ILP: Compute Once, Reuse Often *

Nuno A. Fonseca[1], Ricardo Rocha[2], Rui Camacho[3], and Vítor Santos Costa[2]

[1] Instituto de Biologia Molecular e Celular (IBMC), Universidade do Porto, Portugal
nf@ibmc.up.pt
[2] DCC-FC, Universidade do Porto, Portugal
{ricroc,vsc}@ncc.up.pt
[3] Faculdade de Engenharia & LIAAD, Universidade do Porto, Portugal
rcamacho@fe.up.pt

Abstract. Inductive Logic Programming (ILP) is a powerful and well-developed abstraction for multi-relational data mining techniques. However, ILP systems are not particularly fast, most of their execution time is spent evaluating the hypotheses they construct. The evaluation time needed to assess the quality of each hypothesis depends mainly on the number of examples and the theorem proving effort required to determine if an example is entailed by the hypothesis. We propose a technique that reduces the theorem proving effort to a bare minimum and stores valuable information to compute the number of examples entailed by each hypothesis (using a tree data structure). The information is computed only once (pre-compiled) per example. Evaluation of hypotheses requires only basic and efficient operations on *trees*. This proposal avoids re-computation of hypothesis' value in theory-level search and cross-validation algorithms, whenever the same data set is used with different parameters. In an empirical evaluation the technique yielded considerable speedups.

Keywords: Mode Directed Inverse Entailment, Efficiency, Data Structures, Compilation

1 Introduction

Several multi-relational data mining approaches have been proposed, such as tree-mining, graph-mining, or cross-relational mining [11]. One powerful and well-developed abstraction for multi-relational data mining techniques is Inductive Logic Programming (ILP) [1,2]. ILP has been successfully applied to problems in several application domains [4]. Nevertheless, improvements in efficiency and scalability are necessary to successfully tackle applications that learn from large data-sets and/or generate large hypothesis spaces.

* This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds from the *Programa Operacional "Ciência, Tecnologia, Inovação" (POCTI) e do Programa Operacional "Sociedade da Informação" (POSI) do Quadro Comunitário de Apoio III (2000-2006)*. Nuno A. Fonseca is funded by FCT grant SFRH/BPD/26737/2005.

Research in improving the efficiency of ILP systems has focused on reducing their sequential execution time, either by reducing the number of hypotheses generated (see, e.g., [5,6]), or by efficiently testing candidate hypotheses (see, e.g., [7,8,9]). Another line of research, recommended by Page [10] and pursued by several researchers is the parallelization of ILP systems[17].

It is well known that an ILP system generates many candidate hypotheses which have many similarities among them. Usually, these similarities tend to correspond to common prefixes among the hypotheses. Blockeel et al. [7] defined query-packs as a technique to exploit this pattern and improve the execution time of ILP systems. Inspired by their work, we focus on how to reduce the amount of theorem proving to a minimum. As a first step, in a previous work [3], we argued that in MDIE [16] based systems the ILP search process can be efficiently coded by considering the set of all clauses that can be generated from the bottom clause. This led to an algorithm where a *prefix-tree* is used to represent all clauses that can be generated during the search (much in the way of query packs). But, instead of actually *evaluating* these clauses, we estimate coverage by counting the number of bottom-clauses that generated those clauses. Initial results showed that such an approach can indeed improve performance over standard ILP search.

The above work uses a single tree to represent the whole search space. In this work, we go one step further by proposing a novel two step approach to MDIE-based ILP where:

1. A pre-compilation step defines the search space by generating a set of clauses per example (where a tree can be used to encode the set of clauses).
2. A search step implements a search using algorithms constructed from an algebra of set operations implemented over these sets of clauses.

Our original motivation was the observation that the same set of clauses is generated from the same example at different computation steps (i.e., at different steps of theory construction or when performing cross-validation). Hence, computing the set of clauses for each example a single time, before execution, could significantly improve performance. Indeed, experimental results do show a large reduction in execution time. Moreover, we believe that our approach also provides a novel, and very modular, framework for ILP algorithm design, where the search can be easily encoded using set operations.

The remainder of the paper is organised as follows. In Section 2 we provide a brief introduction to ILP and MDIE. Section 3 introduces the reader to the rationale of seeing the examples as set of clauses and in Section 4 we present a first algorithm that exploits this idea. Next, in Section 5, we describe the proposed two step algorithm ($T-once$ MDIE). In Section 6 some implementation details are discussed. In Section 7 we present an empirical evaluation of the impact in execution time and accuracy of our algorithm. Finally, in Section 8 we discuss our work and draw conclusions.

2 Background

The predictive ILP problem can be defined as follows. Let E^+ be the set of positive examples, E^- the set of negative examples, $E = E^+ \cup E^-$, and B the prior knowledge (*background knowledge*). In general, B and E can be arbitrary logic programs. The aim of an ILP system is to find a hypothesis (also referred to as a theory) H, in the form of a logic program, such that $B \wedge E^- \wedge H \not\models \Box$ (Consistency) and $B \wedge H \models E^+$ (Completeness), assuming that $B \not\models E^+$ and $B \wedge E^- \not\models \Box$.

Mode-Directed Inverse Entailment (MDIE) [16] uses *inverse entailment* together with *mode restrictions* as the basis to perform induction. The key idea in MDIE is to find all literals that could be used in hypotheses that explain the example. This is achieved through the construction of the *bottom-clause*, that can be considered as the set of all such literals.

Construction of the bottom-clause $\bot_e$ often proceeds as a standard fixed-point calculation algorithm. Starting from the example e, and using the mode declarations, we scan the mode language for all possible clauses of the form $e \leftarrow l_1$. We collect all answers for l_1 as a set $\mathcal{L}_1 = \cup_i l_{1i} \ \setminus \mathcal{L}_0$, where $\mathcal{L}_0 = \{e\}$. Next we generate the set $\mathcal{L}_2$ with all clauses of the form $l_1 \leftarrow l_2$, where $l_1 \in \mathcal{L}_1$. We keep repeating the process until reaching a fixed point (which may be the whole data-base) or reaching some user-defined constraint. Therefore, the bottom clause $\bot_e$ can be seen as $\cup_j \mathcal{L}_j$. Most ILP systems use the bottom clause in order to bound (anchor) the search space lattice. Therefore, most applications try to have relatively small bottom-clauses, as otherwise the search space is as big as if one just enumerates clauses.

3 Examples as Set of Clauses

MDIE-based systems use bottom-clauses to generate sets of clauses. Given a bottom-clause $\bot_e$, the refinement operator generates clauses from $\bot_e$ that will cover at least the example e. Let us call this set $\mathcal{S}$. The clauses in $\mathcal{S}$ share e, so we can say that e *forms* $\mathcal{S}$. Note that, in general, $\mathcal{S}$ will be arbitrarily large, and we will need to impose some further restrictions, such as clause length restrictions. Moreover, note that even if complete, $\mathcal{S}$ does not correspond to all clauses that cover e. Indeed, it is well known that the bottom-clause is not complete: we can generate clauses that cover an example e which cannot be refined from $\bot_e$ [19].

Still, it is interesting to try to understand the meaning of $\mathcal{S}$. An important question in this regard is: if a clause c generated for example e covers example x, will c or, to be more precise, a variant of c, be in x's bottom clause, $\bot_x$? We would expect this to be true for ground clauses. Indeed, if this was not the case there must be at least a ground clause $h \leftarrow g_1, \ldots, g_{i-1}, g_i$ not refined from $\bot_x$, such that $h \leftarrow g_1, \ldots, g_{i-1}$ can be refined from $\bot_x$. Moreover, g_i must be in $\bot_e$ but not in $\bot_x$. On the other hand, if g_i was in $h \leftarrow g_1, \ldots, g_{i-1}, g_i$ it can be reached from $h, g_1, \ldots, g_{i-1}$, so it must also be in $\bot_x$.

Consider, for example, the following bottom-clause for an example e:

$$\bot_e = \ l(A) \leftarrow h\_c(A,B), h\_c(A,C), d(B), o\_c(B), f(C).$$

and the following clause c:

$$c \ = \ l(A) \leftarrow h\_c(A,B), h\_c(A,C), d(C), o\_c(B).$$

Careful examination shows that $\bot_e$ is entailed by clause c. On the other hand, the closest clause c' that can be generated from the bottom-clause is:

$$c' \ = \ l(A) \leftarrow h\_c(A,B), h\_c(A,C), d(B), o\_c(B).$$

Although $c' = c\theta$, c' is a more *specific* version of the original clause, it is not a variant. In this case, we cannot find a variant, even though the example indeed covers the clause.

This suggests the following approach: given an example e construct the corresponding bottom clause $\bot_e$ and generate a set $\mathcal{S}$ with all legal clauses c such that c θ-subsumes $\bot_e$. Next, given a set of examples $\{e_1^+, e_2^+, \ldots, e_n^+, e_1^-, e_2^-, \ldots, e_m^-\}$ construct the corresponding sets of clauses $\{\mathcal{S}_1^+, \mathcal{S}_2^+, \ldots, \mathcal{S}_n^+, \mathcal{S}_1^-, \mathcal{S}_2^-, \ldots, \mathcal{S}_m^-\}$: finding the best clauses should be just a question of searching for clauses that appear in most $\mathcal{S}_i^+$ and not in $\mathcal{S}_i^-$. More precisely, if we allow no noise, then we would like to find the clause with the largest coverage from $\cup_i \mathcal{S}_i^+ \setminus \cup_j \mathcal{S}_j^-$.

We are not interested in the examples, but in the set of all clauses of interest, $\mathcal{S}$ (which would to a first approximation be close to $\cup_i \mathcal{S}_i^+$). Now, this set may grow quickly, and therefore needs a compact and fast representation. It makes sense to represent sets of clauses by structures optimised for quick access and sharing, such as the *tries* discussed in Section 6.

4 T-MDIE

Assuming that the above representation works, one approach to *estimate* the coverage of all clauses is: walk over all examples and generate all clauses subsuming the bottom-clause such that for each clause c generated from an example $e \in E$:

- If $c \in \mathcal{S}$, somehow state that c covers e.
- If $c \notin \mathcal{S}$, add c to $\mathcal{S}$ and state that c covers e.

This basic algorithm can be optimised if we visit positive examples first, and assume we do not care about clauses that only cover negative examples:

- If the example $e \in E^+$ and $c \in \mathcal{S}$, state that c covers one more positive example.
- If the example $e \in E^+$ and $c \notin \mathcal{S}$, add c to $\mathcal{S}$ and state that c covers one positive example.
- If the example $e \in E^-$ and $c \in \mathcal{S}$, state that c covers one more negative example.

– If the example $e \in E^-$ and $c \notin \mathcal{S}$, do nothing.

We therefore need to define an abstract set that we call *decorated set* $\mathcal{S}$ with all clauses and their coverage. A *decorated set* $\mathcal{S}$ is a set whose elements are clauses, and attached to each element are several counters (one counter for each class of the learning problem). With this abstraction we can easily implement any theory construction algorithm as shown in Figure 1. The main difference with systems like Progol or Aleph concerns the inner procedure *learn_T_MDIE()*. We next describe how clauses are being learned in the *T-MDIE* approach [3].

generalise_T_MDIE(B, E^+, E^-, C):
Given: background knowledge B, finite training set $E = E^+ \cup E^-$, constraints C.
Return: a hypothesis H that explains E and satisfies C.

1. $H = \emptyset$
2. **while** $E^+ \neq \emptyset$ **do**
3. $\quad h = learn\_T\_MDIE(B, E^+, E^-, C)$
4. $\quad E^+ = E^+ \setminus covered(h)$
5. $\quad H = H \cup h$
6. $\quad B = B \cup h$
7. **endwhile**
8. **return** H

Fig. 1. The greedy cover algorithm of a MDIE system implementation.

The *T-MDIE* algorithm has two basic stages (see Figure 2). First a decorated set $\mathcal{S}$ is constructed (lines 1 to 9) and then the best clause (according to some metric) is found by inspection of the set (line 10). The decorated set $\mathcal{S}$ is constructed as described above. First, all positive examples are processed and then a pruning procedure, *prune*(), is invoked to remove useless clauses from $\mathcal{S}$ (e.g., clauses with positive coverage lower than some predefined minimum number of positive examples). Next, all negative examples are also processed and then the set is pruned again. While processing the negative examples, the negative counters of the clauses in $\mathcal{S}$ are updated whenever a negative example generates a matching clause. This means that the clauses generated from the negative examples that are not in $\mathcal{S}$ are discarded.

learn_T_MDIE(B, E^+, E^-, C):
Given: background knowledge B, finite training set $E = E^+ \cup E^-$, constraints C.
Return: the *best* hypothesis that explains some of the E^+ and satisfies C.

1. $\mathcal{S} = \emptyset$
2. **foreach** $e \in E^+$ **do**
3. $\quad fillSet(\mathcal{S}, B, e, C)$
4. **endforeach**
5. $\mathcal{S} = prune(\mathcal{S}, C)$
6. **foreach** $e \in E^-$ **do**
7. $\quad fillSet(\mathcal{S}, B, e, C)$
8. **endforeach**
9. $\mathcal{S} = prune(\mathcal{S}, C)$
10. **return** $bestClauseInTree(\mathcal{S}, C)$

Fig. 2. The learning algorithm of $T - MDIE$.

The set $\mathcal{S}$ is filled in three main steps (see Figure 3): i) for each example we generate a bottom clause (line 2); ii) using the bottom clause we generate all valid clauses[4] (line 4), normalise them (line 5), and insert them in the set (line 6). Normalisation orders the literals according to the Prolog "@ <" order relation. We generate all renaming of existential variables to check if a variant already exists in the tree, therefore guaranteeing a unique representation for each clause. The $insertUpdateInSet()$ procedure works as follows. If the example class is positive the clause is inserted into $\mathcal{S}$ and the positive counter updated. If the class is negative, only the negative counter of the clause is updated (the clause is not added to $\mathcal{S}$, only the coverage is updated).

$fillSet(\mathcal{S}, B, e, C)$:
Given: decorated set $\mathcal{S}$, background knowledge B, example e, constraints C.

1. $class = getExampleClass(e)$
2. $bottom = saturate(e, B, C)$
3. **do**
4. $\quad clause = findNewValidClause(bottom, C)$
5. $\quad clause = normalise(clause)$
6. $\quad insertUpdateInSet(clause, \mathcal{S}, class)$
7. **while** $clause \; ! = \emptyset$

Fig. 3. From an example to a set of clauses.

The algorithm is shown to be complete when compared to the traditional approach of computing the coverage (PROLOG resolution). Therefore, the coverage calculated for a clause by the algorithm should be interpreted as an estimate since it may not be the exact (*correct*) value.

5 T-once MDIE

The construction of the set $\mathcal{S}$ requires saturating all the examples (both positives and negatives). Often, one may want to perform repeated runs on the same examples. For instance, one may want to experiment with different parameters, or one may be performing cross-validation. Next, we show how repeated saturation and clause generation can be avoided by *decoupling* the generation of $\mathcal{S}$ from its usage. The process is divided into two steps: a compilation step, where a $\mathcal{S}_e$ is generated for each example e and stored on disk. The stored $\mathcal{S}_e$ are loaded at runtime, therefore avoiding the saturation and generation of clauses.

Our algorithm works as follows. At compilation time, we construct a *set of clauses per example*; more precisely, we represent the saturated clause for each example e as a separate $\mathcal{S}_e$ as depicted in the algorithm presented in Figure 4. At learning time, we obtain clause coverage information through an algebra of *basic operations*, such as *union* or *subtraction*, on decorated sets. As an example, search for the best clause can be described as a search in the decorated set obtained from the *union* of all $\mathcal{S}_e$. Next, we show in more detail how such approach can

[4] Clauses satisfying the language and bias constraints.

be used to implement greedy coverage in a MDIE-based ILP system. It should be clear that similar operations can be used to implement other ILP algorithms.

compileAnswerSet(B, E, C):
Given: background knowledge B, finite training set $E = E^+ \cup E^-$, constraints C.

1. **foreach** $e \in E$ **do**
2. $class = getExampleClass(e)$
3. $bottom = saturate(e, B, C)$
4. $\mathcal{S}_e = \emptyset$
5. **do**
6. $clause = findNewValidClause(bottom, C)$
7. $clause = normalise(clause)$
8. $\mathcal{S}_e = insertInSet(clause, \mathcal{S}, class)$
9. **while** $clause\ != \emptyset$
10. $saveSet2File(\mathcal{S}_e, e, C)$
11. **endforeach**

Fig. 4. Compilation of examples procedure.

Figure 5 shows one approach to implement a MDIE-based algorithm (such as the default algorithms in Progol [16] and Aleph [12]) using the decorated sets. Like the algorithm presented in the previous section, the $learn\_T{-}once\_MDIE()$ algorithm has two main stages: first, it generates a $\mathcal{S}$ by loading the compiled decorated sets and merging them using a *joinAdd()* procedure; then the best clause is recursively selected using greedy cover removal.

$learn\_T - once\_MDIE(E, C)$:
Given: finite training set $E = E^+ \cup E^-$, constraints C.
Return: a hypothesis H that explains E and satisfies C.

1. $\mathcal{S} = \emptyset$
2. **foreach** $e \in E^+$ **do**
3. $\mathcal{S}_e = loadSetFromFile(e, C)$
4. $\mathcal{S} = joinAdd(\mathcal{S}, \mathcal{S}_e)$
5. **endforeach**
6. $\mathcal{S} = prune(\mathcal{S}, C)$
8. $H = \emptyset$
9. **while** $E^+ \neq \emptyset$ **do**
10. $h = bestClauseInTree(\mathcal{S}, C)$
11. $E^+ = E^+ \setminus covered(h)$
12. $H = H \cup h$
13. $\mathcal{S} = subtract(\mathcal{S}, \{\mathcal{S}_e |\ e \in covered(h)\ and\ e \in E^+\})$
14. **endwhile**
15. **return** H

Fig. 5. The greedy cover algorithm of a MDIE system implementation with pre-compilation.

6 Implementation Issues

Our algorithms depend on the ability to implement efficiently operations such as *union* and *subtraction* of sets. Furthermore, we need a data structure to store the

decorated sets (clauses and respective coverages). To do so efficiently, we used tries [14]. A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term (clause). An essential property of the trie data structure is that common prefixes are represented only once. This naturally applies to ILP since the hypothesis space is structured as a lattice and hypotheses close to one another in the lattice have common prefixes (literals).

Using Tries to Represent Hypotheses In order to maximise the number of common trie nodes when storing clauses in a trie, we used Prolog lists to represent the clauses. A clause of the form $Head :- Body_1, ..., Body_n$ is stored in the trie structure as an unique path corresponding to the list $[Head, Body_1, ..., Body_n]$. Such a path always starts at the root node in the trie, follows a sequence of trie nodes and terminates at a leaf data structure, the *ilp frame* data structure, that we used to extend the original trie structure to store associated information with the clause, namely information concerning the number of positive and negative examples covered by the clause. Figure 6 presents an example of a trie storing three clauses.

An important point when using tries to represent terms is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [15], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR_0, ..., VAR_N$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t. For example, in the term $[eastbound(T), has\_car(T, C), long(C)]$, $numbervar(T)$ and $numbervar(C)$ are respectively VAR_0 and VAR_1.

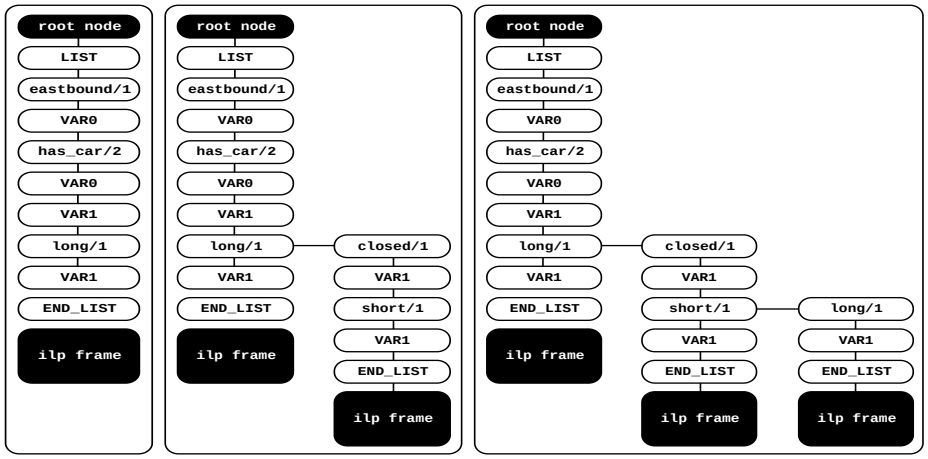

Fig. 6. Using tries to represent:
(a) *C = eastbound(T) :- has_car(T,C), long(C).*
(b) *C and D = eastbound(T) :- has_car(T,C), closed(C), short(C).*
(c) *C, D and E = eastbound(T) :- has_car(T,C), closed(C), long(C).*

Basic Trie Operations In the proposed *T-once* algorithm, an decorated set is constructed once and for each example. Therefore, since tries are used to represent the decorated sets we need to be able to perform some basic trie operations such as the union and subtraction of tries. The *trie-joinAdd()* and *trie-subtract()* procedures implement these operations. Given two tries, A and B, the *trie-joinAdd(A,B)* procedure returns a trie R representing the union of both tries, that is, if a term $t \in A$ or $t \in B$ then $t \in R$ and $ilp\_frame(t_R) = ilp\_frame(t_A) + ilp\_frame(t_B)$, where $ilp\_frame(t)$ represents the information concerning the number of positive and negative examples covered by t.

The *trie-subtract(A,B)* procedure returns a trie R equivalent to A but with the information concerning the number of positive and negative examples covered by the terms in B subtracted from the terms in A. More formally, if a term $t \in A$ then $t \in R$ and $ilp\_frame(t_R) = ilp\_frame(t_A) - ilp\_frame(t_B)$. Terms represented in B but not in A are ignored.

Searching through a chain of sibling trie nodes that represent alternative paths is done sequentially. When the chain becomes larger than a threshold value (8 in our implementation), we dynamically index the nodes through a hash table to provide direct node access and therefore optimise the search. Further hash collisions are reduced by dynamically expanding the hash tables. Hence, if the total number of trie nodes in tries A and B is respectively N_A and N_B, then the time complexity of the *trie-joinAdd(A,B)* and *trie-subtract(A,B)* procedures is $O(N_A + N_B)$.

7 Experiments and Results

The goal of the experiments was to evaluate the impact of the proposed approach on the execution time and quality of the models when dealing with real application problems. We implemented the two algorithms in the April ILP system [13]. For each data set the system was executed with the following configurations: standard MDIE implementation using a deterministic top-down breadth-first search (DTD-BF), $T - MDIE$, and $T - once$.

7.1 Experimental Settings

The experiments were performed on an AMD Athlon(tm) MP 2000+ dual-processor PC with 2 GB of memory, running Linux (kernel 2.6.12) Fedora. The data sets used were downloaded from the Machine Learning repositories at the Universities of Oxford[5] and York[6]. Table 1 characterises the data sets in terms of number of positive and negative examples as well as background knowledge size (number of relations used). The total number of examples ranges from 205 in the Mutagenesis data set up to 1762 in the Pyrimidines data set.

[5] `http://www.comlab.ox.ac.uk/oucl/groups/machlearn/`
[6] `http://www.cs.york.ac.uk/mlg/index.html`

Data set	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$
Carcinogenesis	202	174	44
Mutagenesis	136	69	21
Pyrimidines	881	881	244

Table 1. Data sets. $\mid E^+ \mid$ is the number of positive examples, $\mid E^- \mid$ is the number of positive examples, and $\mid B \mid$ is the number of relations in the background knowledge.

The search was constrained to clauses with 3 literals (maximum) in the body. The clause length was constrained due to the time taken by the DTD-BF algorithm. In practice we tested $T - once$ algorithm on some datasets with a clause length up to 5 literals in the body. We performed a 10-fold cross validation to evaluate the training time and accuracy.

7.2 Results and Discussion

Naturally, $T - once$ MDIE execution time requires some time to compile the examples. Table 2 presents the compilation (pre-processing) time, in seconds, taken for each data set, the average number of clauses compiled (refined), and the average file size of each compiled example (in kbytes). Cleary, the compilation is not a particularly fast process and further improvements should be made. Nevertheless, compilation is performed only *once*, as long as saturation-related settings are not changed or the clause length used is not increased. Therefore, in subsequent runs where other parameters (e.g., as noise) are changed there is no need to recompile the examples.

Data set	Time (sec)	Clauses	Size
Carcinogenesis	2,840	19,351 k	920 kb
Mutagenesis	6,054	12,659 k	301 kb
Pyrimidines	1,451	2,079 k	33 kb

Table 2. Compilation (pre-processing) time, average number of clauses compiled by example (in thousands), and average file size (in kbytes) of each compiled example.

Table 3 compares the execution times of $DTD-BF$, $T-MDIE$, and $T-once$ algorithms. The values presented are the average of a 10-fold cross validation and the sum of the execution times (within brackets). The results show that once the examples are compiled, $T-once$ is several times faster than $T-MDIE$ and $DTD-BF$ in all datasets. However, if we take into account the compilation time then the best approach, for the 10-fold runs, is clearly $T-MDIE$. Naturally, the gains of using $T - once$ increase with the number of runs performed. Therefore, it is well suited for cross-validation and for performing parameter tuning.

Finally, Table 4 presents the average accuracy for the two approaches. It shows that in spite of the coverage computed to be an estimate, the improvements in performance are not obtained at a cost of the quality of the models generated.

Data Set	$DTD-BF$	$T-MDIE$	$T-once$
Carcinogenesis	617 (*6,170*)	59 (*590*)	14 (*2,980*)
Mutagenesis	2,487 (*24,870*)	43 (*430*)	34 (*6,394*)
Pyrimidines	570 (*5,700*)	89 (*890*)	20 (*1,651*)

Table 3. Average execution time (in seconds) and cross-validation total execution time (within brackets). The total execution time of $T-once$ algorithm includes the compilation time.

Data set	$DTD-BF$	$T-MDIE$	$Diff$
Carcinogenesis	50 (3)	58 (9)	*+8*
Mutagenesis	75 (11)	74 (9)	*-1*
Pyrimidines	83 (3)	80 (1)	*-3*

Table 4. Average accuracy (standard deviation within brackets)

8 Conclusions

We have presented a novel approach to the execution of MDIE algorithms. Our approach proceeds in two steps. In the first step we compile each example as a set of clauses. In the second step we implement ILP search as a set of operations over these sets of clauses. Since such operations can be implemented very efficiently, our approach can generate major speedups over traditional ILP execution.

The reuse of the initial computation of the pre-compilation step pays-off whenever there is a large amount of repetition in clause evaluation. That happens when the induced theory has several clauses. In this case, after each iteration the covered examples are *removed* and we only need to perform subtraction operations between the sets of clauses, an operation that can be efficiently implemented using tries. The technique also pays-off when using cross-validation.

A further advantage of the approach is that it can be easily parallelisable, as the first step runs independently for every example. Moreover, we believe that our approach is a step forward in facilitating experimentation with different parameters, and namely in using internal cross-validation for parameter selection in ILP. On the other hand, the approach applies to MDIE-based algorithms only, and it needs further investigation when exploring longer clauses or in data sets with large numbers of examples (some techniques from [20] may help in that direction).

Last, an interesting insight from our approach is that we can abstract the ILP search procedure as a process of *tree-mining* over the trees representing individual examples. We believe that this suggests new and exciting directions for future research in this area.

References

1. S. Muggleton. Inductive logic programming. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 43–62. Ohmsma, Tokyo, Japan, 1990.

2. S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–317, 1991.
3. R. Camacho, N. A. Fonseca, R. Rocha and V. S. Costa. *ILP :- Just Trie It.* 17th International Conference on Inductive Logic Programming, 2007.
4. Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.
5. C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, 1996.
6. Rui Camacho. Improving the efficiency of ilp systems using an incremental language level search. In *Annual Machine Learning Conference of Belgium and the Netherlands*, 2002.
7. Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
8. Vítor Santos Costa, Ashwin Srinivasan, and Rui Camacho. A note on two simple transformations for improving the efficiency of an ILP system. *LNCS*, 1866, 2000.
9. Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 2002.
10. David Page. ILP: Just do it. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 3–18. Springer-Verlag, 2000.
11. Jiawei Han and Micheline Kimber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
12. Aleph. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.
13. Nuno A. Fonseca, Fernando Silva, and Rui Camacho. April - An Inductive Logic Programming System. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA06)*, volume 4160 of *LNAI*, pages 481–484, 2006. Springer-Verlag.
14. E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.
15. L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-Commutative Discrimination Nets. In *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74, 1993. Springer-Verlag.
16. S. Muggleton, *Inverse Entailment and Progol*, New Generation Computing, Special issue on Inductive Logic Programming. 245-286, vol 13, N. 3-4, 1995.
17. Nuno A. Fonseca and Fernando Silva and Rui Camacho, *Strategies to Parallelize ILP Systems*, Proceedings of the 15th International Conference on Inductive Logic Programming (ILP 2005), LNAI, vol 3625, pp 136–153, 2005.
18. Nuno A. Fonseca and Ricardo Rocha and Rui Camacho and Fernando Silva *Efficient Data Structures for Inductive Logic Programming*, Proceedings of the 13th International Conference on Inductive Logic Programming LNAI vol 2835, pp 130–145, 2003.
19. Akihiro Yamamoto *Which Hypotheses Can Be Found with Inverse Entailment?* ILP '97: Proceedings of the 7th International Workshop on Inductive Logic Programming, pp 296–308, 1997.
20. Hendrik Blockeel and Luc De Raedt and Nico Jacobs and Bart Demoen, *Scaling Up Inductive Logic Programming by Learning from Interpretations*, Data Mining and Knowledge Discovery, vol. 3, N. 1, pp 59-93, 1999.