On the Implementation of an ILP System with Prolog

Nuno Fonseca, Vitor Santos Costa, Fernando Silva, Rui Camacho

Technical Report Series: DCC-2003-03

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150-180 Porto, Portugal

Tel: +351+22+6078830 – Fax: +351+22+6003654

http://www.dcc.fc.up.pt/Pubs/treports.html

On the Implementation of an ILP System with Prolog

Nuno Fonseca[1], Vitor Santos Costa[2], Fernando Silva[1], Rui Camacho[3]

[1] DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
{nf,fds}@ncc.up.pt

[2] COPPE/Sistemas, Universidade Federal do Rio de Janeiro
Centro de Tecnologia, Bloco H-319, Cx. Postal 68511 Rio de Janeiro, Brasil
vitor@cos.ufrj.br

[3] Faculdade de Engenharia & LIACC, Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
rcamacho@fe.up.pt

October 2003

Abstract

Inductive Logic Programming (ILP) systems is a set of Machine Learning techniques that have been quite successful in knowledge discovery in relational domains. These systems implemented in Prolog are among the most successfull ILP systems. They challenge the limits of Prolog systems due to heavy usage of resources, such as database accesses and memory usage, and very long execution times. In this work we discuss the fundamental performance issues found in an ILP engine – the April system. Namely, we evaluate the impact of a fundamental technique, called coverage caching, that stores previous results in order to avoid recomputation. To understand the results obtained we profiled April's execution and present initial results. We advocate that the indexing mechanisms used in YAP Prolog database are inefficient and that improvement of these mechanisms may lead to significant improvements in Prolog based ILP systems.

Keywords: Inductive Logic Programming, Coverage Caching, Prolog indexation

1 Introduction

Inductive Logic Programming (ILP) [1, 2] is an established subfield of Machine Learning (ML). The objective of ILP is the induction of first-order clausal theories from a set of examples and prior knowledge. From a theoretical point of view, ILP is at the intersection of inductive learning and logic programming [2].

There are two major motivations for using ILP. First, ILP provides an excellent framework for learning in multi-relational domains. Second, the theories learned by general purpose ILP systems are in an high-level formalism, which is often understandable and meaningful for the domain experts. The advantages of ILP have been demonstrated through successful applications in difficult, industrially and scientifically relevant problems. Examples include engineering, natural language processing, environmental sciences, and the life sciences. For a survey of initial ILP applications see [3]. A more up-to-date list of applications of ILP systems to real world problems can be found in [4].

One major criticism of ILP systems is that they often have long running times. Several approaches have thus been proposed to improve ILP performance, such as several efficiency improvements [5, 6, 7, 8, 9], sampling on examples or on the search space [10], and parallelism [11, 12, 13]. Most of the refered techniques can be combined with others leading to further improvements. Understanding which techniques can contribute the most can thus be quite a difficult task.

April is a new ILP system that aims at exploiting several optimisation techniques for logic programming induction. Our initial interest in designing April is to evaluate the performance impact of different ILP techniques. Ultimately, we plan to use April to experiment with novel performance issues, and namely with parallelism.

We have chose to implement April in Prolog, as several other ILP systems. The major reason to do so is that the inference mechanism implemented by the Prolog engine is fundamental to most ILP learning algorithms. ILP systems can therefore benefit from the extensive work performed to improve the performance of Prolog systems (see eg. [14, 15]). On the other hand, ILP is a non-classical Prolog application because it uses large sets of ground facts and requires storing a large search tree. As pointed out by De Raedt [16] and Page[17], the performance of inductive logic implementations could be significantly improved by using special purpose Prolog implementations, thus benefiting from improvements in the Prolog technology.

In this work we discuss the fundamental performance issues of an ILP engine - the April ILP system that runs on the YAP [18] Prolog system. In particular we evaluate the impact of a fundamental technique, called coverage caching [7], that stores previous results in order to avoid recomputation. Such technique uses intensively the Prolog database. To understand the results obtained with coverage caching we profiled April's execution and present initial results. We argue in this report that the indexing mechanisms used in the YAP Prolog database are inefficient and that the improvement of such mechanisms may lead to significant improvements in Prolog based ILP systems. Our claim is justified by experimental results.

The contribution of our work is twofold: to an ILP researcher it provides an evaluation of the coverage caching technique implemented in Prolog using well known datasets; to a Prolog implementation researcher it shows the need of efficient internal database indexing mechanisms.

The remainder of this report is organized as follows. Section 2 provides some ILP background. A description of the April ILP system is provided in Section 3. Section 4 analyses the impact of the coverage caching technique on both memory usage and execution time. In Section 5 we explain the coverage caching impact results by analysing YAP's profiling data. Finally, in Section 6, we draw some conclusions.

2 ILP

This section briefly presents some concepts and terminology of Inductive Logic Programming necessary for the understanding of this report, but is not intended as an introduction to the field of ILP. For such introduction we refer to [19, 20, 21].

2.1 Problem

The objective of an ILP system is the induction of logic programs. As input an ILP system receives a set of examples (divided in positive and negative examples) of the concept to learn, and sometimes some prior knowledge (or *background knowledge*). Both examples and background knowledge are usually represented as logic programs. An ILP system tries to produce a logic program where positive examples succeed and the negative examples fail.

From a logic perspective, the 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 background knowledge. In general, B and E can be arbitrary logic programs, but is usual for E to be set of ground Prolog terms. The aim of an ILP system is to find a set of hypotheses (also referred as a theory) H, also a logic program, such that the following conditions hold:

- **Prior Satisfiability**: $B \not\models E^-$
- **Prior Necessity**: $B \not\models E^+$

- **Posterior Satisfiability**: $B \wedge H \nvDash E^-$ (Consistency)
- **Posterior Sufficiency**: $B \wedge H \vDash E^+$ (Completeness)
- **Posterior necessity**: $B \wedge h_i \vDash e_1^+ \vee e_2^+ \vee \ldots \vee e_n^+ (\forall h_i \in H, e_j \in E^+)$

The sufficiency condition is sometimes named *completeness* with regard to positive evidence, and the posterior satisfiability is also known as *consistency* with the negative evidence. Posterior necessity states that each hypothesis h_i should not be vacuous.

The consistency condition is sometimes relaxed to allow hypotheses to be inconsistent with a small number of negative examples (noise level). This allows ILP systems to deal with noisy data (examples and background knowledge), i.e., sets of examples or background knowledge that contain some inaccuracies or other flaws.

2.2 ILP as a search problem

As explained before, the normal problem of ILP is to find a consistent and complete theory, i.e., a set of clauses that imply all given positive examples and is consistent with the given negative examples. Since it is not immediately obvious which set of clauses should be picked as the theory, a search through the permitted clauses is performed to find a set with the desired properties.

To find a satisfactory theory, an ILP system searches through a search space of the permitted clauses. Thus, learning can be seen as searching for a correct theory [22]. The states in the search space (designated as *hypothesis space*) are concept descriptions (hypothesis) and the goal is to find one or more states satisfying some quality criterion. For efficiency reasons the search space is structured by imposing a *generality order* upon the clauses. Such order on clauses is usually denoted by $\preceq$. A clause C is said to be a generalization of D (dually: D is a specialization of C) if $C \preceq D$ holds. There are many generality orders, the most important are subsumption and logical implication. In both of these orders, the most general clause is the empty clause $\Box$. The refinement operators [23] generalize or specialize a hypothesis, thus generating more hypotheses.

The search can be done in two ways: specific-to-general [24] (or *bottom-up*); or general-to-specific [23, 25, 26, 6] (or *top-down*). In the generic-to-specific search the initial hypothesis is, usually, the more general hypothesis (i.e., $\Box$). That hypothesis is then repeatedly specialized through the use of refinement operators in order to remove inconsistencies with the negative examples. In the specific-to-general search the examples, with the use of background knowledge, are repeatedly generalized by applying refinement operators.

The hypotheses generated during the search are evaluated to determine their quality. A widely used approach to score a hypothesis is by computing its accuracy and coverage. The *accuracy* is the percentage of examples correctly classified by a hypothesis. The *coverage* of a hypothesis h is the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from $B \wedge h$. The time needed to compute the coverage of a hypothesis depends, primarily on the cardinality of E^+ and E^-.

2.3 Bias

ILP is a complex problem. Its complexity has two major origins: the size of the search space, and; the coverage computation. In most of the problems tackled by ILP practitioners, the search space is infinite and the non-determinate nature of the background knowledge makes the evaluation of each example hard. Practical ILP systems attenuate the complexity of the problem through the use of techniques to make the search more efficient and by imposing all sorts of restrictions, mostly syntactic, on candidate hypothesis (in order to reduce the search space). Such restrictions are called *bias*. Even a biased hypothesis space can be too large to make a complete search inviable (from a computational point of view). Several types of bias have been studied [27]. A declarative representation of the bias should be used so bias setting and shifting can be easily performed. Declarative bias may help ILP systems to be more adaptable to particular learning tasks.

2.4 Mode-Directed Inverse Entailment

Mode-Directed Inverse Entailment [26] (MDIE) is a technique widely implemented in ILP systems that uses inverse entailment together with mode restrictions to find a hypotheses set H. MDIE is the basis of the April's induction algorithm. To explain the main idea underlying Inverse Entailment, let us take the specification of the ILP problem: given background knowledge B and a set of examples E, find the simplest consistent hypothesis H such that

$$B \wedge H \models E$$

Since the goal is to find the simplest hypothesis, each clause in H should explain at least one example (otherwise there is a simpler H' which will do). If we take the case of H and E being single Horn clauses, it is possible to rearrange the problem as

$$B \wedge \neg E \models \neg H$$

Let $\neg \perp$ be the (potentially infinite) conjunction of ground literals which are true in all models of $B \wedge \neg E$. Since $\neg H$ must be true in every model of $B \wedge \neg E$ it must contain a subset of the ground literals in $\neg \perp$. Therefore

$$B \wedge \neg E \models \neg \perp \models \neg H$$

and for all H

$$H \models \perp$$

A subset of the solutions for H can be found by considering the clauses that θ-subsume $\perp$. Since, in general, $\perp$ can have infinite cardinality *mode declarations* are used to constrain the search for clauses which θ-subsume $\perp$. $\perp$ is usually designated as the bottom clause.

3 The April system

This section describes the April ILP system. April is a non-incremental (empirical), non-interactive, single predicate learning system. It generates non-redundant theories, can handle non-ground background knowledge, uses non-determinate predicates, uses a strong typed language and explicit bias declarations such as mode, type, and determination declarations. April combines ideas and features from several systems, such as Progol [26] et seq. [28, 29], Indlog [6], and CILS [30]. At the same time April aims to be an efficient and flexible ILP system. April has been used in the research of novel techniques that improve ILP systems efficiency. Its efficiency has been achieved through low memory consumption and low response time. Flexibility is achieved by a modular implementation and by providing the user with a high level of customizations. This customization allows April to emulate other systems by changing the configuration settings. However, we should note that April's emulation capabilities are no substitute for an exact implementation of the original algorithm.

April is implemented mainly in Prolog and runs in the YAP [18] prolog compiler. By using a Prolog compiler like YAP, April takes advantage of its tested and fast deductive engine. YAP also implements advanced techniques of Logic Programming, such as implicit and explicit parallelism [14], and tabling [15], that may be exploited in the future by April to improve response time.

The choice of using Prolog, and Prolog engines, implies some limitations concerning implementation of complex and efficient data structures. To circumvent this limitation some data structures have been implemented in C to improve response time and/or reduce memory consumption. The implemented data structures were made available in YAP as an external module. An example of such data structures are the RL-Trees and the Tries [31].

Next we present a small example, followed by a simplified description of April's main algorithm and architecture. A detailed description of the system is available as a technical report [32].

3.1 An Example

April constructs logic programs from examples and background knowledge. The syntax for examples, background knowledge, and hypotheses is YAP Prolog (hence ISO-Prolog standard). For instance, given various examples of the `multiplication(NumberA,NumberB,NumberR)` predicate, that multiplies two numbers, and some background knowledge, April can construct a definition of the `multiplication` predicate.

Positive Examples	Negative Examples	Background knowledge
`multiplication(0,1,0).`	`multiplication(0,0,1).`	`plus(X,Y,Z):-`
`multiplication(0,2,0).`	`multiplication(1,2,5).`	`number(X),`
`multiplication(0,3,0).`	`multiplication(2,3,12).`	`number(Y),`
`multiplication(1,4,4).`	`multiplication(2,2,6).`	`Z is X+Y.`
`multiplication(1,5,5).`	`multiplication(2,2,2).`	
`multiplication(2,3,6).`	`multiplication(3,4,3).`	`dec(X,Y):-`
`multiplication(2,4,8).`	`multiplication(3,3,6).`	`number(X),`
`multiplication(3,6,18).`	`multiplication(3,4,10).`	`Y is X-1.`
`multiplication(4,5,20).`	`multiplication(4,7,11).`	

Table 1: Examples and background knowledge of the multiplication predicate.

With the positive and negative examples in Table 1 April induces the following theory (program):

```
multiplication(A,B,C):-A=C,plus(A,A,A).
multiplication(A,B,C):-dec(A,D),mult(D,B,E),plus(B,E,C).
```

3.2 Induction Algorithm

April accepts as input: a training set consisting of positive (E^+) and, optionally, some negative (E^-) ground examples; background knowledge (B) in the form of logic programs; and a set of constraints C that include determination declarations, mode and type declarations, background predicates' properties, and facilities to change system parameters. As output, April generates a reduced[1] theory H that is consistent and complete, although these conditions may be relaxed by the constraints C.

The main algorithm of April is presented in Algorithm 1. Note that April has many configuration options and several options slightly modify the behavior of the algorithm presented. The outcome is that April has several algorithms that can be seen as "mutations" of the one presented. The outer cycle contains two inner cycles that we call *clause generation cycle* and *clause consumption cycle*. The idea underlying these two cycles refers to the cautious induction method implemented in CILS [30]. Like in CILS, April first generates a set of candidate clauses (clause generation). It then selects the clauses with higher quality and adds them to the theory. The difference between April and CILS is that April performs a more greedy selection, that may result in a slightly worst quality of the final theory (when compared to CILS). The outer cycle ends when there are no positive examples left or when the constraints are satisfied.

The clause generation cycle produces a *samplesize* number of clauses, each clause h_i is generated based in one example e_i^+. At each iteration, an example e_i^+ is selected sequentially or randomly from E_{cur}^+ (the choice is made by the user). The selected example e_i^+ is then saturated and flattened using B and C, producing the bottom clause ($\perp$). The saturation gathers all "relevant" ground atoms that can be derived from $B \wedge H \wedge \neg e_i^+$ and satisfy the constraints C. All relevant atoms collected during the construction of the bottom clause are flattened. Flattening is a method to make a theory function-free and was introduced in ILP by Rouveirol [33, 34]. Like other ILP systems, April flattens

[1]Let T be set of clauses. T is *reduced* if and only if T contains no redundant clauses.

Algorithm 1 April's main algorithm

```
Input : E^- and E^+                                                    /* The training set */
B                                                                      /* Background knowledge */
C                                                                      /* Set of constraints */
Output: H                                                              /* A theory */
H = ∅
E^+_cur = E^+
SampleSize = C(samplesize)                                             /* Size of the sample */
while notfinish_condition_ok() do                                      /* Default condition: E^+_cur ≠ ∅ */
   Best = pool_best()            /* Get best clause in the pool, first interaction is NULL */
   j = 1
   do                                                                  /* Clause generation cycle */
      e^+_i = select_example(E^+_cur, C, SampleSize, j)
      ⊥ = saturate(B, H, C, e^+_i)
      h_i = reduction(⊥, B, H, C, E^+_cur, E^-, Best)
      if h_i better than Best then Best = h_i
      add2pool(h_i)
      incrementj
   while j < SampleSize and j <| E^+_cur |

   while Best ≠ NULL                                                   /* Clause consumption cycle */
    E^+_cur = E^+_cur − covered(Best)                                  /* remove redundant examples */
    H = H ∪ Best                                                       /* Add best clause to theory */
    pool_remove(Best)                                                  /* Remove Best from pool */
    Best = pool_next_best()                                            /* Select next best clause in the pool */
    if End_Consumption(C) then break
   end while
end while
H = rem_redundant_clauses(B, H, E^+)                                   /* Remove redundant clauses from H */
```

all function symbols by introducing equalities. The bottom clause generated will contain all literals that may be found in the clauses generated during the search. A clause (hypothesis) is generated by performing a search through the hypothesis space bounded bellow by $\perp$. The *Best* clause is used by the refinement procedure to improve pruning, thus reducing the search space and improving efficiency. The clause h_i found at each iteration is added to a pool of clauses. The pool keeps the clauses found ordered by the number of positive and negative examples covered.

The clause consumption cycle tries to "consume" the clauses found, i.e. add the clauses to H. The best clause in the pool is added to H, the examples covered by *Best* removed from E^+_{cur}, and finally *Best* is removed from the pool. Then, all clauses in the pool will have their coverages recomputed (by invoking *pool_next_best*()). Those clauses in the pool that have a coverage of 1 are immediately removed and the others reordered. The best clause in the pool is then used as the new *Best* clause. The cycle ends when all clauses in the pool have been considered or the constraints C are satisfied.

Finally, the theory found is reduced, i.e., the redundant clauses that may exist in the theory are removed.

3.3 Module Architecture

April is implemented as a set of Prolog modules. This modularity allows developers with knowledge of the Prolog language to create an ILP system suited to their needs either by selecting a subset of the modules, either by replacing an existing module with their own implementation.

Figure 1 presents April's module architecture. The modules are organized in three major categories: data modules, functional modules, and extension modules. The data modules are used to store data, while the functional ones implement an algorithm or some functionality. The extension

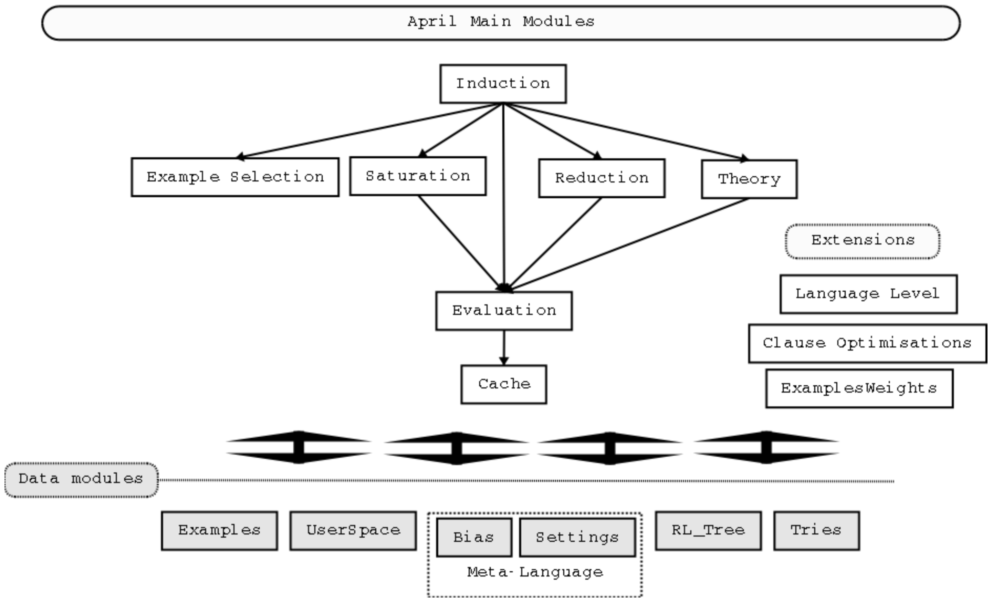

Figure 1: April's Architecture

modules, include modules that implement ideas available in other ILP systems, or described in papers.

Currently, there are three extension modules: the *language level module*, that implements Camacho's Incremental Language Level search [9]; the *clause optimisations* module, that implements the optimisations described by Costa et al. [8]; and the *ExamplesWeights* module, that provides functionalities to allow examples to have weights [35].

The *induction* module implements April's main algorithm. The *example selection module* provides the functionalities for selecting examples. The *saturation module* constructs a bottom clause from a given example. The *reduction module* performs the search through the hypotheses space to find a clause. The clauses generated during the search are evaluated to compute their coverage against the given examples. The coverage computation and the explicit calls to the Prolog interpreter are done in the *evaluation module*. The *theory module* processes the clauses found by the reduction module to generate the final theory that is presented to the user. The *cache module* implements the coverage caching scheme described in Section 3.4. The *examples module* manages the examples provided by the user. The bias declarations provided by the user are stored in the *bias module*, and configuration settings in the *settings module*. The *UserSpace module* stores all background knowledge provided by the user and the clauses that are part of the final theory. The RL-tree and Tries modules provide an interface to the correponding data structures. These data structures are described and evaluated in [31].

3.4 Coverage Caching

An efficient coverage computation is crucial for the performance of an ILP system. Several approaches have been proposed to improve coverage computation efficiency (see e.g [13, 8, 5, 6]), here we will describe the technique known as coverage caching [7].

The coverage of a clause h_i is computed by testing the candidate clause against the positive and negative examples. This is done by verifying for each example e in E if $B \wedge h_i \vdash e$. The time needed to compute the coverage of a clause depends, primarily on the cardinality of E^+ and E^-.

The idea in coverage caching is to maintain a cache with the coverage lists and accuracy of the clauses generated. This technique is used in other ILP systems, such as Aleph [29] and Indlog [6], to reduce the computation time spent in coverage tests.

The coverage lists are used in these systems as follows. A hypothesis S is generated by applying a refinement operator to another hypothesis G. Let $Cover(G) = \{all\ e \in E\ such\ that\ B \wedge G \models e\}$, where G is a clause, B the background knowledge, and E is the set of positive (E^+) and negative examples (E^-). Since G is more general than S then $Cover(S) \subseteq Cover(G)$. Taking this into account, when testing the coverage of S it is only necessary to consider examples in $Cover(G)$, thus reducing the coverage computation time. Cussens [7] extended this scheme by proposing what is designated as coverage caching. The coverage lists are permanently stored and reused whenever necessary, thus reducing the need to compute the coverage of a particular clause only once. Coverage lists reduces the effort in coverage computation at the cost of significantly increasing memory consumption.

The data structure used to maintain coverage lists in systems like Indlog, Aleph, and April are Prolog lists. For each clause two lists are kept: a list of positive examples covered and a list of negative examples covered. A number is used to represent an example in the list. The positive examples are numbered from 1 to $\mid E^+ \mid$, and the negative examples from 1 to $\mid E^- \mid$. The systems mentioned reduce the size of the coverage lists by transforming a list of numbers into a list of intervals. For instance, consider the coverage list $[1, 2, 5, 6, 7, 8, 9, 10]$ represented as a list of numbers. This list is represented as a list of intervals as $[1-2, 5-10]$.

In order to reduce execution time the cache must be very efficient, by this we mean that insertions and retrievals of elements in the cache should be done very fast. In April we use the YAP Prolog internal database to store clauses's coverage. Prolog databases are known to be a little slow, nevertheless it was the only solution available within the Prolog language. The impact of coverage caching on April's performance is analysed in more detail in the next section.

4 April performance

To analyse the initial performance of April, in particular the impact of the coverage caching technique on both memory usage and execution time, we conducted a series of experiments using datasets from the Machine Learning repositories of the Universities of Oxford[2] and York[3]. The experiments were performed on an AMD Athlon(tm) MP 2000+ dual-processor PC with 2GB of memory, running the Linux RedHat (kernel 2.4.20) operating system. We used version 0.5 of the April ILP system and version 4.3.24 of YAP Prolog.

Table 2 characterizes the datasets in terms of number of positive and negative examples as well as background knowledge size. Furthermore, it describes the April settings used for each dataset. The parameter *nodes* specifies an upper bound on the number of hypotheses generated during the search of an acceptable hypothesis. The i-depth corresponds to the maximum depth of a literal with respect to the head literal of the hypothesis [36]. *Samplesize* defines the number of examples used to induce a clause. The *language* (*lang.*) parameter specifies the maximum number of occurrences of a predicate symbol in a hypothesis [6]. *MinPos* specifies the minimum number of positive examples that a hypothesis must cover in order to be accepted. Finally, the parameter *noise* defines the maximum number of negative examples that a hypothesis may cover in order to be accepted.

Note that in order to speedup the experiments we limited the search space of some datasets by setting the parameter *nodes* to 1000. This reduces the total memory usage and execution time needed to process the dataset. However, since we are comparing the memory consumption and execution time when using coverage caching or not using it, the estimate obtained still gives a good idea of the impact of the feature.

Table 3 presents the impact of activating coverage caching in April. It shows the total number of hypotheses generated ($\mid H \mid$), the execution time, the memory usage, and the impact in performance for execution time and memory usage (given as a ratio between using coverage caching and not using coverage caching). The memory values presented correspond to the total memory used by April. The coverage lists were represented as lists of intervals.

As expected, the results indicate a significant increase in memory usage when coverage caching is activated. However, unexpectedly the use of coverage caching also increased the execution time, more than 5 times for the larger datasets (i.e. datasets with greater number of examples and $\mid H \mid$).

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

Dataset	Characterization			April's Settings					
	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$	nodes	i	samplesize	lang.	minpos	noise
amine uptake	343	343	32	1000	2	20	-	50	20
carcinogenesis	162	136	44	1000	3	10	3	20	10
choline	663	663	31	1000	2	10	-	50	20
krki	342	658	1	no limit	1	all	2	1	0
mesh	2272	223	29	1000	3	20	3	10	5
multiplication	9	15	3	no limit	2	all	2	1	0
pyrimidines	881	881	244	1000	2	10	-	75	20
proteins	848	764	45	1000	2	10	-	100	100

Table 2: Settings used in the experiments

Dataset	$\mid H \mid$	Time (sec.)		Memory (bytes)		yes/no(%)	
		no	yes	no	yes	Time	Memory
amine uptake	66933	58.37	357.4	3027460	11255228	612.30	371.77
carcinogenesis	142714	616.38	506.65	7541316	13542528	82.19	179.57
choline	803366	1840.25	13596.07	5327052	32537788	738.81	610.80
krki	2579	3.78	1.15	2225176	2318084	30.42	104.17
mesh	283552	637.34	3241.73	7255884	25733376	508.63	354.65
multiplication	478	8.87	8.93	4261768	4422080	100.67	103.76
pyrimidines	372320	915.95	5581.91	5659544	27856496	609.41	492.20
proteins	433271	7837.96	794.4	27075788	27495636	10.13	101.55

Table 3: Coverage caching results

On the other hand, the `proteins` dataset shows a reduction of around 90% in the execution time which is what one would like to observe when employing a caching mechanism.

5 Profiling April execution

The observed overheads in execution time presented in the previous section, being so unexpected, prompted us to further investigate the reasons for this behavior. A issue that we would like to clarify is whether coverage caching reduced the number of goal invocations executed. We decided to activate YAP's profiling and then rerun the April system for all datasets previously considered and observe the impact of coverage caching on the number of calls and retries.

Table 4 shows the total number of calls and retries performed by YAP with the cache activated and deactivated. The result values represent the aggregate number of calls and retries for all datasets. Note that the number of retries shown, with the cache deactivated, is lower than the real value because in some datasets the YAP counters overflowed. In these cases the maximum value possible was used instead. The use of cache reduced the number of calls by 90% and reduced the number of retries by at least 15%. This shows that the use of caching clearly achieves the goal of reducing computation but surprisingly the execution time increased by 56%. Note that the number of calls were reduced by 30 billions approximately.

To identify the causes for the increase in execution time, when using caching, we analyzed the profile logs in more detail to locate the modules that may be responsible for the overhead. Table 5 presents the distribution of the number of calls among the Prolog modules used by April. For each module, the table shows the number of calls and their weight within the total number of calls when cache is activated or deactivated, together with the variation in the number of calls when cache is activated. To simplify the table analysis, we disregard the modules whose weight was less than 1%. We also do not show the number of retries because the values are rather low in most of the modules.

Module	cache=yes	cache=no	yes/no
Calls	3,141,742,379	33,508,263,954	0.09
Retries	26,112,058,881	>30,730,206,551	0.84
Time (sec.)	38731.23	24718.04	1.56

Table 4: Total number of calls and execution time

The main exception is the `user_space` module that contains the background knowledge and is the module where examples coverage is performed.

Module	cache=yes		cache=no		CallsVariation
	Calls	Weight	Calls	Weight	
prolog	1,276,198,146	0.40	8,198,176,038	0.24	-6,921,977,892
utils	135,853,979	0.04	1,513,841,607	0.04	-1,377,987,628
configuration	359,712,522	0.11	7,787,249,741	0.23	-7,427,537,219
reduction	148,255,423	0.04	193,442,229	0.00	-45,186,806
idb_cache	365,087,943	0.11	52	0.00	+365,087,891
evaluation	58,306,774	0.01	3,805,843,802	0.11	-3 747,537,028
saturation	222,043,653	0.07	297,438,260	0.00	-75,394,607
user_space	257,179,423	0.08	11,406,353,178	0.34	-11,149,173,755

Table 5: Calls distribution among April's modules

The results in Table 5 show that the use of cache reduced the number of calls in all modules except for module `idb_cache` that implements the cache itself. The 365 million operations made by the cache module appear to be more expensive than the 30 billion operations whose execution were avoided by the coverage caching.

We further analyzed the profile logs trying to identify the predicates that were causing the inefficiency problems. Table 6 presents a summary of the number of calls for the predicates considered more relevant. Since the number of calls for most of the predicates decreased with the use of cache, we selected those predicates whose number of calls were still very high, or increased, or operate the Prolog database.

Table 6 shows that in the `prolog` module the number of calls increased only for the `assert`, `recorda`, `numbervars`, `copy_term`, and `ground` predicates. The increase of calls in the `idb_cache` module was most felt in the `idb_keys` predicate. All the other predicates in the `idb_cache` make calls to the predicates in the `prolog` module, in particular to the `recorded` predicate that YAP could not show in the profile statistics. From the profile results we estimated that the number of calls to the `recorded` predicate increased by around 22,456,790 when using coverage caching.

Since YAP does not provide the cumulative time spent computing each predicate, we did further experiments to measure the impact of each of those predicates in the execution time. We observed that the predicates that deal with the internal database and clausal database are the main source of execution time overhead. The slowdown caused by these predicates appears to be exponential with the increase of database entries. In particular, the dynamic predicate `idb_keys` and the database predicate `recorded` are those with biggest impact. These two heavily used predicates are the main cause for coverage caching inefficiency. As the reduction or elimination of Prolog database operations is not possible, a solution to cope with this problem could be to improve the indexing mechanism of the YAP Prolog internal database. Moreover, we find that it would be very much useful the support of an efficient indexing mechanism using multiple keys.

Predicate	cache=yes	cache=no	Variation
prolog:abolish/1	13,304	17,204	-3,900
prolog:assert/1	98,362	5,663	+92,699
prolog:assertz/1	1,592,288	2,049,054	-456,766
prolog:numbervars/3	5,265,269	4,349	+5,260,920
prolog:eraseall/1	5,902,918	7,734,758	-1,831,840
prolog:recordz/3	5,665,526	7,562,647	-1,897,121
prolog:copy_term/2	5,677,883	515,905	+5,161,978
prolog:call/1	6,396,015	8,314,571	-1,918,556
prolog:erase/1	20,674,230	24,155,488	-3,481,258
prolog:recorda/3	25,866,551	23,760,276	+2,106,275
prolog:integer/1	33,843,978	1,401,877,319	-1,368,033,341
prolog:set_value/2	41,545,320	1,411,971,018	-1,370,425,698
prolog:ground/1	110,305,158	90,361,520	+19,943,638
prolog:get_value/2	166,244,097	942,962,169	-776,718,072
idb_cache:idb_keys	5,166,049 (789,534)	0	+5,166,049

Table 6: Number of calls for some predicates. The `idb_cache:idb_keys` predicate is a dynamic predicate used to store cache keys. The value in parenthesis is the number of recalls.

6 Conclusions

This report presented an Inductive Logic Programming (ILP) system implemented in Prolog. ILP systems are non-classical Prolog applications because of the use of large sets of ground facts and high resource consumption (memory and CPU). Together with a description of the system implementation we provided results showing the impact on memory usage and execution time of a technique called coverage caching. This technique uses intensively the internal database to store results in order to avoid recomputation.

An empirical analysis of the coverage caching technique used by April running under the Yap Prolog showed that its use degrades the execution time although it significantly reduces the number of Prolog calls and retries. To pinpoint this unexpected behavior we profiled April using YAP Prolog. The analysis of the profile data obtained lead us to conclude that the use of YAP's database is the cause for performance degradation.

Improving the indexing mechanism of YAP Prolog internal database, moreover including efficient support for indexing with multiple keys, will certainly improve April's performance as well as other applications that use the database intensively. It is our hope that these findings will motivate Prolog implementors to further excel their implementations.

Acknowledgments

The work presented in this paper has been partially supported by project APRIL (Project POSI/SRI/40749/2001) and funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*. Nuno Fonseca is funded by the FCT grant SFRH/BD/7045/2001.

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] I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 1995.

[4] Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.

[5] 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.

[6] Rui Camacho. *Inductive Logic Programming to Induce Controllers.* PhD thesis, Univerity of Porto, 2000.

[7] James Cussens. Part-of-speech disambiguation using ilp. Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, 1996.

[8] 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.

[9] 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.

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

[11] L. Dehaspe and L. De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.

[12] T. Matsui, N. Inuzuka, H. Seki, and H. Itoh. Comparison of three parallel implementations of an induction algorithm. In *8th Int. Parallel Computing Workshop*, pages 181–188, Singapore, 1998.

[13] Y. Wang and D. Skillicorn. Parallel inductive logic for data mining. In *In Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*, Boston, 2000. ACM Press.

[14] Ricardo Rocha, Fernando M. A. Silva, and Vitor Santos Costa. Yapor: an or-parallel prolog system based on environment copying. In *Portuguese Conference on Artificial Intelligence*, pages 178–192, 1999.

[15] R. Rocha, F. Silva, and V. Costa. Yaptab: A tabling engine designed to support parallelism, 2000.

[16] Luc De Raedt. A perspective on inductive logic programming. In *The logic programming paradigm - a 25 year perspective*, pages 335,346. Springer-Verlag, 1999.

[17] 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 *Lecture Notes in Artificial Intelligence*, pages 3–18. Springer-Verlag, 2000.

[18] V. Costa, L. Damas, R. Reis, and R. Azevedo. *YAP Prolog User's Manual.* Universidade do Porto, 1989.

[19] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[20] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[21] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, February 1997.

[22] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[23] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

[24] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[25] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667, pages 3–20. Springer-Verlag, 1993.

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

[27] 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.

[28] S. Muggleton. Learning from positive data. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 358–376. Springer-Verlag, 1996.

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

[30] Simon Anthony and Alan M. Frisch. Cautious induction: An alternative to clause-at-a-time induction in inductive logic programming. *New Generation Computing*, 17(1):25–52, January 1999.

[31] N. Fonseca, R. Rocha, R. Camacho, and F. Silva. Efficient data structures for inductive logic programming. In T. Horváth and A. Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 130–145. Springer-Verlag, 2003.

[32] Nuno Fonseca, Fernando Silva, Rui Camacho, and Vitor S. Costa. Induction with April - A preliminary report. Technical report, DCC-FC & LIACC, Universidade do Porto, 2003. DCC-2003-02.

[33] C. Rouveirol and J-F. Puget. A simple solution for inverting resolution. In K. Morik, editor, *Proceedings of the 4th European Working Session on Learning*, pages 201–210. Pitman, 1989.

[34] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In S. Muggleton, editor, *Inductive Logic Programming*, pages 63–92. Academic Press, 1992.

[35] L. Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.

[36] S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.