atom(M,A2,c,16),atom(M,A1,h,3).`
q_4	0.061	`atom(M,A1,h,3),atom(M,A2,c,16),bond(M,A2,A1,1).`
q_5	0.081	`atom(M,A1,h,3),bond(M,A2,A1,1),atom(M,A2,c,16).`
q_6	0.091	`bond(M,A2,A1,1),atom(M,A1,h,3),atom(M,A2,c,16).`

query (`M` in the example) to the identifier of example e. The children of a node $v_{i,j}$ correspond to the solutions of $l_i\theta_{i,j}$. The size of the SLD-tree thus depends on the length of the query, which is the maximum depth of the tree, and the number of children of each node.

We will call this number of children the non-determinacy of a literal given the corresponding input substitution (Definition 1). Because the non-determinacy of a literal depends on the input substitution, it also depends on the position of the literal in the SLD-tree and hence also on its position in the query.

Definition 1. *(Non-determinacy) The non-determinacy* $\text{nondet}(l,\theta)$ *of a literal l given an input substitution θ is the number of solutions or answer substitutions that are obtained by calling $l\theta$.*

Example 1. We continue the Carcinogenesis example. The non-determinacy of `atom(M,A1,h,3)` given `{M/m1}` is equal to the number of hydrogen atoms of type 3 in molecule `m1`. If we consider `{M/m1,A1/a1}` then the non-determinacy is either 1 if `a1` is a hydrogen atom of type 3 or 0 if it is another kind of atom.

Non-determinacy as defined above is always associated with one specific call, i.e., with one literal and input substitution. The notion of average non-determinacy (Definition 2) is useful to make more general statements about the number of solutions of a literal.

Definition 2. *(Average non-determinacy) Consider a distribution over possible input substitutions $\mathcal{D}$. We define the average non-determinacy of a literal l with respect to $\mathcal{D}$ as:*

$$\overline{\text{nondet}}(l,\mathcal{D}) = \sum_{(\theta,p)_k \in \mathcal{D}} p_k \cdot \text{nondet}(l,\theta_k)$$

with p_k the probability of θ_k given $\mathcal{D}$.

Example 2. Consider all N molecules in the Carcinogenesis domain and let $\mathcal{D}$ be a uniform distribution over the input substitutions that ground the key argument of a literal to one of the molecule identifiers. The average non-determinacy of `bond(M,A1,A2,N)` with respect to $\mathcal{D}$ is about 20, which is the average number of bonds that occurs in a molecule. Whenever $\mathcal{D}$ is not specified in the future it refers to the same distribution as is defined in this example.

Consider again the fastest (q_1) and slowest (q_6) permutation from Table 1. In permutation q_1 the average non-determinacy of l_1 is low because few molecules contain carbon atoms of type 16. The non-determinacy of l_2, given an input substitution, is at most 4 because carbon atoms can have at most 4 bonds. The non-determinacy of l_3 is either zero or one because `A1` is ground at that point. Since only few molecules contain carbon atoms of type 16, the average SLD-tree size will be small for q_1. In permutation q_6 the `bond/4` literal is executed first. Because both `A1` and `A2` are free at that point, the non-determinacy of this call will be equal to the number of bonds in the molecule (about 20 on average). The non-determinacies of l_2 and l_3 will be either zero or one. The SLD-trees for q_6 will be much bigger on average and hence the execution time, which is proportional to the average SLD-tree size, will also be longer.

3 Requirements

We start by listing a number of requirements for a reordering transformation for first order queries in the context of ILP.

R1 (Correctness) The reordering transformation should be correct, i.e. the transformed query should succeed (fail) for the same examples as the original query succeeds (fails).

R2 (Optimality) The reordering transformation should approximate the optimal transformation, which replaces the original query by the permutation that has the shortest average execution time.

R3 (Efficiency) The reordering transformation itself should be efficient. The time for performing the transformation should be smaller than the efficiency difference that can be obtained by executing the transformed query instead of the original one.

If all predicates are defined by sets of facts, then the order of the literals does not influence the result of the query (cf. the switching lemma [8]) and the correctness requirement (R1) is met. If some predicates have input arguments that should be ground (e.g., background predicates that perform a certain computation), then it is possible that a given permutation is invalid because it breaks this constraint. The reordering transformation must make sure not to select such a permutation.

The efficiency requirement (R3) can be easily met in the context of a relational database where executing a query typically involves (slow) disk access. In ILP, query execution is much faster. In many applications the entire data set resides in main memory. Even if the data set does not fit completely in main memory at once, ILP systems can still use efficient caching mechanisms that load examples one by one [1, 15] to speed up query execution. When queries are executed fast, transformation time becomes more important. ILP systems also consider different subsets of the data during the refinement process. Some queries are executed on very small sets of examples. If a query is executed on a smaller set, it will be executed faster, which again makes R3 more difficult to achieve.

To meet the optimality requirement (R2), query optimizers in relational database management systems place selection operators with a high selectivity, joins that are expected to return a small number of tuples, and projections that select few attributes, as early as possible. Similarly, a reordering transformation for first order queries should place literals with a low average non-determinacy first in order to avoid unnecessary backtracking.

An important difference between relational database management systems and ILP systems is that relational database management systems execute queries bottom-up instead of top-down. The user is always interested in obtaining all solutions for a given query. In ILP, query execution stops after finding the first solution (success). This is because the ILP system only needs to know for a given example whether a given query succeeds or not. In this context, a top-down execution strategy is more efficient.

A consequence of the top-down strategy is that a query optimizer for first order queries should place literals that ground many variables as early as possible. Grounding variables decreases the non-determinacy of the literals that follow, which decreases execution time.

Typical for ILP applications are predicates that have a long execution time (e.g., background predicates that compute aggregates). Such predicates should be placed after fast predicates with a low non-determinacy and before predicates with a high non-determinacy. Due to the top-down strategy, the expensive predicate is not executed if the low non-determinacy predicate fails and may be executed several times if the high non-determinacy predicate has several solutions for a given example.

A last difference is that in relational database systems queries are mostly generated directly by humans and not by a refinement operator ρ as is the case for ILP systems. In ILP systems the efficiency of the original query (as it is generated by ρ) depends (much) on the definition of ρ and on the language bias specification that is used.

4 Reordering Transformation

4.1 Reordering Dependent Literals

In this section we introduce a possible reordering transformation for first order queries suitable to be implemented in ILP systems. We will introduce two versions of this transformation. In this section we describe the first version T_1. The second version, which extends T_1, will be discussed in Section 4.2.

Given a query q, the optimal reordering transformation T_1 should return the permutation $T_1(q)$ that has the shortest average execution time. More formally, we can define T_1 as follows ($\overline{\mathrm{cost}}(q)$ represents the average execution time of q and $\mathrm{perms}(q)$ the set of all permutations of q).

$$\mathrm{T}_1(q) = \mathrm{argmin}_{q_k \in \mathrm{perms}(q)}\ \overline{\mathrm{cost}}(q_k)$$

Of course, $\overline{\mathrm{cost}}(q_k)$ can not be computed without executing permutation q_k on all examples. Therefore, we will try to approximate T_1 by replacing the average execution time by an estimate. We call the resulting approximate transformation $\widehat{\mathrm{T}_1}$.

$$\widehat{\mathrm{T}_1}(q) = \mathrm{argmin}_{q_k \in \mathrm{perms}(q)}\ \widehat{\mathrm{cost}}(q_k)$$

We will now explain how the estimated average execution time $\widehat{\mathrm{cost}}(q)$ of a query q can be computed.

The average execution time of a query q depends on the individual execution times of its literals. The execution time and average execution time of a literal can be defined in the same way as non-determinacy and average non-determinacy where defined in Section 2.

Definition 3. *(Cost) The execution time* $\mathrm{cost}(l, \theta)$ *of a literal l given an input substitution θ is defined as the time necessary to compute all solutions to the call $l\theta$ (i.e., the execution time of the Prolog query '*`?-` $l\theta, \mathrm{fail.}$*').*

Definition 4. *(Average cost) Consider a distribution over possible input substitutions $\mathcal{D}$. We define the average execution time of a literal l with respect to $\mathcal{D}$ as:*

$$\overline{\mathrm{cost}}(l, \mathcal{D}) = \sum_{(\theta,p)_k \in \mathcal{D}} p_k \cdot \mathrm{cost}(l, \theta_k)$$

with p_k the probability of θ_k given $\mathcal{D}$.

Using the definitions of average non-determinacy and average execution time of literals we can compute the average execution time of a query q as follows.

$$\overline{\mathrm{cost}}(q) = \sum_{i=1}^{n} w_i \cdot \overline{\mathrm{cost}}(l_i, \mathcal{D}_i) \tag{1}$$

$$w_1 = 1.0; \qquad w_i = \overline{\prod_{j=1}^{i-1} \mathrm{nondet}(l_j, \theta_j)} \;\approx\; \prod_{j=1}^{i-1} \overline{\mathrm{nondet}}(l_j, \mathcal{D}_j),\ i \geq 2$$

Equation (1) computes the average execution time by summing the average execution times of all individual literals. Each term is weighted with a weight w_i which is the average number of calls to this literal in a SLD-tree generated by the ILP system. The correct value of w_i is the average of the product of the non-determinacies of the preceding literals. For transformation $\widehat{\mathrm{T}_1}$ we will approximate this value by the product of average non-determinacies as shown above.

Each literal in a given query will be called with different instantiations for its arguments (one for each occurrence of the literal in the SLD-tree for a given example). We average over these different instantiations by considering for each literal l_i, a distribution of input substitutions $\mathcal{D}_i$.

Example 3. Consider the query $q = l_1(X), l_2(X)$ and suppose that l_1 has non-determinacy 2 and execution time 2ms. Literal l_2 has average non-determinacy 0.5 and average execution time 1ms with respect to $\mathcal{D}_2$. Distribution $\mathcal{D}_2$ is a distribution over the answer substitutions for X of l_1. The average execution time of q is $\overline{\text{cost}}(q) = 1 \times 2\text{ms} + 2 \times 1\text{ms} = 4\text{ms}$ because l_1 will be called once, which takes 2ms, and l_2 will be called twice (once for each solution to l_1), which also takes 2ms.

In order to perform $\widehat{\text{T}_1}(q)$ we need to estimate the average execution time of each permutation q_k. We compute $\widehat{\text{cost}}(q_k)$ with (1) where the average non-determinacy and execution time are replaced by estimates, which we compute based on the data.

In practice, it is not feasible to obtain for each literal in each query an estimate for the corresponding distribution $\mathcal{D}_i$ and associated $\overline{\text{cost}}(l_i, \mathcal{D}_i)$ and $\overline{\text{nondet}}(l_i, \mathcal{D}_i)$. Therefore, we will use an approximate uniform distribution to estimate average cost and non-determinacy. An algorithm for constructing such an approximate distribution is discussed in Section 4.3. In order to select the appropriate approximate distribution we will use the following constraint that holds on $\mathcal{D}_i$ and that must also hold for reasonable approximate distributions.

Each input substitution in $\mathcal{D}_i$ must assign a value to the input variables of l_i that are grounded by calls to preceding literals. Consider again $q = l_1(X), l_2(X)$, and assume that l_1 grounds X. Then all input substitutions in $\mathcal{D}_2$ must also ground X. More formally,

$$\forall (\theta, p)_k \in \mathcal{D}_i : \text{vars}(\theta_k) \supseteq \left(\text{vars}(l_i) \;\cap\; \bigcup_{j=1}^{i-1} \text{groundvars}(l_j) \right)$$

with p_k the probability of θ_k given $\mathcal{D}_i$, $\text{vars}(\theta_k)$ the variables grounded by input substitution θ_k, $\text{vars}(l_i)$ the set of (input) variables of l_i and $\text{groundvars}(l_i)$ the set of variables that is grounded by calling l_i.

Note that (1) computes the execution time for generating a complete SLD-tree, which may have several succeeding paths. As said before, most ILP systems stop execution after finding the first success. Having said that, we will (for simplicity reasons) still use (1) to estimate the average execution time of the different permutations.

In order to meet R1 (correctness), the transformation must rule out queries that call background predicates p with free variables for arguments that are required to be ground. This is accomplished by setting $\widehat{\text{cost}}(p, \mathcal{D}) = \infty$ for distributions $\mathcal{D}$ that do not ground all these variables.

Because the transformation requires computing all permutations of q, it has an exponential time complexity in the number of literals n. For large values of

n it may become difficult to meet requirement R3 with such a transformation. However, in our experiments, where query lengths varied from 1 to 6, this was not a problem.

4.2 Reordering Sets of Literals

Some queries can be split in several independent sets of literals [13, 6]. Consider for example the following query.

```
atm(M,A1,n,38),bond(M,A1,A2,2),atm(M,A3,o,40).
```

This query succeeds if a given molecule contains a nitrogen atom of type 38 with one double bond and an oxygen atom of type 40. If l_1 and l_2 succeed, but l_3 fails, then alternative solutions for l_1 and l_2 will be tried. However, this is useless if l_3 does not depend on l_1 and l_2, i.e. if it does not share variables with l_1 and l_2. The *cut-transformation* proposed in [13] avoids this unnecessary backtracking, by putting cuts between the independent sets of literals.

```
atm(M,A1,n,38),bond(M,A1,A2,2),!,atm(M,A3,o,40).
```

If S_1 and S_2 are independent sets of literals, then the execution time of S_2 will not depend on the execution time of S_1. This implies that S_1 and S_2 can be transformed by our reordering transformation $\widehat{T_1}$ independently. Once S_1 and S_2 have been transformed the question remains whether we should execute S_1`,!,`S_2 or S_2`,!,`S_1. The efficiency of both permutations of the sets may be different because the set after the cut does not need to be executed if the first set fails for a given example. The following equation defines transformation $\widehat{T_2}$, which extends $\widehat{T_1}$, for reordering a query that is a conjunction of independent sets separated by cuts.

$$\widehat{\mathrm{T}_2}(S_1, \texttt{!}, S_2, \texttt{!}, \ldots, S_m) = \mathrm{argmin}_{q_k \in \mathrm{perms}(S_1, \texttt{!}, S_2, \texttt{!}, \ldots, S_m)} \widehat{\mathrm{cost}}(q_k)$$

We approximate the average execution time of $S_1, \texttt{!}, S_2, \texttt{!}, \ldots, S_m$ as follows.

$$\overline{\mathrm{cost}}(S_1, \texttt{!}, S_2, \texttt{!}, \ldots, S_m) = \sum_{i=1}^{m} w_i \cdot \overline{\mathrm{cost}}(S_i)$$

$$w_1 = 1.0; \qquad w_i = \prod_{j=1}^{i-1} \mathrm{P}_{\mathrm{succeeds}}(S_j),\ i \geq 2$$

with w_i the probability that S_i will be executed and $\mathrm{P}_{\mathrm{succeeds}}(S_i)$ the probability that S_i succeeds. We estimate $\mathrm{P}_{\mathrm{succeeds}}(S_i)$ in the following, rather ad-hoc way.

$$\mathrm{P}_{\mathrm{succeeds}}(S) = \min\left(1.0\ ,\ \prod_{l_i \in S} \overline{\mathrm{nondet}}(l_i, \mathcal{D}_i)\right) \qquad (2)$$

The motivation for (2) is that if a set has a low non-determinacy (e.g., if it has on average 0.3 solutions) then we can use this value as an estimate for $\mathrm{P}_{\mathrm{succeeds}}(S_i)$. If the non-determinacy is high (e.g., if the set has on average 5 solutions) then we assign a probability of 1.0.

In order to transform a query that contains several independent sets of literals, we first transform each of these sets separately using $\widehat{T_1}$ and after that reorder the sets using $\widehat{T_2}$. Both $\widehat{T_1}$ and $\widehat{T_2}$ require estimates for $\overline{\mathrm{nondet}}(l_i, \mathcal{D}_i)$ and $\overline{\mathrm{cost}}(l_i, \mathcal{D}_i)$. We will show how these can be obtained from the training examples in the following section.

4.3 Estimating the Cost and Non-determinacy of a Literal

The average non-determinacy and execution time of a literal depend both on the constants that occur as arguments of the literal and on the relevant distribution of input substitutions $\mathcal{D}$. In this section we describe an algorithm that automatically estimates average non-determinacy and execution time for each possible combination of constants and for different uniform distributions $\mathcal{D}$ (with different groundness constraints for the arguments). The algorithm takes as input a set of predicate type definitions. Consider the following type definitions from the Carcinogenesis example.

```
type(atom(mol:key,atom:any,element:const,atype:const,charge:any).
type(bond(mol:key,atom:any,atom:any,btype:const).
```

Each argument of a type definition is of the form a:b with a the type of the argument and b its tag. Arguments marked with tag `key` are grounded to the example key by each substitution in $\mathcal{D}$. Arguments marked with `const` will always occur with one of the possible constants for the corresponding type filled in. Arguments with tag `any` can be free or can be ground depending on $\mathcal{D}$. The algorithm works as follows.

1. For each type definition $\mathrm{typedef}_k$, collect in a set C_k the possible combinations of constant values that occur in the data for the arguments marked with `const`.
2. For each type type_i that occurs with tag `any`, collect for each example e_j in a set $D_{i,j}$ the constants that occur for arguments of type_i in e_j for any of the predicates that has at least one such argument.
3. Run for each type definition $\mathrm{typedef}_k$ the algorithm shown in Fig. 1.

The algorithm from Fig. 1 computes for a given type definition, for each possible combination of constants C and for different input substitution distributions $\mathcal{D}$ the average execution time and non-determinacy with respect to $\mathcal{D}$ for a literal l with name and arity as given by the type definition and the constants from C filled in.

The first loop of the algorithm is over the different combinations of constants and the second loop over the different distributions. For each distribution, a different subset I of the arguments with tag `any` will be ground. For the

```
function compute_avg_nondet_and_cost(typedef$_k$)
$I_k$ = set of all arguments from typedef$_k$ that have tag any
for each $C \in C_k$
    for each $I \in 2^{I_k}$
        create literal $l$ (name/arity as in typedef$_k$, constants from $C$ filled in)
        cost = 0; nondet = 0; count = 0
        for each example $e_j$
            $\theta_j$ = substitution that replaces key variable by identifier $e_j$
            if $I = \emptyset$ then
                nondet = nondet + nondet$(l, \theta_j)$; cost = cost + cost$(l, \theta_j)$
                count = count + 1
            else
                $\mathcal{P}$ = the set of all possible combinations of constants for the
                    attributes in $I$ (constructed using $D_{i,j}$)
                for each combination of constants $P \in \mathcal{P}$
                    $\sigma = \theta_j \cup$ substitution for each constant in $P$
                    nondet = nondet + nondet$(l, \sigma)$; cost = cost + cost$(l, \sigma)$
                    count = count + 1
        $\widehat{\text{cost}}[\text{typedef}_k, \mathcal{D}^{(I)}, C]$ = cost / count
        $\widehat{\text{nondet}}[\text{typedef}_k, \mathcal{D}^{(I)}, C]$ = nondet / count
```

Fig. 1. An algorithm for estimating average non-determinacy and execution time.

`bond(M,A1,A1,T)` type definition, there are two arguments with tag `any`. The first distribution $\mathcal{D}^{(1)}$ that the algorithm considers will leave both atom arguments free. Consecutive iterations will consider $\mathcal{D}^{(2)}$ with `A1` ground, $\mathcal{D}^{(3)}$ with `A2` ground and finally for $\mathcal{D}^{(4)}$ with both `A1` and `A2` ground.

The part inside the double loop computes for literal l the average execution time and non-determinacy with respect to $\mathcal{D}^{(I)}$. The first instructions construct l based on the type definition and constants in C. If all arguments with tag `any` are free (i.e., $I = \emptyset$), then $\mathcal{D}^{(I)}$ becomes a distribution of substitutions that unify the key argument with one of the example identifiers. The average cost and non-determinacy for such a distribution can be estimated by averaging over all examples e_j (cf. the then branch in Fig 1). If some of the arguments with tag `any` are ground in $\mathcal{D}^{(I)}$, then constructing a uniform distribution over possible input substitutions is more complex (cf. the else branch in Fig 1). In that case, the averages have to be computed over each possible combination of relevant constants (from $D_{i,j}$).

Example 4. We will illustrate the algorithm for the `bond/4` type definition. The first step is to collect constants in C_k. For `bond/4`, there is only one argument tagged `const`: the bondtype. The corresponding constants are $\{1, 2, 3, 7\}$ (7 representing an aromatic bond). Type `atom` is the only type tagged `any`. We collect for each example e_j in set $D_{\text{atom},j}$ all constants of type `atom` that occur, i.e., for each example the set of atom identifiers. The next step is to run the algorithm of Fig. 1. It will compute the average non-determinacy and execution time for bond/4 with different constant combinations from C_k filled in and for different subsets of its two tag any arguments ground. A subset of the results is shown in Table 2.

Table 2. Average non-determinacy and execution time for bond(M,A1,A2,T) with different constant combinations filled in and for different subsets of its two tag any arguments ground. The results were obtained with YAP Prolog version 4.4.2. Only the relative values of the costs are important.

A1	A2	T	$\overline{\text{nondet}}(l, \mathcal{D})$	$\overline{\text{cost}}(l, \mathcal{D})$
free	free	1	20.4	0.0081
free	ground	1	0.74	0.0034
ground	free	1	0.74	0.0016
ground	ground	1	0.017	0.0022
free	free	2	1.36	0.0074
...	...	...	...	...

Note that our algorithm uses uniform distributions over the possible identifiers to estimate average non-determinacy and execution time. If a literal occurs somewhere in the middle of a conjunction, the distribution over input substitutions will probably not be uniform. Using the estimates based on uniform distributions is an approximation.

The computational complexity of the algorithm is $O(2^{|I_k|}|C_k|)$, i.e., exponential in the number of variables that occur in a type definition. In practice, the arity of predicates is usually relatively small, so that this approach is not problematic. Moreover, the computation needs to happen only once, not once for each query to be transformed; so the requirement that the query transformation must be efficiently computable, remains fulfilled.

In those cases where the complexity of the algorithm is nevertheless prohibitive, other techniques for estimating the selectivity of literals could be used instead of this straightforward method; see for instance [11].

5 Experimental Evaluation

5.1 Aims

The aim of our experiments is to obtain more insight in the behavior of the reordering transformation, both with and without the reordering of independent sets.

A first experiment compares for a given query q the efficiency of $\widehat{T_1}(q)$ and $\widehat{T_2}(q)$ with the efficiency of the original query and that of the most efficient permutations $T_1(q)$ and $T_2(q)$. It also provides more insight in the influence of the query length on the obtainable efficiency gain. A second experiment compares the efficiency of running the ILP system Aleph without a reordering transformation with its efficiency when using $\widehat{T_1}$ and $\widehat{T_2}$. In each experiment we use the average non-determinacies and execution times estimated as described in Section 4.3 to perform $\widehat{T_1}$ and $\widehat{T_2}$ (see also Table 2).

5.2 Materials

All experiments are performed on the Carcinogenesis data set [14], which is the same data set that has been used as running example throughout this paper. As Prolog system we use YAP Prolog[2] version 4.4.2 on an Intel P4 1.8GHz system with 512MB RAM running Linux. We also use the ILP system Aleph[3] version 4 (13 Nov 2002) which implements a version of Progol [9] and runs under YAP. Aleph uses the *cut* and *theta* transformations as defined in [13]. The language bias we use in the experiments only contains modes for the `atom` and `bond` predicates and is defined as follows.

```
:- mode(*,bond(+drug,+atomid,-atomid,#integer)).
:- mode(*,atom(+drug,-atomid,#element,#integer,-charge)).
:- set(clauselength,7), set(nodes,50000).
```

We have implemented both transformations $\widehat{T_1}$ and $\widehat{T_2}$ as an add-on library for Aleph. This library (written in C++) is available from the authors upon request.

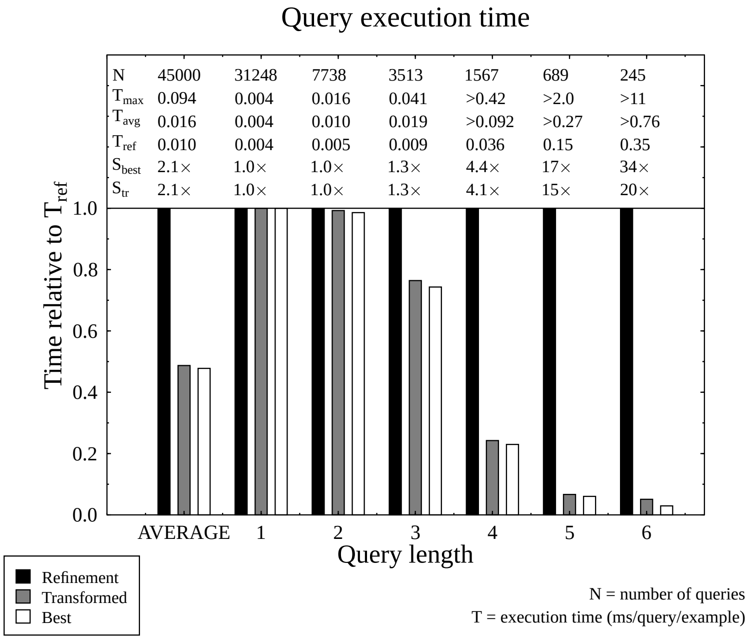
5.3 Experiment 1

$\widehat{T_1}$ In this experiment we have sampled 45000 sets of linked literals (i.e., sets that are separated by cuts in the queries, after they have been processed by the *cut-transform*) uniformly from the queries that occur during a run of Aleph on the Carcinogenesis data set. For each permutation of each of these queries we estimated the average execution time over all 330 molecules. Figure 2 presents the results. The first bar represents the average execution time of the original queries q that are generated by the refinement operator of Aleph, the second bar is the average execution time of the permutations that are selected by $\widehat{T_1}(q)$ and the last bar the average execution time of the most efficient permutations $T_1(q)$. Above each set of bars, N is the number of queries, $\mathrm{T}_{\mathrm{max}}$ the average execution time of the slowest permutations, $\mathrm{T}_{\mathrm{avg}}$ the average of the average execution time over all permutations, $\mathrm{T}_{\mathrm{ref}}$ the average execution time of the original queries, $\mathrm{S}_{\mathrm{best}}$ the efficiency gain of $T_1(q)$ over the original query and S_{tr} the efficiency gain of $\widehat{T_1}(q)$. In some cases $\mathrm{T}_{\mathrm{max}}$ and $\mathrm{T}_{\mathrm{avg}}$ are marked with a >-sign. This occurs because we have set a maximum execution time of 4s for each permutation (in order to make the experiment feasible). When this time is exceeded the execution of the permutation is aborted and its execution time is

[2] http://yap.sourceforge.net

[3] http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/

excluded from the averages. All execution times are averaged over all queries and examples (the first set of bars) and also averaged over all examples and queries with a given length (the other bars). The averages shown for the long queries have a rather large variance.



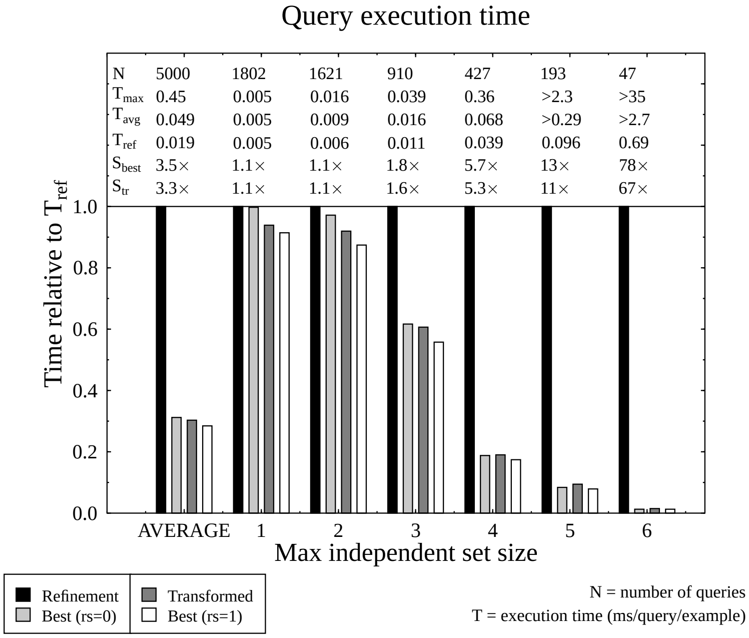
N	45000	31248	7738	3513	1567	689	245
$\mathrm{T}_{\mathrm{max}}$	0.094	0.004	0.016	0.041	>0.42	>2.0	>11
$\mathrm{T}_{\mathrm{avg}}$	0.016	0.004	0.010	0.019	>0.092	>0.27	>0.76
$\mathrm{T}_{\mathrm{ref}}$	0.010	0.004	0.005	0.009	0.036	0.15	0.35
$\mathrm{S}_{\mathrm{best}}$	2.1×	1.0×	1.0×	1.3×	4.4×	17×	34×
S_{tr}	2.1×	1.0×	1.0×	1.3×	4.1×	15×	20×

Fig. 2. Experiment on independent sets of literals.

The main conclusion from this experiment is that $\widehat{T_1}$ almost always succeeds in selecting the most efficient permutation or a permutation with execution time close to that of the most efficient one (because S_{tr} is close to $\mathrm{S}_{\mathrm{best}}$). The difference is larger for longer queries, probably because the approximations made for estimating the cost a query are less accurate in that case. Surprisingly there is almost no gain possible for queries of length 2. This is because in that case the refinement operator of Aleph always generates the most efficient permutation (if compared to $\mathrm{T}_{\mathrm{avg}}$ it is possible to gain a factor of 2 in efficiency). Aleph always generates `atom,bond` instead of `bond,atom`. The second one is less efficient than the first one because the highly non-determinate `bond` literal comes first. If we look at longer queries we see that the transformation yields higher gains (up to 20× for queries of length 6). Comparing $\mathrm{T}_{\mathrm{ref}}$ with $\mathrm{T}_{\mathrm{avg}}$ shows that the original queries are already very efficient. This would probably not be the case anymore if we would use a more naively defined language bias (e.g. one that allows adding bonds with the two atom identifiers free): $\widehat{T_1}$ would yield much higher gains.

Note also that Aleph generates much more small independent sets than larger ones, which also reduces the average gain. This again depends on the language bias that is used.

$\widehat{\boldsymbol{T_2}}$ In this experiment we have sampled 5000 queries uniformly from the queries that occur during a run of Aleph. Each of these queries can contain up to 6 independent sets of linked literals. We estimated for each permutation of the sets and for each permutation of the literals in each set the average execution time over all examples. Figure 3 presents the results. The first bar again corresponds to the execution time of the original query q, the second bar to that of the best possible permutation without reordering the sets (i.e., $T_1(q)$), the third bar to that of the permutation returned by $\widehat{T_2}$ and the last one to that of $T_2(q)$. The other quantities are defined similarly to the previous experiment. One important difference is that here the sets of bars are averaged over queries with a given maximum independent set size (1 - 6).



N	5000	1802	1621	910	427	193	47
T_{max}	0.45	0.005	0.016	0.039	0.36	>2.3	>35
T_{avg}	0.049	0.005	0.009	0.016	0.068	>0.29	>2.7
T_{ref}	0.019	0.005	0.006	0.011	0.039	0.096	0.69
S_{best}	3.5×	1.1×	1.1×	1.8×	5.7×	13×	78×
S_{tr}	3.3×	1.1×	1.1×	1.6×	5.3×	11×	67×

N = number of queries
T = execution time (ms/query/example)

Fig. 3. Experiment on entire clauses.

The main conclusion is that not much efficiency can be gained by reordering the sets. This is expected because reordering sets does not alter the average non-determinacy of literals. Contrary to the previous experiment some gain can be obtained for queries that have 6 independent literals and queries where all independent sets have at most two literals. Transformation $\widehat{T_2}$ performs reasonably well for a low maximum set size and less good for queries with larger independent sets. Note that, to simplify the presentation, Fig. 3 does not show results for $\widehat{T_1}$.

5.4 Experiment 2

In this experiment we compare the runtime of Aleph without any reordering transformation, with its runtime when using $\widehat{T_1}$ and $\widehat{T_2}$. Table 3 presents the results for the original Carcinogenesis data set and Table 4 for an over-sampled version in which each molecule occurs 5 times. The total execution time is split up in three components: the query execution time $\mathrm{T}_{\mathrm{exec.}}$, the time used for performing the reordering transformation $\mathrm{T}_{\mathrm{trans.}}$ and a term $\mathrm{T}_{\mathrm{other}}$ which includes, for example, the time for computing the bottom-clauses and the time taken by the refinement operator.

Table 3. Runtime of Aleph (in seconds) on the Carcinogenesis data set.

Transformation	$\mathrm{T}_{\mathrm{exec.}}$	$\mathrm{T}_{\mathrm{trans.}}$	$\mathrm{T}_{\mathrm{other}}$	$\mathrm{S}_{\mathrm{exec.}}$	$\mathrm{S}_{\mathrm{total}}$
none	2230	/	3900	/	/
$\widehat{T_1}$	1330	100	3920	1.7	1.15
$\widehat{T_2}$	1310	110	3850	1.7	1.16

Table 4. Runtime of Aleph (in seconds) on a 5× over-sampled version of the Carcinogenesis data set.

Transformation	$\mathrm{T}_{\mathrm{exec.}}$	$\mathrm{T}_{\mathrm{trans.}}$	$\mathrm{T}_{\mathrm{other}}$	$\mathrm{S}_{\mathrm{exec.}}$	$\mathrm{S}_{\mathrm{total}}$
none	22720	/	14530	/	/
$\widehat{T_1}$	13720	320	14550	1.7	1.3
$\widehat{T_2}$	13690	370	14570	1.7	1.3

Table 3 and Table 4 both show an efficiency gain in query execution time of 1.7 times. This is less than the average gain of 3.5 times that was obtained in the previous experiment (Fig. 3). The difference is that in that case, each query was executed on the entire data set. Aleph considers different subsets of the data in each refinement step: longer queries are executed on a smaller subset. Because longer queries yield higher gains, the average gain drops. Comparing $\mathrm{T}_{\mathrm{trans.}}$ with $\mathrm{T}_{\mathrm{exec.}}$ shows that the transformation introduces no significant overhead (7.5% on the original data set and 2.3% on the over-samples version).

Recall from the previous experiment (Fig. 2 and Fig. 3) that the efficiency gain obtained with $\widehat{T_1}$ and $\widehat{T_2}$ is close to the best possible efficiency gain (i.e., that of T_1 an T_2). This means that the rather low gain (factor 1.7) obtained in this experiment is not caused by the fact that we use an approximate transformation, but rather by the fact that the queries generated by the refinement operator are already very efficient. This depends on the language bias that is used: here it is well designed and generates efficient queries. Another reason for the low gain is that the refinement operator generates much more small sets of linked literals than larger ones.

6 Conclusion

We have introduced a query transformation that aims at reducing the average execution time of queries by replacing them with one of their permutations. Similar techniques are used in relational database management systems but there are some important differences. ILP systems execute queries top-down instead of bottom-up, look only for the first solution, may use expensive background predicates, and valid permutations of queries are constrained by the modes of the literals. Queries are also generated by the system itself and not (directly) by humans.

We have defined two versions of the transformation. Our first version transforms a query by replacing it with the permutation that has the lowest estimated average execution time. For computing this estimate we make two important approximations. The first one is that we use an approximate formula for computing the average execution time of a query based on the average execution times and non-determinacies of its literals. The second one is that we use uniform distributions over input substitutions to estimate the average execution time and non-determinacy of a literal. Such a uniform distribution will differ from the actual distribution of input substitutions for the literal, which depends on its position in the SLD-trees.

A second version of the transformation extends the first one and is able to handle queries that are composed of different independent sets of linked literals, separated by cuts. Such queries are generated by the *cut-transform* [13], which is implemented in the ILP system Aleph.

Our experiments show that the two versions of the transformation both approximate the theoretical optimal transformations, which replace a query by the permutation that has minimal average execution time, very well. The obtainable efficiency gain of the transformation over using the original queries, as generated by the refinement operator of the ILP system, depends much on the language bias that is used. The language bias used in our experiments was well designed in the sense that the efficiency of the generated queries was close to that of the best possible permutations. Our experiments further show that the efficiency gain increases with the length of the sets of linked literals. By reordering the sets themselves, not much efficiency can be gained.

The time complexity of our transformation is exponential in query-length (or independent set size) because it considers all possible permutations of a given query. We would like to develop a different version of our transformation that uses a greedy method to find an approximate solution in less time. In this way it could be possible to obtain a better trade-off between the overhead introduced by the transformation itself and the efficiency gained by executing the transformed queries.

Kietz and Lübbe introduce in [6] an efficient subsumption algorithm which is based on similar ideas. Their algorithm moves deterministic literals to the front while executing a query on a given example. One important difference is that their transformation is dynamic, because it is performed for each query and each example. The transformation described in this work is static: a given query is transformed once and then executed on all examples. One advantage of a static transformation is that the possible overhead introduced by the transformation itself decreases if the number of examples increases. Further work will include comparing both transformations.

Another approach for improving the efficiency of query execution is the use of query-packs [2]. The basic idea here is that sets of queries that have a common prefix, as they are generated by the refinement operator of a typical ILP system, can be executed more efficiently by putting them in a tree structure called a query-pack. In further work, we intend to combine the transformations presented here with query-packs. Combining query transformations with query-packs is difficult because the transformation may ruin the structure of the pack. Currently we are working on combining query-packs with the transformations described in [13].

Because the language bias has an important influence on the efficiency gain, we would like to try our transformation on more data sets with different types of language bias. One interesting approach here would be to use a language bias that generates larger sets of linked literals. Such a language bias is useful for first order feature construction, where one is interested in predictive relational patterns, which can be used for example, in propositional learning systems.

Acknowledgments

Jan Struyf is a research assistant and Hendrik Blockeel a post-doctoral fellow of the Fund for Scientific Research (FWO) of Flanders.

References

1. 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.
2. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 2001. Submitted.
3. M. Carlsson. Freeze, indexing, and other implementation issues in the WAM. In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming (ICLP87)*, Series in Logic Programming, pages 40–58. MIT Press, 1987.

4. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2nd edition, 1989.
5. M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2), 1984.
6. J.U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In *Proceedings of the 11th International Conference on Machine Learning*, pages 130–138. Morgan Kaufmann, 1994.
7. A. Krall. Implementation techniques for prolog. In N. Fuchs and G. Gottlob, editors, *Proceedings of the Tenth Logic Programming Workshop, WLP 94*, pages 1–15, 1994.
8. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
9. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
10. C. Nédellec, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 82–103. IOS Press, 1996.
11. D. Pavlov, H. Mannila, and P. Smyth. Beyond independence: Probabilistic models for query approximation on binary transaction data. *IEEE Transactions on Knowledge and Data Engineering*, 2003. To appear.
12. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
13. V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 2002. In press.
14. A. Srinivasan, R.D. King, and D.W. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.
15. J. Struyf, J. Ramon, and H. Blockeel. Compact representation of knowledge bases in ILP. In *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 254–269. Springer-Verlag, 2002.
16. F. Železný, A. Srinivasan, and D. Page. Lattice-search runtime distributions may be heavy-tailed. In S. Matwin and C. Sammut, editors, *Inductive Logic Programming, 12th International Conference, ILP 2002*, volume 2583 of *Lecture Notes in Computer Science*, pages 333–345. Springer-Verlag, 2003.