*Fundamenta Informaticae XXI (2001) 1001–1023*
*IOS Press*

# Compile the hypothesis space: do it once, use it often

**Nuno A. Fonseca**
*Instituto de Biologia Molecular e Celular (IBMC) & CRACS, University of Porto, Portugal, nf@ibmc.up.pt*

**Rui Camacho**
*FEUP & LIAAD, University of Porto, Portugal, rcamacho@fe.up.pt*

**Ricardo Rocha**
*DCC-FCUP & CRACS, University of Porto, Portugal, ricroc@dcc.fc.up.pt*

**Vítor Santos Costa**
*DCC-FCUP & CRACS, University of Porto, Portugal, vsc@dcc.fc.up.pt*

**Abstract.** Inductive Logic Programming (ILP) is a powerful and well-developed abstraction for multi-relational data mining techniques. Despite the considerable success of ILP, deployed ILP systems still have efficiency problems when applied to complex problems. In this paper we propose a novel technique that avoids the procedure of *deducing* each example to evaluate each constructed clause. The technique is based on the Mode Directed Inverse Entailment approach to ILP, where a bottom clause is generated for each example and the generated clauses are subsets of the literals of such bottom clause. We propose to store in a *prefix-tree* all clauses that can be generated from all bottom clauses together with some extra information. We show that this information is sufficient to estimate the number of examples that can be deduced from a clause and present an ILP algorithm that exploits this representation. We also present an extension of the algorithm where each prefix-tree is computed only once (compiled) per example. The evaluation of hypotheses requires only basic and efficient operations on trees. This proposal avoids re-computation of hypothesis' value in theory-level search, in cross-validation evaluation procedures and in parameter tuning. Both proposals are empirically evaluated on real applications and considerable speedups were observed.

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

Address for correspondence: IBMC, AC Nuno Fonseca, Rua do Campo Alegre 823, 4150-180 Porto, Portugal

## 1. Introduction

Multi-relational data-mining algorithms analyse complex and structured data (e.g., data stored in different tables of a relational database). Recent years have seen a wide range of multi-relational data mining approaches, in domains such as tree-mining, graph-mining, and cross-relational mining [14, 22]. One powerful and well-studied abstraction for multi-relational data mining is through Inductive Logic Programming (ILP) [31]. In ILP, the tables are abstracted as *relations*, and the goal is to learn *rules* that describe interesting patterns in the data, where rules are usually expressed in a language of First Order Logic. ILP can be used as a framework for supervised and unsupervised learning and with a wide range of learning algorithms [34], and has been successfully applied to problems in very different application domains [24]. ILP thus provides a very general framework for multi-relational data-mining. Unfortunately, this generality comes at a computational cost. Therefore, it has been argued that improvements in efficiency and scalability are necessary to successfully tackle applications that learn from large data sets and/or require the search of large hypothesis spaces [40].

Most ILP systems execute by proposing and evaluating rules, until a *good* rule is found. Usually, one has to search a huge space of rules, hence the performance of an ILP system primarily depends on the size of the search space. This suggests that in order to reduce execution time one should reduce the actual number of rules generated, either through better bias (e.g., [36, 7]) or through stochastic search (e.g., [45]). If this is not possible or desirable, one has to focus on the amount of time spent per node. At each node, the ILP system will generate and score a new rule. Since generating a rule is straightforward, research has focused on scoring in order to achieve speedups. Rules are scored based on the number of examples they cover, therefore faster scoring requires faster theorem proving (see, e.g., [4, 43, 42]). Last, one should observe that parallel systems can speed up ILP in a number of ways [18], and that sequential and parallel improvements may be combined.

In this paper we propose a novel approach to improve execution time of ILP systems. Our approach starts from the observation that most execution time in ILP is spent in computing *coverage*, that is, which examples are satisfied by a rule. In fact, most of this work is redundant: the same clause can be generated several times, and several clauses may be very similar. Query-packs [4] address this redundancy in an interesting way: first, one generates a number of clauses; second, one groups this clauses as a tree, so that one can take advantage of common prefixes; last, for each example one performs theorem proving over the whole tree, or pack. Although query packs reduce redundancy, they do not totally prevent redundant theorem proving; even within a query pack one may repeat computations [47]. Furthermore, because storing the exact coverage of every rule is very expensive, one has to restart the query pack from scratch every time one needs to add a new rule to a theory.

Ideally, we would like to do theorem proving only once. One way to do so is through *tabling* [42] or memoing: one can avoid computation by always reusing previously computing solutions. If one tables conjunctions of goals, one needs to compute answers for a conjunction only once, and reuse these answers for all new queries. Unfortunately, the number of conjunctions, their query and answer patterns grows memory usage rather quickly. In practice, one has to choose a balance between re-computation and saving space.

The approaches above assume that the problem is that we have a vast space of rules, and that we want to know how examples are covered by these rules. These approaches take advantage of the fact that rules are similar, or repeated, but they do not take advantage of the fact that *the space of rules is not independent from the space of examples*. To make our point more precise, consider Mode Directed

Inverse Entailment (MDIE) [32], a popular approach followed by several ILP systems (e.g., Golem [33], Progol [32], Aleph [46], Indlog [6], and April [19]). MDIE systems do not just generate rules: instead, they first select an example as *seed* and create its *bottom clause*. Intuitively, the bottom clause includes the literals related to the seed. Thus, only clauses (rules) that subsume it need to be considered in order to prove the seed. MDIE systems take advantage of this to reduce the search space: they only enumerate clauses that satisfy the seed.

In this work, we take advantage of a key observation: to a first approximation, MDIE can be used to build an algorithm for *enumerating the clauses that satisfy an example*. If we can find such an algorithm, the problem of whether a clause covers an example, which needs theorem-proving, reduces to the problem of whether a clause is in the set of clauses satisfying the example. In practice, as it is widely known, there are both theoretical and practical issues in enumerating *all* clauses entailed by an example [32, 48, 3]. A first problem is that, as discussed in this paper, the incompleteness of refinement operators means that the enumeration algorithms need to be approximate. A second problem, is how to represent compactly large sets of clauses. Given the characteristics of ILP search, *prefix-trees* are a natural approach to storing sets of clauses.

We propose two algorithms that avoid the procedure of *deducing* each example to evaluate each constructed clause. In our first algorithm, T-MDIE, we visit examples one by one and store all entailing clauses in a prefix-tree. The major contribution of T-MDIE is that, instead of actually evaluating these clauses, we estimate the coverage of a clause by counting the number of bottom clauses that generated the clause. Empirical results show that such an approach can indeed improve execution time over standard ILP search. In our second algorithm, TO-MDIE, we observe that in T-MDIE 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). The TO-MDIE algorithm thus separates execution in two steps:

1. A compilation step defines the search space by generating a set of clauses per example. Such set of clauses is encoded in a prefix-tree, as before.

2. The traditional search step is replaced by search algorithms constructed from an algebra of set operations implemented over these sets of clauses.

Experimental results for TO-MDIE do show a large reduction in execution time. Moreover, we believe that the TO-MDIE approach provides a novel, 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 sets of clauses and in Section 4 we present a first algorithm, called T-MDIE, that exploits this idea. Next, in Section 5, we describe the proposed two step algorithm that we called TO-MDIE. In Section 6, some implementation details are discussed. In Section 7 we point out and compare research work that is related to ours. In Section 8 we present an empirical evaluation of the impact in execution time and accuracy of our two algorithms. Finally, in Section 9 we discuss our work and draw conclusions.

## 2. Background

This section briefly presents some basic concepts and terminology of Inductive Logic Programming but is not meant as an introduction to ILP. For such introduction we refer to [34, 13, 37]. Throughout the text it is assumed that the reader is familiar with Logic Programming terminology, nevertheless we next provide a very short and incomplete introduction to Logic Programming for ease of reference (for a more complete treatment we refer to [28]).

### 2.1. Logic Programming

The following concepts are often referred in ILP. Without loss of generality we may consider each clause $c$ as a sequential definite clause [25], i.e., a sequence of literals, in the form $l_0 \leftarrow l_1, \ldots, l_n$ $(n \geq 1)$ where $l_0$ is the head literal of the clause and each $l_i$ $(1 \leq i \leq n)$ is a body literal at depth $i$. A literal $l_i$ can be represented by $p_i(A_1, \ldots, A_{ia})$ where $p_i$ is a predicate symbol with arity $ia$ and $A_1, \ldots, A_{ia}$ are argument terms. A term is a variable (represented by an upper case letter followed by a string of lower case letters and digits) or a function symbol (represented with a lower case letter followed by a string of lower case letters, digits or underscores) followed by a bracketed $n$-tuple of terms. A variable represents an unspecified term for which a value can be assigned (usually designated as instantiated or bound). A variable can be instantiated only once with another variable or a term. Let $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, $\theta$ is said to be a *substitution* when each $X_i$ is a variable and each $t_i$ is a term. The application of $\theta$ to a term $t$, denoted by $t\theta$, is the act of replacing every occurrence of $X_i$ in $t$ by $t_i$.

A clause $l_0 \leftarrow l_1, \ldots, l_n$ $(n \geq 1)$ can be interpreted as $l_0$ if $l_1$ and ... and $l_n$ and is usually represented as '$l_0 : -l_1, \ldots, l_n$'. A fact is a body-less clause (e.g., represented as $l_0$.). A recursive clause has at least one literal in the body with the same predicate symbol as the head literal. A finite set of clauses is called a clausal theory (or logic program) and represents a conjunction of clauses. A theory that contains recursive clauses is called a recursive theory.

### 2.2. ILP Problem

The most common task addressed by predictive ILP systems 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*). The aim of an ILP system is to find an 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 $B \wedge E^- \not\models \Box$ and $B \not\models E^+$. In general, $B$ and $E$ can be arbitrary logic programs. However, in this paper it is assumed that $E$ is constituted only by facts.

A classical ILP example is the *Michalski train problem* [30]. In this problem, the theory to be found should explain why trains are travelling eastbound. Figure 1 presents the set of positive and negative examples, together with part of the background knowledge (describing the train $east1$). There are five examples of trains known to be travelling eastbound (the set of positive examples) and five examples of trains known to be travelling westbound (the set of negative examples). All our observations about these trains, such as size, number, position, contents of carriages, etc, constitutes the background knowledge. We next describe a method used for learning a theory.

**Positive Examples**

```
eastbound(east1).
eastbound(east2).
eastbound(east3).
eastbound(east4).
eastbound(east5).
```

**Negative Examples**

```
eastbound(east6).
eastbound(east7).
eastbound(east8).
eastbound(east9).
eastbound(east10).
```

**Background Knowledge**

```
has_car(east1,car_11).          has_car(east1,car_12).
has_car(east1,car_13).          has_car(east1,car_14).
short(car_12).                  short(car_14).
closed(car_12).                 long(car_11).
long(car_13).                   open_car(car_11).
open_car(car_13).               open_car(car_14).
shape(car_11,rectangle).        shape(car_12,rectangle).
shape(car_13,rectangle).        shape(car_14,rectangle).
wheels(car_11,2).               wheels(car_12,2).
wheels(car_13,3).               wheels(car_14,2).
load(car_11,rectangle,3).       load(car_12,triangle,1).
load(car_13,hexagon,1).         load(car_14,circle,1).
...                             ...
```

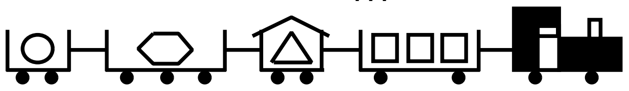

Figure 1. Michalski train problem: examples and part of the background knowledge (for the train $east1$).

## 2.3. Mode-Directed Inverse Entailment

Mode-Directed Inverse Entailment (MDIE) [32] is an approach followed by several ILP systems to address the ILP problem described above (e.g., [32, 46, 6, 1, 39]). MDIE uses *inverse entailment* together with *mode restrictions*[1] as the basis to perform induction. The key idea in MDIE is to find all literals that could be used in hypotheses (clauses) that explain an example (seed). This is achieved through the construction of a bottom clause, that can be considered as the set of all such literals.

In MDIE, the procedure to find a clause can be described as follows:

1. Pick an example $e$ from $E^+$ (the *seed*).

2. Build a bottom clause (or *most specific clause*) $\perp_e$ that entails the selected seed example. $\perp_e$ is a clause that explains an example $e$ relatively to the background knowledge $B$ (and $H$ if the target predicate is recursive)[2]. The bottom clause is usually a clause with several literals, i.e., it has the form $l_0 : -l_1, l_2, \ldots$, where $l_i$ are ground consequences of $B \wedge e$. Since, in general, $\perp_e$ can have infinite cardinality, the ground consequences are derived from $B$ using a depth-bound proof procedure for some selected depth.

3. Find the best consistent clause(s) more general than $e$ by performing a general-to-specific search[3] in the space of clauses bounded below by $\perp_e$. The clauses' bodies generated during the search are subsets of the literals from $\perp_e$. The generation of clauses is performed by a function, termed *refinement operator*, which computes (generates) a set of specializations (dually, generalizations) of a (set of) clause(s). Note that, in general, the number of clauses generated can be arbitrarily

[1]The mode restrictions [32], more specifically the type and input/output mode declarations, supply information concerning the arguments of each predicate that may appear in the hypotheses. The type declarations of the predicate to be learned are useful because the learner needs only to consider a subset of the hypotheses space that is type-conform.

[2]In general, a bottom clause can also be constructed as the relative least general generalization of two (or more) examples [33] with respect to the given background knowledge $B$, or as the most specific resolvent of an example [35] with respect to $B$.

[3]Although it is usual to perform a general-to-specific search, other directions may be pursued (see e.g., [39]).

```
:-modeh(1,eastbound(+train)).   % mode declaration for head literal eastbound/1
:-modeb(2,has_car(+train,-car)).% mode declaration for body literal has_car/2
:-modeb(1,short(+car)).         % mode declaration for body literal short/1
:-modeb(1,closed(+car)).        % mode declaration for body literal closed/1
:-modeb(1,long(+car)).          % mode declaration for body literal long/1
```

Figure 2. Example of mode declarations using a Progol style mode language for the trains problem. Each mode declaration has two arguments: the first, the recall number, indicates how many solutions to a goal can be introduced in the bottom clause; the second argument has the predicate and predicate's arguments type and input/output (-/+) mode.

large, and the user needs to impose some further restrictions, such as a limit in clause length. Once we have several clauses we need some way to rank them. There are many measures to assess the quality of clauses [27], but all of them involve computing the coverage of a clause, i.e., the number of positive (positive cover) and negative examples (negative cover) derivable from the background knowledge.

Note that constraints are imposed on 2) and 3) in order to ensure that the algorithm terminates. The great advantage of using a bottom clause is that it bounds (anchors) the search lattice.

As an example, consider the set of examples and background knowledge given for the train problem (Figure 1 and mode declarations in Figure 2). To build the bottom clause we start by picking a seed (positive) example (e.g., $eastbound(east1)$). Next, using the given mode declarations, all ground consequences are deduced:

```
eastbound(east1):- has_car(east1,car_11),has_car(east1,car_12),
                   long(car_11), short(car_12), closed(car_12).
```

Then, the clause is variabilized by transforming the predicates' input/output arguments, thus obtaining the bottom clause for train $east1$ ($eastbound(east1)$):

$$\perp_{east1} = eastbound(A) : -has\_car(A,B), has\_car(A,C), long(B), short(C), closed(C).$$

Having the bottom clause, the next step, is to search for the best consistent clause more general than $eastbound(east1)$. The clauses' bodies generated during the search are subsets of the literals from the bottom clause. For instance, using a clause length limit of 4, an ILP system would generate the following clauses given the bottom clause $\perp_{east1}$:

```
eastbound(A):- has_car(A,B).
eastbound(A):- has_car(A,C).
eastbound(A):- has_car(A,B), long(B).
eastbound(A):- has_car(A,C), short(C).
eastbound(A):- has_car(A,C), closed(C).
eastbound(A):- has_car(A,C), short(C), closed(C).
```

We briefly described how a clause can be learned. The process of generating a theory (set of clauses) often involves the use of a kind of covering algorithm where one clause is learnt at a time. In a nutshell, the algorithm proceeds as follows. After learning a clause, all covered positive examples are separated (removed) from the training set and the next clause is learned from the remaining examples. The process repeats while the training set is not empty. In Section 4 an instance of this algorithm is outlined.

## 3. Examples as Sets of Clauses

The key idea of this work is that we can estimate a clause's coverage by simply visiting the examples where the clause can be refined from the example's bottom clause. This raises a number of issues. The first one is whether this estimate is exact or approximate. The second one is whether this estimate can be implemented effectively, as otherwise it would be of little interest.

We will address the first question next. Given a bottom clause $\perp_e$, we have seen that the refinement operator enumerates clauses that subsume $\perp_e$[4]. We are interested in the connection between the clauses that cover an example $e$ and the set of clauses $\mathcal{S}_e$ that subsume $\perp_e$. Unfortunately, it is well known that the set of clauses in $\mathcal{S}_e$ does not correspond to all clauses that cover $e$. For instance, given a recursive theory, we can generate clauses that cover an example $e$ which cannot be refined from $\perp_e$ [48].

In this work, we are interested on non-recursive theories. In this case, even though $\theta$-subsumption and logical implication are equivalent [23], the problem is that the refinement operator is known to be not complete [32]. Incompleteness stems from only considering literals from left to right and from considering them only once [3]. A weaker form of completeness would be useful when comparing clauses from different bottom clauses. Consider clause $c \in \mathcal{S}_{e_i}$. Clearly, $c$ must cover $e_i$. Consider now a different example $e_j$. Can we prove if $c$ covers $e_j$, is it the case that $c \in \mathcal{S}_{e_j}$? Unfortunately, the answer is again negative, and this follows immediately from the previous results. For example, consider the following example taken from [3]:

$$\begin{aligned}
&\perp_{e_1} = g(A) :- p(A, A)\\
&\perp_{e_2} = g(A) :- p(A, A), p(A, B)\\
&C_1 = g(A) :- p(A, A)\\
&C_2 = g(A) :- p(A, A), p(A, B)
\end{aligned}$$

Clearly, $C_2$ satisfies both examples, but $\mathcal{S}_{e_1}$ will only contain $C_1$, as $C_2$ is never generated by the refinement operator (remember that the refinement operator selects a literal only once).

One could investigate this result further by researching for classes of clauses such that we can achieve clause completeness. For example, it is straightforward to show by constructive induction that if a ground clause covers examples $e_k$ and $e_l$, and it belongs to $\mathcal{S}_{e_k}$, then it belongs to $\mathcal{S}_{e_l}$ (just observe that the last literal must be in both $\perp_{e_k}$ and $\perp_{e_l}$). In this work we follow a different approach: we study whether $\mathcal{S}_e$ provides a convenient *approximation* to whether the clause covers $e$. To do so, we try to validate this argument *empirically*.

Next, we discuss the second question, whether it is practical to represent and manipulate $\mathcal{S}_e$. Our first observation is that, as it is well known, the set of clauses $\mathcal{S}_e$ can grow very quickly with bottom clause

[4]The subsumption order is the generality order most often used in ILP and is defined as follows. Let $c_1$ and $c_2$ be clauses. A clause $c_1$ subsumes $c_2$ if there exists a substitution $\theta$ such that $c_1\theta \subseteq c_2$.

size, since it corresponds to combinations of literals within the bottom clause. Manipulating arbitrarily large sets of clauses is clearly impractical. We address this problem through a *length* restriction on the clauses contained in $\mathcal{S}_e$: we define $\mathcal{S}_e^l$ as the set of clauses refined from $\bot_e$ with at most $l$ literals. We can obtain this set in two ways: discarding all clauses of length $> l$ from $\mathcal{S}_e$, or, given the observation that the refinement operator is incremental on length, by only refining clauses with up to length $l$. Note that in both cases if a clause $c$ belongs to $\mathcal{S}_e^l$, then it also belongs to $\mathcal{S}_e$.

Given this restriction, the question is now whether we can store all clauses of length $\leq l$ effectively. In Section 6 we discuss in detail how prefix-trees can be used for this purpose.

## 4. T-MDIE

Our first algorithm approximates traditional ILP by using the sets $\mathcal{S}_e$ to simulate full breadth-first search up to a certain length $l$ [8]. For each example $e$, we generate all clauses subsuming the bottom clause and if $c \in \mathcal{S}_e^l$, state that $c$ covers $e$. The algorithm uses two multisets to represent clauses and their coverage: $\mathcal{M}^+$ store clauses covering the positive examples, and $\mathcal{M}^-$ stores clauses covering the negative examples. It works as follows:

- Construct $\mathcal{M}^+$ by generating all clauses subsuming the bottom clauses for the positive examples $E^+$.
- Prune $\mathcal{M}^+$ by discarding clauses with a multiplicity inferior to a predefined minimum number of positive examples.
- Construct $\mathcal{M}^-$ by generating all clauses subsuming the bottom clauses for the negative examples $E^-$.
- Prune $\mathcal{M}^+$ and $\mathcal{M}^-$ by discarding clauses with a multiplicity in $\mathcal{M}^-$ greater than a predefined number of negative examples (also referred as the *noise*).
- Enumerate the clauses in $\mathcal{M}^+$ and select the best clause by estimating $c$'s positive coverage as $c$'s multiplicity in $\mathcal{M}^+$, and its negative coverage as $c$'s multiplicity in $\mathcal{M}^-$.

Figure 3 shows the actual T-MDIE algorithm. First the multisets are constructed (lines 1 to 10) and then the best clause (according to some metric) is found by inspection of the multisets (line 11).

Most often, we would use the T-MDIE algorithm as the inner step in any theory construction algorithm. Figure 4 shows the algorithm being used to implement greedy coverage. The difference regarding systems such as Progol and Aleph concerns the inner procedure $learn\_T - MDIE$*()*.

### 4.1. Details

When implementing T-MDIE we found it convenient to reduce redundancy as much as possible. Figure 5 shows, in more detail, the inner loop of the T-MDIE algorithm.

The $fillMultiSet()$ procedure starts by initializing an empty set $\mathcal{S}$ to keep track of the clauses being found. Then, it generates the bottom clause for the given example $e$ (line 2). Next, it uses the bottom clause to generate *all* valid clauses $c$ satisfying the language and bias constraints (line 4). Each clause $c$

```
$learn\_T - MDIE(B, E^+, E^-, C)$:
    Given: background knowledge $B$, finite training set $E = E^+ \cup E^-$, constraints $C$.
    Return: the best clause that explains some of the $E^+$ and satisfies $C$.

1.  $\mathcal{M}^+ = \emptyset$
2.  $\mathcal{M}^- = \emptyset$
3.  foreach $e \in E^+$ do
4.      $fillMultiSet(\mathcal{M}^+, B, e, C)$
5.  endforeach
6.  $prunePositives(\mathcal{M}^+, C)$
7.  foreach $e \in E^-$ do
8.      $fillMultiSet(\mathcal{M}^-, B, e, C)$
9.  endforeach
10. $pruneNegatives(\mathcal{M}^+, \mathcal{M}^-, C)$
11. return $bestClause(\mathcal{M}^+, \mathcal{M}^-, C)$
```

Figure 3. The learning algorithm of T-MDIE.

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

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

Figure 4. The greedy cover algorithm of a MDIE system implementation.

is then normalised (line 5) before looking it up on $\mathcal{S}$ in order to be added to the given multiset $\mathcal{M}$ if not repeated (lines 6 to 9). The normalisation consists of two steps:

- The first step is motivated by prior work on query optimisation [43]. It removes redundant literals, and separates independent components.
- The second step orders the literals according to the Prolog standard order relation. The standard order relation in Prolog orders terms as follows [11]: variables roughly by age; floating-point numbers; integers; and compound terms ordered by the functor's name, arity, and arguments.

Notice that this algorithm will generate a number of clauses that would never be generated by an ordinary ILP system: namely, clauses that only cover negative examples. As we are only interested in

```
$fillMultiSet(\mathcal{M}, B, e, C)$:
    **Given:** multiset $\mathcal{M}$, background knowledge $B$, example $e$, constraints $C$.

1.  $\mathcal{S} = \emptyset$
2.  $bottom = saturate(e, B, C)$
3.  **do**
4.      $c = findNewValidClause(bottom, C)$
5.      $normalise(c)$
6.      **if** $c \notin S$ **then**
7.          $\mathcal{S} = \mathcal{S} \cup \{c\}$
8.          $\mathcal{M} = \mathcal{M} \cup \{c\}$
9.      **endif**
10. **while** $c \neq \emptyset$
```

Figure 5. Filling the multisets.

clauses that at least cover a predefined minimum number of positive examples, we can implement the following improvement when constructing $\mathcal{M}^-$. When considering a clause $c$ for a negative example, if $c \notin \mathcal{M}^+$, we can discard $c$ and all its refinements, as they do not cover sufficient positive examples.

## 4.2. T-MDIE in the Real World

Next we address two major issues we found to be important in practise: completion of the saturated clause and syntactic redundancy.

**Completeness and Recall Number** In almost every data set, ILP can only generate a subset of the full saturated clause. This subset is controlled by a depth factor $i$ on the maximum length of variable chains, and also by the *recall number*. Next, we discuss how these two factors affect our algorithm.

As we discussed, the $i$ constraint is a syntactic constraint that is applied uniformly to every goal while generating the bottom clause. By induction, it should be clear that if a variable chain respects the $i$ constraint in a saturated clause, it will respect the same constraint on every other saturated clause.

The *recall number* parameter indicates how many solutions to a goal can be introduced in the bottom clause. If set to `*`, it will include every answer. On the other hand, if set to a lower threshold than the actual number of different answers a goal can generate, this parameter becomes a source of incompleteness. As the answer order will be different with different examples, using low-values of this parameter results in incorrect execution when using our algorithm.

**Syntactically Redundant Clauses** It is very important to reduce the size of the multisets $\mathcal{M}$. The *switching lemma* [28] tells us that if a conjunction of goals $G_1, \ldots, G_n$ is satisfiable, then any permutation of these goals is also satisfiable. ILP systems often take advantage of this principle to reduce the number of clauses they actually need to generate: if one generates $a(X), b(X)$ there is no point in also generating $b(X), a(X)$. On the other hand, traditional ILP systems cannot use any ordering of goals, as they must respect an ordering that respects the mode declarations given by the user. Since our algorithm does not actually evaluate goals, this is unnecessary: we can choose any ordering between goals when checking for redundant goals. In this vein, we try to simplify all syntactically redundant clauses into normalised clauses, as described in the previous section, so that all syntactically equivalent clauses will have a canonical representation. However, when the clauses are presented to the user, the literals in the body are reordered so that the clause is in accordance to the mode declarations.

## 5. TO-MDIE

It is often necessary to perform repeated runs on the same examples. For example, the greedy covering algorithm (Figure 4) needs to consider examples repeatedly whenever it tries to add a clause to a theory. The same examples will also be considered whenever we try the algorithm with different parameters, or when we perform cross-validation.

In all these cases, we repeatedly perform saturation and clause generation steps. The TO-MDIE algorithm addresses this problem by *decoupling* the generation of $\mathcal{S}_e$ from its usage. With TO-MDIE, induction is divided into two steps:

- A *compilation step*, where a set $\mathcal{S}_e$ is generated for each example $e$ and stored on disk.
- A *learning step*, where the sets $\mathcal{S}_e$ are loaded from disk at run-time, therefore avoiding the saturation and generation of clauses.

The compilation algorithm is outlined in Figure 6. It basically follows the $fillMultiSet()$ algorithm (Figure 5), except that at the very end it generates a separate file per example.

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

1.  **foreach** $e \in E$ **do**
2.      $\mathcal{S}_e = \emptyset$
3.      $bottom = saturate(e, B, C)$
4.      **do**
5.          $c = findNewValidClause(bottom, C)$
6.          $normalise(c)$
7.          **if** $c \notin \mathcal{S}_e$ **then**
8.              $\mathcal{S}_e = \mathcal{S}_e \cup \{c\}$
9.          **endif**
10.     **while** $c \,! = \emptyset$
11.     $saveToFile(\mathcal{S}_e, e, C)$
12. **endforeach**
```

Figure 6. Compiling a set of clauses per example.

The learning algorithm is motivated by the observation that the T-MDIE algorithm can be described as the operation of adding every clause in $\mathcal{S}_e$ to the current $\mathcal{M}$. If $\mathcal{S}_e$ is a set, it is also a multiset, so T-MDIE can be seen as implementing the multiset join, $\uplus$, of every example.

Figure 7 shows in more detail how such operations can be employed to implement greedy coverage in a MDIE-based ILP system using the compiled multisets. Like the algorithm presented in the previous section, the $learn\_TO - MDIE()$ algorithm has two main stages. First, it generates the multisets $\mathcal{M}^+$ and $\mathcal{M}^-$ by loading the compiled examples and by merging them using the multiset join operation and then, the best clause is selected. Next, it uses a multiset subtraction operation to implement greedy cover removal. It should be clear that the same or similar operations can be used to implement other ILP algorithms.

```
$learn\_TO - MDIE(E^+, E^-, C)$:
    **Given:** finite training set $E = E^+ \cup E^-$, constraints $C$.
    **Return:** a theory $H$ that explains $E$ and satisfies $C$.

1.  $\mathcal{M}^+ = \emptyset$
2.  **foreach** $e \in E^+$ **do**
3       $\mathcal{S}_e = loadFromFile(e, C)$
4.      $\mathcal{M}^+ = \mathcal{M}^+ \uplus \mathcal{S}_e$
5.  **endforeach**
6.  $\mathcal{M}^- = \emptyset$
7.  **foreach** $e \in E^-$ **do**
8       $\mathcal{S}_e = loadFromFile(e, C)$
9.      $\mathcal{M}^- = \mathcal{M}^- \uplus \mathcal{S}_e)$
10. **endforeach**
11. $prunePositives(\mathcal{M}^+, C)$
12. $H = \emptyset$
13. **while** $E^+ \neq \emptyset$ **do**
14.     $h = bestClause(\mathcal{M}^+, \mathcal{M}^-, C)$
15.     $E^+ = E^+ \setminus covered(h)$
16.     $H = H \cup h$
17.     $\mathcal{M}^+ = \mathcal{M}^+ \setminus \uplus(\mathcal{S}_{covered(h)})$
18. **endwhile**
19. **return** $H$
```

Figure 7. The TO-MDIE algorithm. Each $\mathcal{S}_e$ is assumed to have been compiled and stored in disk.

## 6. Implementation Issues

The implementation of our algorithms depends on the efficient implementation of operations such as *union* and *subtraction* of multisets. Furthermore, we need a data structure to store the multisets. To do so efficiently, we used tries [20]. Tries were originally invented to index dictionaries, and have since been generalised to index recursive data structures such as terms. Please refer to [2, 21, 41] for the use of tries in automated theorem proving, term rewriting and tabled logic programs. An essential property of the trie data structure is that common prefixes are stored only once. This naturally applies to ILP since the hypothesis space is structured as a lattice and clauses close to one another in the lattice have common prefixes (literals).

## 6.1. Using Tries to Store Clauses

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. At the entry point we have the root node. Internal nodes store tokens in terms and leaf nodes specify the end of terms. Each root-to-leaf path represents a term described by the tokens labelling the nodes traversed. For example, the tokenized form of the term $p(X, q(Y, X), Z)$ is the stream of 6 tokens: $p/3, X, q/2, Y, X, Z$. Two terms with common prefixes will branch off from each other at the first distinguishing token.

Trie's internal nodes are four field data structures. One field stores the node's token, one second field stores a pointer to the node's first child, a third field stores a pointer to the node's parent and a fourth field stores a pointer to the node's next sibling. Each internal node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing sequentially through the list of sibling pointers. When a list of sibling nodes 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.

In order to minimize the number of nodes when storing clauses in a trie, we use Prolog lists to represent clauses. A clause of the form '$l_0$ :- $l_1, \ldots, l_n$' is thus stored in the trie structure as the list $[l_0, l_1, \ldots, l_n]$. Figure 8 presents an example of a trie storing three clauses. Initially, the trie contains the root node only. Next, we store the clause '$eastbound(T)$ :- $has\_car(T, C), long(C)$' and nine nodes (corresponding to nine tokens) are added to represent it (Figure 8(a)). The clause '$eastbound(T)$ :- $has\_car(T, C), closed(C), short(C)$' is then stored which requires eleven nodes. As it shares a common prefix with the previous clause, we save the six initial nodes common to both representations (Figure 8(b)). The clause '$eastbound(T)$ :- $has\_car(T, C), closed(C), long(C)$' is stored next and we save eight nodes, the same six as before plus two more nodes common with the second stored clause (Figure 8(c)). The *MDIE frame* data structure, at the end of each path, extends the original trie structure to store associated information with each clause. This information is the number of times a clause appears on each multiset, i.e., the number of positive and negative examples covered by the clause. This representation is discussed in more detail next.

An important point when using tries to store terms is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [2], 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$_0, \ldots,$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 clause '$eastbound(T)$ :- $has\_car(T, C), long(C)$', $numbervar(T)$ and $numbervar(C)$ are respectively VAR$_0$ and VAR$_1$.

## 6.2. Multiset Operations

A multiset can be formally defined as a pair $(M, m)$ where $M$ is the underlying multiset and $m$ is a mapping from the elements in $M$ to the natural numbers. This provides a natural intuition to implement multisets over tries. Given that each clause $c$ in the underlying multiset will ultimately correspond to a leaf in the trie, we extend leaves with the mapping $m$. In fact, we can do better: given that in practice

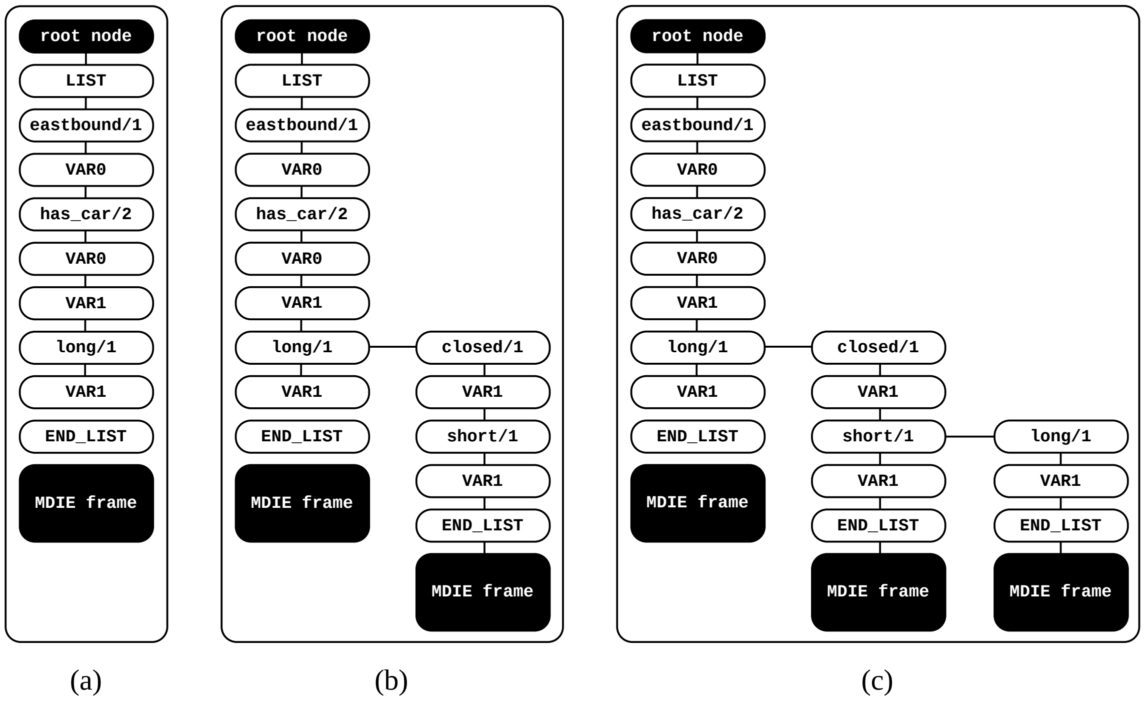

Figure 8. Using tries to store clauses. Initially, the trie contains the root node only. Next, we store the clauses: (a) '$eastbound(T)$ :- $has\_car(T, C), long(C)$'; (b) '$eastbound(T)$ :- $has\_car(T, C), closed(C), short(C)$'; and (c) '$eastbound(T)$ :- $has\_car(T, C), closed(C), long(C)$'.

both $\mathcal{M}^+$ and both $\mathcal{M}^-$ have the same underlying set of clauses, we *use the same trie* for both multisets.

To implement the proposed TO-MDIE algorithm, we need to be able to perform some basic trie operations such as the union and subtraction of tries. Figures 9 and 10 show respectively the $trieJoin()$ and $trieSubtract()$ procedures that implement these operations.

Given two multiset tries, $T_1$ and $T_2$, the $trieJoin(T_1, T_2)$ procedure returns in the first argument trie, $T_1^f$, the multiset join of both given tries, that is, if a term $t \in T_1$ or $t \in T_2$ then $t \in T_1^f$ and $frameMDIE(t_{T_1^f}) = frameMDIE(t_{T_1}) + frameMDIE(t_{T_2})$, where $frameMDIE(t)$ represents the information concerning the number of positive and negative examples covered by $t$. The $trieSubtract(T_1, T_2)$ procedure returns in the first argument trie, $T_1^f$, a trie equivalent to the initial $T_1$ trie but with the information concerning the number of positive and negative examples covered by the terms in $T_2$ subtracted from the terms in $T_1$. More formally, if a term $t \in T_1$ then $t \in T_1^f$ and $frameMDIE(t_{T_1^f}) = frameMDIE(t_{T_1}) - frameMDIE(t_{T_2})$. Terms stored in $T_2$ but not in $T_1$ are ignored.

Since tries provide complete discrimination for terms and permit lookup and possibly insertion to be performed in a single pass through a term, the time complexity of the $trieJoin(T_1, T_2)$ and $trieSubtract(T_1, T_2)$ procedures is linear in the total number of nodes in both tries $T_1$ and $T_2$.

```
$trieJoin(parent\_dest, parent\_src)$:
    **Given:** two internal trie nodes.

1.  $child\_src = nodeChild(parent\_src)$
2.  while $child\_src$ do
3       $child\_dest = getChildWithToken(parent\_dest, nodeToken(child\_src))$
4.      if $child\_dest$ then
5.          if $isLeaf(child\_dest)$ then
6.              $frameMDIE(child\_dest) += frameMDIE(child\_src)$
7.          else
8.              $trieJoin(child\_dest, child\_src)$
9.          endif
10.     else
11.         $trieCopy(parent\_dest, child\_src)$
12.     endif
13.     $child\_src = nodeSibling(child\_src)$
14. endwhile
```

Figure 9. The $trieJoin()$ procedure.

```
$trieSubtract(parent\_dest, parent\_src)$:
    **Given:** two internal trie nodes.

1.  $child\_src = nodeChild(parent\_src)$
2.  while $child\_src$ do
3       $child\_dest = getChildWithToken(parent\_dest, nodeToken(child\_src))$
4.      if $child\_dest$ then
5.          if $isLeaf(child\_dest)$ then
6.              $frameMDIE(child\_dest) -= frameMDIE(child\_src)$
7.          else
8.              $trieSubtract(child\_dest, child\_src)$
9.          endif
10.     endif
11.     $child\_src = nodeSibling(child\_src)$
12. endwhile
```

Figure 10. The $trieSubtract()$ procedure.

## 7. Related Work

The work presented in this paper is a new proposal to the solution of efficiency of ILP systems. Let us remember that the main sources of long execution times in ILP are: i) the size of the search space; ii) the highly search space redundancy and; iii) the theorem proving effort, either because of a large number of examples or because theorem proving is hard. We can find in the literature proposals for all of the above cases.

Techniques to reduce the search space size include language bias, as described in [36] and [7], and/or improved search methods such as parallel search ([18]) or stochastic search (e.g., [45]). Notice also that parallel and sequential improvements can be combined. Reducing the search space redundancy has been handled by the use of query transformations [43] and query-packs [4]. Query transformations and query-packs may also be used to reduce the theorem proving effort. Wherever the number of examples is too large, a technique such as lazy evaluation[6] has also proved to be very useful. Evaluating individual hypotheses may also be speed up using stochastic matching methods as described in [44] taking a constraint satisfaction approach as in [29] or by designing an efficient algorithm for matching clauses as in [12].

The above mentioned proposals to improve the execution time of ILP systems assume that hypotheses are dynamically evaluated during the search for the best clause and that theorem proving is required for such evaluation. This assumption is basically quite different from our proposal. In the method presented in this paper there is no theorem proving effort associated with the evaluation of hypotheses. That crucial point, that is responsible for the long execution time of ILP systems, is replaced by operations of trees that have extremely low execution times. Although we still need to explicitly generate the clauses of the search space, that is done only once. Because of this step we can profit from techniques that reduce the search space, techniques that avoid redundancy[5] and from using parallelism to speed up our system (by compiling each example search space in parallel).

The idea of having a compilation step to improve speed can be found in the Inductive Mercury Programming (IMP) system [15]. The IMP approach is however different from the one described here because IMP performs the usual search step with dynamic evaluation of each hypotheses generated. Compilation is used to allow for optimizations on the code (background knowledge and examples) in order to execute faster. The IMP uses the Mercury declarative language which requires the compilation of the background knowledge and examples but implies strong restrictions of the programs to use.

In our implementation tries are used to store efficiently the clauses of the search space. To the best of our knowledge, FARMER [38] was the first system where they were used as a technique to improve efficiency when learning Association Rules, in this case using the Warmr approach [10]. In a similar fashion, April uses them as a technique to reduce the amount of memory storage [17].

## 8. Experiments and Results

The goal of the experiments is to evaluate the proposed approaches on real application problems. The impact is assessed through considering execution time and model quality.

### 8.1. Experimental Settings

We followed a 10-fold cross validation methodology to assess the training time and accuracy. The data sets used were downloaded from the Machine Learning repositories at the Universities of Oxford[6] and

[5]Although some redundancy is already avoided by the use of tries and by the way clauses are stored in the tries.
[6]`http://www.comlab.ox.ac.uk/oucl/groups/machlearn`

York[7]. The NCTRER data set [26] was kindly provided by the Leuven Machine Learning research group. 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.

| **Data set** | $\mid E^+ \mid$ | $\mid E^- \mid$ | $\mid B \mid$ |
|---|---|---|---|
| Carcinogenesis | 202 | 174 | 44 |
| Mutagenesis | 136 | 69 | 21 |
| NCTRER | 131 | 101 | 5 |
| Pyrimidines | 881 | 881 | 244 |

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

The algorithms were implemented in the April ILP system [19]. For each data set we compare T-MDIE and TO-MDIE to a standard MDIE implementation using a Deterministic Top-Down Breadth-First search (DTD-BF). In this algorithm, DTD-BF, no limit on the number of clauses generated was imposed since T-MDIE and TO-MDIE consider all clauses up to the maximum clause length given (4 literals in the body, unless otherwise stated – clause length parameter set to 5). This means that in all algorithms considered, all valid clauses up to the given clause length are generated. The covering algorithm used in all algorithms tested follows the so-called *induce-max* approach implemented in Aleph [46]: each time a clause is committed to the theory, it is the best clause found using all uncovered positive examples as seed. We implemented DTD-BF using two techniques known to speedup the execution of ILP systems: query-transformations [43] and coverage caching [9]. Other relevant experimental settings (see [46] for a full description of the parameters) are $minpos = 5$, $i = 2$ and $noise = 30\%$. The experiments were performed on an AMD Athlon(tm) MP 2000+ dual-processor PC with 2 GB of memory, running Fedora Linux (kernel 2.6.12). The runs that took more than 2 days were aborted: *n.a.* (stands for data not available) is used for the outcomes of such experiments.

## 8.2. Results

First, we discuss how the running time for the three algorithms compares on the four data sets. Figure 11 depicts the execution times needed by the three algorithms as maximum clause length ranges from 2 (one literal in the body) to 5 (4 literals in the body). Notice that the time-scale in Figure 11 is logarithmic. Also, notice that some configurations exceed the maximum time limit of two days.

The results clearly show that TO-MDIE outperforms T-MDIE, and that T-MDIE outperforms DTD-BF. The T-MDIE algorithm outperforms DTD-BF in almost all applications and at almost all clause lengths. The only exception is for Carcinogenesis at small values of clause lengths, where the overhead of the method is most significant. Notice that the improvement tends to grow for larger clause lengths, and is often of more than an order of magnitude, even though DTD-BF performs more sophisticated pruning than T-MDIE. In fact, DTD-BF is not practical for clause length 5 on these data sets. Clearly, estimating the coverage of the clauses instead of using Prolog resolution pays off in terms of running time.

[7]`http://www.cs.york.ac.uk/mlg`

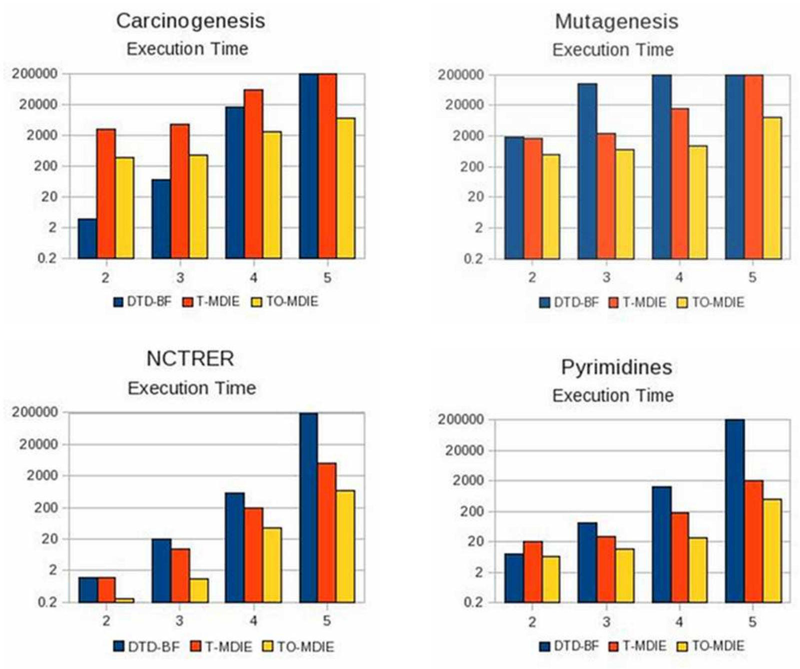

Figure 11. Performance evaluation (time in seconds) for the different strategies from 2 to 5 literals in the clause.

TO-MDIE further outperforms T-MDIE, and thus DTD-BF, in all applications and with different clause lengths. The only exception is observed in Carcinogenesis when the clause limit is set to 2 or 3. In these cases, DTD-BF is quite fast and the generation of extra clauses by TO-MDIE does not pay off. Nevertheless, the performance improvements, when using TO-MDIE, increase as the maximum depth is increased. In practise, this is very useful since the most interesting clauses have rarely one or two literals in the body.

It is interesting to study algorithm TO-MDIE in more detail. Table 2 presents information about the time and size of search space explored to obtain each example's trie, the size of the actual tries, and the time spent using the tries to learn. As expected, the search space is much larger than the size of the actual tries: tries are in fact at most 10KB per example, even with clause length equals 5. Table 2 also clearly shows that the average time taken to generate a theory, after compiling the examples, is very low (1 or 2 seconds). TO-MDIE time is mostly spent compiling the tries, suggesting that further improvements should focus there. On the other hand, we remember the reader that as compilation is performed only once, subsequent runs have no need to recompile the search space associated to each example (unless parameters are changed).

Notice that the number of clauses generated by TO-MDIE can be enormous, e.g., almost 700 million clauses generated in Carcinogenesis for a search space with a maximum depth of 5. The large number of clauses generated by TO-MDIE will become a problem as the maximum clause length increases. Hence, improving the refinement operator to generate less clauses should improve the performance of TO-MDIE and also T-MDIE.

Table 3 shows for each application and clause length limit the average accuracy and standard deviation. With one exception, we found that variations are not significant, and result only from different

| **Data set** | **Search Space** | **Trie Size** | **Time** | |
|---|---|---|---|---|
| | | | **Compilation** | **Learning** |
| Carcinogenesis | 683,883,665 | 10.04 | 76,206 | 1 |
| Mutagenesis | 529,054,555 | 2.34 | 78,205 | 1 |
| NCTRER | 10,261,522 | 0.82 | 7,003 | 1 |
| Pyrimidines | 54,138,100 | 8.47 | 4,777 | 2 |

Table 2. Average search space (in clauses per example) and compiled file size (in KB per example), compilation time (in seconds), and learning time (in seconds) at clause length=5.

orders in clause generation, which affects the generated theory and thus their predictive power. The exception is NCTRER, where DTD-BF has worst performance than the other two methods. We observed that the theories produced by DTD-BF for NCTRER have a single clause while the theories produced by T-MDIE and TO-MDIE have two clauses. This is a consequence of the covering algorithm itself. Recall that the values of the clauses may be slightly different when computed in DTD-BF and T-MDIE or TO-MDIE, which may change the ranking of clauses and lead to the selection of different clauses in DTD-BF than in T-MDIE or TO-MDIE. Therefore, the selection of a particular clause in DTD-BF prevents that further clauses are added to the theory.

| **Data set** | **Accuracy** | | |
|---|---|---|---|
| | **DTD-BF** | **T-MDIE** | **TO-MDIE** |
| Carcinogenesis | 63 (6) / 63 (6)/ 63 (6) / *n.a.* | 66 (7) / 63 (7) / 62 (8) / *n.a.* | 66 (7) / 63 (7) / 62 (8) / 62 (9) |
| Mutagenesis | * / 75 (10) / *n.a.* / *n.a.* | * / 67 (8) / 72 (11) / *n.a.* | * / 67 (7) / 72 (11) / 76 (7) |
| NCTRER | * / * / 42 (3) / 44 (5) | * / * / * / 75 (9) | * / * / * / 75 (9) |
| Pyrimidines | * / * / 51 (0) / 51 (1) | * / * / 51 (0) / 54 (2) | * / * / 51 (0) / 54 (2) |

Table 3. Accuracy (in each cell the 4 values obtained with a maximum clause length of 2/3/4/5 respectively) and standard deviation (bracketed). The '*' means that no theory was found with the settings provided.

To conclude, TO-MDIE reduced considerably the execution time without significantly reducing the accuracy. In fact, overall, there is no significant difference in accuracy when using either of the three algorithms.

## 9. Conclusions

We presented two novel approaches to improve the execution time of ILP algorithms based on MDIE were presented. The first approach (T-MDIE) attempts to improve performance by reducing the theorem proving effort on all clauses constructed during the search stage of a MDIE algorithm. This was possible by using a tree-like data structure to store all generated clauses, and their coverage. Coverage information allows the system to estimate efficiently the value of clauses. The reduction in the theorem proving effort is paid with an increase of the search space considered in the searches.

The second approach, that we name TO-MDIE, involves the compilation of the search space of each example. It 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 set operations over these sets of clauses. Since such operations can be implemented very efficiently, our approach can generate major speedups over traditional ILP execution and, also, when compared to the first approach.

The reuse of the initial computation of the 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 and theory-level search.

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 [5] may help in that direction). It would also be interesting to experiment with other refinement operators, and to study whether our ideas can be used effectively for recursive programs.

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 [16]. We believe that this suggests new and exciting directions for future research in this area.

## Acknowledgements

This work has been partially supported by projects STAMPA (PTDC/EIA/67738/2006), JEDI (PTDC/EIA/66924/2006), ILP-Web-Service (PTDC/EIA/70841/2006), and by Fundação para a Ciência e Tecnologia. Nuno A. Fonseca is funded by the FCT grant SFRH/BPD/26737/2006.

## References

[1] Anthony, S., Frisch, A. M.: Cautious Induction: An Alternative to Clause-at-a-time Induction in Inductive Logic Programming, *New Generation Computing*, **17**(1), January 1999, 25–52.

[2] Bachmair, L., Chen, T., Ramakrishnan, I. V.: Associative-Commutative Discrimination Nets, *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development*, number 668 in Lecture Notes in Computer Science, Springer-Verlag, Orsay, France, 1993.

[3] Badea, L., Stanciu, M.: Refinement Operators Can Be (Weakly) Perfect, *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings* (S. Dzeroski, P. A. Flach, Eds.), 1634, Springer, 1999, ISBN 3-540-66109-3.

[4] Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming through the Use of Query Packs, *Journal of Machine Learning Research*, **16**, 2002, 135–166.

[5] Blockeel, H., Raedt, L. D., Jacobs, N., Demoen, B.: Scaling Up Inductive Logic Programming by Learning from Interpretations, *Data Mining and Knowledge Discovery*, **3**(1), 1999, 59–93.

[6] Camacho, R.: *Inducing Models of Human Control Skills using Machine Learning Algorithms*, Ph.D. Thesis, Department of Electrical Engineering and Computation, Universidade do Porto, 2000.

[7] Camacho, R.: Improving the efficiency of ILP systems using an Incremental Language Level Search, *Annual Machine Learning Conference of Belgium and the Netherlands*, 2002.

[8] Camacho, R., Fonseca, N. A., Rocha, R., Costa, V. S.: ILP :- Just Trie It. *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP 2007)* (H. Blockeel et al., Eds.), 4894, Springer-Verlag, 2008.

[9] Cussens, J.: *Part-of-Speech Disambiguation Using ILP*, Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, 1996.

[10] Dehaspe, L., Toironen, H.: *Relational Data Mining*, chapter Discovery of relational association rules, Springer-Verlag, Berlin, 2000, 189–208.

[11] Deransart, P., Ed-Dbali, A., Cervoni, L., Ed-Ball, A. A.: *Prolog, The Standard : Reference Manual*, Springer Verlag, 1996.

[12] Di Mauro, N., Basile, T., Ferilli, S., Esposito, F., Fanizzi, N.: An Exhaustive Matching Procedure for the Improvement of Learning Efficiency, *Inductive Logic Programming: 13th International Conference (ILP03)* (T. Horváth, A. Yamamoto, Eds.), 2835, Springer-Verlag, 2003.

[13] Džeroski, S., Lavrač, N.: *Inductive Logic Programming: Techniques and Applications*, Ellis Horwood, 1994.

[14] Džeroski, S., Lavrač, N., Eds.: *Relational Data Mining*, Springer-Verlag New York, Inc., 2000.

[15] Fisher, B., Cussens, J.: Inductive Mercury Programming, *Inductive Logic Programming, 16th International Conference, ILP 2006* (S. Muggleton, R. P. Otero, A. Tamaddoni-Nezhad, Eds.), 4455, Springer, 2007, ISBN 978-3-540-73846-6.

[16] Fonseca, N. A., Camacho, R., Costa, V. S., Rocha, R.: k-RNN: k-Relational Nearest Neighbour Algorithm, *Proceedings of 2008 ACM Symposium on Applied Computing (SAC 2008)*, ACM, March 2008.

[17] Fonseca, N. A., Rocha, R., Camacho, R., Silva, F.: Efficient Data Structures for Inductive Logic Programming, *Proceedings of the 13th International Conference on Inductive Logic Programming* (T. Horváth, A. Yamamoto, Eds.), 2835, Springer-Verlag, Szeged, Hungary, 2003.

[18] Fonseca, N. A., Silva, F., Camacho, R.: Strategies to Parallelize ILP Systems, *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP 2005)* (S. Kramer, B. Pfahringer, Eds.), 3625, Springer-Verlag, Bonn, Germany, August 2005.

[19] Fonseca, N. A., Silva, F., Camacho, R.: April - An Inductive Logic Programming System, *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA06)*, 4160, Springer-Verlag, Liverpool, September 2006.

[20] Fredkin, E.: Trie Memory, *Communications of the ACM*, **3**, 1962, 490–499.

[21] Graf, P.: Term Indexing, Number 1053 in Lecture Notes in Artificial Intelligence, Springer-Verlag, 1996.

[22] Han, J., Kimber, M.: *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[23] Idestam-Almquist, P.: Generalization of Clauses under Implication, *Journal of Artificial Intelligence Research*, **3**, 1995, 467–489.

[24] ILP Applications, http://www-ai.ijs.si/ ilpnet2/apps/index.html, 2002.

[25] Van der Laag, P.: *An analysis of refinement operators in inductive logic programming*, Ph.D. Thesis, Erasmus Universiteit, Rotterdam, the Netherlands, 1995.

[26] Landwehr, N., Passerini, A., Raedt, L. D., Frasconi, P.: kFOIL: Learning Simple Relational Kernels., *Proceedings of the National Conference on Artificial Intelligence*, 2006.

[27] Lavrač, N., Flach, P., Zupan, B.: Rule Evaluation Measures: A Unifying View, *Proceedings of the 9th International Workshop on Inductive Logic Programming* (S. Džeroski, P. Flach, Eds.), 1634, Springer-Verlag, Berlin, June 1999.

[28] Lloyd, J. W.: *Foundations of Logic Programming,* 2nd edition, Springer-Verlag, Berlin, 1997.

[29] Maloberti, J., Sebag, M.: Fast Theta-Subsumption with Constraint Satisfaction Algorithms, *Machine Learning*, **55**(2), 2004, 137–174, ISSN 0885-6125.

[30] Michalski, R. S., Larson, J. B.: *Selection of Most Representative Training Examples and Incremental Generation of VL918 Hypotheses: The Underlying Mehtodology and the Description of Programs ESEL and AQ11*, Technical Report 867, Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[31] Muggleton, S.: Inductive logic programming, *Proceedings of the 1st Conference on Algorithmic Learning Theory*, Ohmsma, Tokyo, Japan, 1990.

[32] Muggleton, S.: Inverse Entailment and Progol, *New Generation Computing, Special issue on Inductive Logic Programming*, **13**(3-4), 1995, 245–286.

[33] Muggleton, S., Feng, C.: Efficient induction of logic programs, *Proceedings of the First Conference on Algorithmic Learning Theory*, Ohmsha, Tokyo, 1990.

[34] Muggleton, S., Raedt, L. D.: Inductive Logic Programming: Theory and Methods, *Journal of Logic Programming*, **19,20**, 1994, 629–679.

[35] Muggleton, S. H.: Inductive Logic Programming, *New Generation Computing*, **8**(4), 1991, 295–317.

[36] Nédellec, C., Rouveirol, C., Adé, H., Bergadano, F., Tausend, B.: Declarative Bias in ILP, in: *Advances in Inductive Logic Programming* (L. De Raedt, Ed.), IOS Press, 1996, 82–103.

[37] Nienhuys-Cheng, S.-H., de Wolf, R.: *Foundations of Inductive Logic Programming*, vol. 1228 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1997, ISBN 3-540-62927-0.

[38] Nijssen, S., Kok, J. N.: Faster Association Rules for Multiple Relations., *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, 2001.

[39] Ong, I. M., Dutra, I. d. C., Page, C. D., Santos Costa, V.: Mode Directed Path Finding, *Proceedings of the 16th European Conference on Machine Learning*, 3720, 2005.

[40] Page, D.: ILP: Just Do It, *Proceedings of the 10th International Conference on Inductive Logic Programming* (J. Cussens, A. Frisch, Eds.), 1866, Springer-Verlag, 2000.

[41] Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T., Warren, D. S.: Efficient Access Mechanisms for Tabled Logic Programs, *Journal of Logic Programming*, **38**(1), 1999, 31–54.

[42] Rocha, R., Fonseca, N. A., Costa, V. S.: On Applying Tabling to Inductive Logic Programming, *Proceedings of the 16th European Conference on Machine Learning, ECML-05, Porto, Portugal, October 2005*, 3720, Springer-Verlag, Berlin, 2005.

[43] Santos Costa, V., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., Laer, W. V.: Query Transformations for Improving the Efficiency of ILP Systems, *Journal of Machine Learning Research*, **4**, 2003, 465–491.

[44] Sebag, M., Rouveirol, C.: Tractable induction and classification in first-order logic via stochastic matching, *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1997.

[45] Srinivasan, A.: A study of two sampling methods for analysing large datasets with ILP, *Data Mining and Knowledge Discovery*, **3**(1), 1999, 95–123.

[46] Srinivasan, A.: *The Aleph Manual*, University of Oxford, 2004, `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`.

[47] Tronçon, R., Janssens, G., Vandecasteele, H.: Fast Query Evaluation with (Lazy) Control Flow Compilation, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings* (B. Demoen, V. Lifschitz, Eds.), 3132, Springer, 2004, ISBN 3-540-22671-0.

[48] Yamamoto, A.: *W*hich Hypotheses Can Be Found with Inverse Entailment?, *Proceedings of the 7th International Workshop on Inductive Logic Programming* (N. Lavrač, S. Džeroski, Eds.), 1297, Springer-Verlag, 1997.