Induction with April: A preliminary report

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

Technical Report Series: DCC-2003-02

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

Induction with April: A preliminary report

Nuno Fonseca, Fernando Silva, Rui Camacho,Vitor Santos Costa
DCC-FC & LIACC
Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
e-mail: {nf,fds}@ncc.up.pt,rcamacho@fe.up.pt,vsc@ncc.up.pt

May 2003

Abstract

Inductive Logic Programming (ILP) is concerned with the induction of first-order clausal theories. April is a new ILP system that can be classified as an empirical, non-interactive, single predicate learning system.

In this report we describe the architecture and implementation details of April together with a description of its features and an explanation of how to use it. We also propose the use in ILP systems of two efficient data structures: the Trie, used to represent lists and clauses; and the RL-Tree, a novel data structure used to represent clauses coverage list. We empirically evaluate the impact on April's performance of the proposed data structures, together with the impact evaluation of the coverage caching technique. April's development is an on going work. Although the results obtained are encouraging, there are still areas to improve. Areas for further research are identified.

1 Introduction

Inductive Logic Programming (ILP) [1, 2] is an established and healthy [3] subfield of Machine Learning (ML). The main goal of ILP is to investigate the inductive construction of first-order clausal theories and their justification from a set of examples and prior knowledge. As input an ILP system receives a set of examples (divided in positive and negative) of the concept to learn, and sometimes some prior knowledge (or *background knowledge*). Both examples and background knowledge are represented as logic programs. An ILP system tries to produce a logic program where positive examples succeed and the negative examples fail.

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 [4]. A more up-to-date list of applications of ILP systems to real world problems can be found in [5].

One major criticism of ILP systems is that they often have long running times. Thus, the need of efficient and scalable ILP systems is a important issue, specially as the number and complexity of applications domains increases. Several approaches have been proposed to improve ILP performance, such as several sequential execution efficiency improvements [6, 7, 8, 9, 10, 11], and parallelism [12, 13, 14, 15, 16, 17]. Understanding which techniques, or combination of techniques, contribute the most is a quite a difficult task.

April is a new ILP system that tries to combine and use the most number of techniques as possible. It aims to become an efficient, flexible, and scalable ILP system. Up to now only the flexibility and efficiency goals have been addressed, and are focused in this report. We plan to tackle the scalability problem in the near future trough parallelization and by integrating it with a Relational Database Management System (RDBMS). April combines ideas and features from several systems, such as Progol [18] et seq. [19, 20], Indlog [7], and CILS [21]. Besides integrating ideas and features from several ILP systems into a single coherent system, April incorporates novel data structures that improve efficiency (time and memory consumption).

In a nutshell we describe April as a non-incremental (empirical), non-interactive, single predicate learning system. April generates non-redundant theories, can handle non-ground background knowledge, can use non-determinate predicates, makes use of a strong typed language, and makes use of explicit bias declaration such as mode, type, and determination declarations. April, as several other ILP systems, is implemented in Prolog. 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 performance work that has taken place for Prolog [22, 23].

In this report we describe the architecture and implementation of April, together with a description of its features and an explanation of how to use it. We also propose the use in ILP systems of two efficient data structures: the Trie, used to represent lists and clauses; and the RL-Tree, a novel data structure used to represent clauses coverage list. We empirically evaluate the impact on April, in terms of execution time and memory usage, of the use of the RL-Tree and Trie data structures. We also evaluate the impact of a technique, called coverage caching [8], that stores previous results (in the prolog database) in order to avoid recomputation.

The remainder of this report is organized as follows. In the Section 2 we briefly introduce definitions and concepts necessary for the rest of the report. April's description and usage is given in Section 3. Section 4 presents some relevant implementation details. The experiments performed with April and the results obtained are presented in Section 5. Related work is described in Section 6. Finally, Section 7 concludes pointing future work.

2 ILP

This section briefly presents some basic concepts and terminology of Inductive Logic Programming but is not meant as an introduction to the field of ILP. For such introduction we refer to [24, 25, 26].

We start by formalizing the ILP problem, and move to the presentation of some relevant concepts and definitions. Finally, we present Mode-Directed Inverse Entailment (MDIE), the basis of April's induction algorithm.

2.1 ILP setting

From a logic perspective, the ILP problem can be defined as follows. Let E^+ be the set of positive examples (instances of the target concept), E^- the set of negative examples (non instances of the target concept) , $E = E^+ \cup E^-$, and B the background knowledge. In general, B, H, and E can be arbitrary logic programs. However is usual for E to be a set of Prolog atoms. The aim of an ILP system is to find a set of hypothesis (also referred as a theory) H such that some conditions holds.

In **normal semantics** (or normal setting) the following conditions must hold for H:

- **Prior Satisfiability** [24]: $B \wedge E^- \nvDash \Box$
- **Prior Necessity** [24]: $B \nvDash E^+$
- **Posterior Satisfiability** [24]: $B \wedge E^- \wedge H \nvDash \Box$ (Consistency)
- **Posterior Sufficiency** [24]: $B \wedge H \vDash E^+$ (Completeness)

- **Posterior necessity [7]:** $B \wedge h_i \models e_1^+ \vee e_2^+ \vee \ldots \vee e_n^+ (\forall h_i \in H, e_j \in E^+)$

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 the hypothesis to be inconsistent with a small number of negative examples. This allow ILP systems to deal with noisy data, that is, data that has inconsistencies.

2.2 ILP as a search problem

ILP can be mapped into a search through a space of hypothesis (designated as *hypotheses space*). The states in the search space are concept descriptions (hypotheses) and the goal is to find one or more states satisfying some quality criterion.

The ILP problem can be solved by the use of general Artificial Intelligence techniques like generate and test algorithms. However, due to the large and often infinite size of the search space, this approach is too computational expensive to be of interest. To tackle this problem the search space is structured by imposing a *generality order* upon the clauses. Such a partial order on clauses is usually denoted by $\preceq$, and the structured search space designated as generalization lattice. 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 commonly used are subsumption and logical implication. In both of these orders, the most general clause is the empty clause $\square$.

The subsumption order is the generality order most often used in ILP and is defined as follows:

Definition 1 *Let C and D be clauses. A clause C subsumes D, denoted by $C \preceq D$, if there exists a substitution θ such that $C\theta \subseteq D$.*

The ordering of the search space imposed by generalization and specialization allows a justifiable pruning of the search space. The pruning can be performed when:

- $B \wedge H \not\models e$ where e is positive evidence, because none of the specializations of H will imply the evidence.
- $B \wedge H \wedge e \models \square$ where e is positive evidence, because all generalizations of H will also be inconsistent with $B \wedge H$.

The search can be done in two directions: specific-to-general [27] (or *bottom-up*); or general-to-specific [28, 29, 18] (or *top-down*). In the generic-to-specific search the initial hypothesis is, usually, the more general hypothesis (i.e., $\square$). That hypothesis then repeatedly specialized through the application of rules of deductive inference in order to remove inconsistencies with the negative examples. In the specific-to-general search the examples, together with the background knowledge, are repeatedly generalized by applying inductive inference rules.

2.2.1 Inference Rules

The notions of generalization and specialization are incorporated in search algorithms as inductive and deductive inference rules.

Definition 2 *A deductive inference (or specialization) rule r maps a conjunction of clauses G into a conjunction of clauses S such that $G \models S$.*

Definition 3 *A inductive inference (or generalization) rule r maps a conjunction of clauses S into a conjunction of clauses G such that $G \models S$.*

Inference rules define what can be inferred from what. Since a "blind" application of inference rules is very inefficient, inductive logic systems employ *operators* to control the application of inference rules. An operator expands a node in the search space into a set of successor nodes.

Definition 4 *A specialization operator [24] maps a conjunction of clauses G onto a set of maximal specializations of S. A maximal specialization S of G is a specialization of G such that G is not a specialization of S, and there is no specialization S' of G such that S is a specialization of S'.*

Definition 5 *A generalization operator [24] maps a conjunction of clauses S onto a set of minimal generalizations of G. A minimal generalizations G of S is a generalization of S such that S is not a generalization of G, and there is no generalization G' of S such that G is a generalization of G'.*

The Shapiro's MIS [28] system was the first system to use the concept of specialization (refinement) operator for clauses. MIS operators were based on the notion of the specialization rule under subsumption (see Definition 4) with the restriction that G and S contain a single clause. Refinement operators basically employ two syntactic operations on a clause:

1. apply a substitution θ to the clause;
2. add a literal (or a set of literals) to the clause.

Another type of operator is the generalization operator, that is suited for a a bottom-up search. A generalization operator maps a clause S onto a set of clauses that are generalizations of G. Generalization operators perform two basic syntactic operations on a clause:

1. apply an inverse substitution to the clause;
2. remove a literal from the body of the clause.

An ideal refinement operator should be locally finite, complete, and proper [30]. A refinement operator is locally finite if it generates all successors of an hypothesis in the search space and the set of successors is finite. A refinement operator is complete is it generates all hypotheses in the search space by applying the operator repeatedly to the most general clause. A refinement operator is proper if it does not generate equivalent clauses. The refinement operators for full clausal languages have to drop one of the properties of idealness and it is usually the properness property that is sacrificed.

Another concept of an ideal refinement operator is that it should be optimal [24]. An operator is *optimal* when it generates an hypothesis exactly once. Non-optimal refinement operators generate all candidate hypothesis more than once, getting trapped in recomputing the same things several times.

To restrict the application of inference rules is usual to impose further conditions to the operators besides completeness. One of those conditions require that the generated hypothesis satisfy the language bias (described in the Section 2.3).

2.2.2 Justification

During the search through the hypothesis space, an ILP system generates and evaluates candidate hypotheses. Since, there is more than one candidate hypothesis it is necessary to determine which is the best candidate. Usually the hypotheses are scored using a statistical approach or information theory approach. The candidate with best score is then selected.

The score functions evaluate essentially two parameters, the accuracy or coverage of an hypothesis, and its transparency. The *accuracy* is the percentage of examples correctly classified by the hypothesis. The *coverage* of an hypothesis H is the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from $B \wedge H$. The transparency of an hypothesis denotes is readability to humans. For instance, the readability of an hypothesis can be measured by taking into account the number of literals it contains.

Example 1 (Score function) *Let H be an hypothesis, p the number of positive examples covered by H, n the number of negative examples wrongly covered by H, and l the number of literals in H. A example of a score function may be: $f(p,n,l) = y - n - l$.*

2.3 Bias

Bias was initially defined by Michell [31] as "any basis for choosing one generalization over an other, other than strict consistency with the instances". Its important to note that ILP is a complex problem due to the large and potentially infinite size of the search space. Practical ILP systems attenuate the complexity of the problem by imposing all sorts of restrictions, mostly syntactic, on candidate hypotheses to reduce the search space. Such restrictions are called **bias**. Bias defines the hypothesis space and is central to address efficiency. Even a biased hypothesis space can be too extensive for a complete search to be performed

The notion of inductive bias can be organized in three different types [32]: language bias; search bias; preference bias. *Language bias* determines the hypothesis space of the possible concept descriptions, defining the target concept language. In ILP, the concept description language is restricted to Horn clauses. *Search bias* determines which part of the hypothesis space is searched, and how it is searched. It can be through a restriction or preference bias. The *restriction bias* determines which hypothesis should be ignored, while the *preference bias* determines which hypothesis should be considered first. Some examples of search bias are the example selection criteria and the operator used for induction. *Validation bias* establishes an acceptance criterion for the learning system, telling it when the search should stop. This could happen, for instance, when the hypothesis is complete and consistent with the given set of examples.

A bias is *declarative* if it is explicitly represented. A declarative representation of the bias is required so bias setting and shifting can be easily performed. The declarative representation of bias may be achieved by the use of languages that allow bias specification, or by configurable generic methods that allow both specification and implementation of bias. Declarative bias may be used by ILP systems to be more adaptable to particular learning tasks.

The declarative bias used in most ILP systems can be divided [24] in two types: *syntactic bias* and *semantic bias*. Both types can be considered as a particular case of the language bias. Syntactic bias imposes restrictions on the clauses allowed in the hypothesis at the syntactic level. Semantic bias imposes restrictions on the meaning, or behavior of the hypothesis.

Two examples of declarative bias are the predicate mode and determinacy, that are introduced in the next section. The first may be classified in the semantic bias category, and the second in the syntactic bias.

2.4 Mode-Directed Inverse Entailment

Mode-Directed Inverse Entailment [18] (MDIE) is a technique widely used in ILP that uses inverse entailment together with mode restrictions to find a hypothesis H. To explain the main idea behind inverting entailment, let us take the specification of ILP problem: given background knowledge B and positive 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 positive 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 $\perp$.

2.5 ILP systems classification

ILP systems can be classified based on some characteristics like the type of bias employed, the ability to invent new predicates, and the heuristics employed to handle noisy data.

The classification of ILP systems is done in several dimensions, being the main ones: incremental/non-incremental; interactive/non-interactive; single/multiple predicate learning/theory revision.

A ILP system is *non-incremental* (or *empirical*) if the examples are given at the start and do not change afterwards. In *incremental* systems the examples are provided by the user, one example at the time. The incremental systems typically perform a search using generalization and specialization techniques, while non-incremental systems use only one of the techniques.

A system is *interactive* when it poses questions to an oracle (i.e. the user) about the intended interpretation. The answers to the queries may be used to prune large parts of the search space. A system that is not interactive is *non-interactive*.

In *single predicate learning* from examples, the evidence E is composed of examples for one predicate only. In *multiple predicate learning* the aim is to learn a set of possibly interrelated predicate definitions. *Theory revision* is usually a form of incremental multiple predicate learning, where the system starts with a initial approximation of the theory.

3 April

This section describes the use and features of the April system. April is a non-incremental (empirical), non-interactive, single predicate learning system. It generates non-redundant theories, handles non-ground background knowledge, uses non-determinate predicates, and makes use of a strong typed language and of explicit bias declaration such as mode, type, and determination declarations.

The system aims at being an efficient, flexible, and scalable ILP system. April aims to be an efficient system through low memory consumption and by providing low response time. Flexibility is achieved by a modular implementation (described in the next section) and by providing the user with a high level of customizations. This customization allows April to emulate other systems through a change of the configuration settings. However, we should note that April's emulation capabilities are no substitute for an exact implementation of the original algorithm.

Up to now only the flexibility and efficiency goals have been addressed. We plan to tackle the scalability problem in the near future by parallelizing April and by conclude the development of a RDBMS interface (so learning can be done directly from the database).

April is implemented in Prolog and runs on the YAP [33] Prolog compiler. The latest version of April is available from `http://www.ncc.up.pt/~nf/April` in a *tarball* file containing source code and example files.

3.1 Setting up April

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 `member(Number,List)` predicate, that checks if a Number is in the List, and some background knowledge, April can construct a definition of the `member` predicate. Before using April to induce the member predicate definition (a theory) is necessary to prepare some input files.

To induce the member predicate definition, or in general to induce a theory, April requires a number of input files that setup the problem being solved and contain configuration parameters. The files describing the background knowledge and the positive examples are mandatory, while all the others are optional.

The **background knowledge file** encodes information, in the form of Prolog clauses, relevant to the domain of the concept to be learned. It can also contain directives understood by the Prolog compiler being used (for example, `:- consult(someotherfile).`). This file can also contain language and search restrictions for April (see Section 3.4 for a list and description of the parameters available). Even though the system has most of its parameters predefined, each background knowledge file must contain mode, type, and determination declarations. The background file should have a extension **.b**. An example of a background file is given in Example 2.

Example 2 (member.b)

```
:- determination(member/2,member/2).
:- determination(member/2,n/1).
:- modeh(1,member(+int,+list )).
:- modeb(1,n(+int)).
:- modeb(1,member(+int,+list)).
:- modeb(1,+int=+int).
:- modeb(1,+list=+list).

:- typestructure(list,[int|list]).
list([]).
list([Int|List] ) :-
        integer(Int),
        list(List).

int(Number):- integer(Number).
```

The **positive examples file** should contain instances of the concept to be learned encoded as Prolog ground facts. The filename should have the extension ".f" and the name should be the same as that used for the background knowledge. Example 3 is an example of a positive examples file.

Example 3 (member.f)

```
member(0,[0]).
member(2,[2]).
member(3,[2,3]).
member(3,[4,2,3]).
member(5,[4,2,3,5]).
```

The **negative examples file** should contain non instances of the concept to be learned also encoded as Prolog ground facts. April is capable of learning from positive examples only, so the negative examples file is not mandatory. The filename should have the extension ".n" and the filename should be the same as that used for the background knowledge.

Example 4 (member.n)

```
member(0,[1,2]).
member(1,[3]).
member(3,[]).
member(3,[1,2]).
member(3,[1,2,4]).
member(0,[1]).
member(0,[4]).
```

A more compact way of expressing negative information can be encoded through the use of non-ground constraints. Such constraints (see Section 3.5.4) can be specified in the background knowledge file.

Another input file is the **settings file**. This file is optional and it is intended to store April' configuration parameters and therefore having them completely separated from background knowledge. An example of a parameters setting file is given in Example 5.

Example 5 (member.s)

```
:- set(nodes,10000).
:- set(noise,0).
```

The **weights file** is optional and is intended to store initial example weights to be used by the weighted_coverage heuristic (explained in Section 3.5.2). Its name should be the same as the background file but with the extension ".w". A different filename can be defined through the parameter ***weights_file***. For each example, its order number, class (pos or neg), miss classifications, and weight are provided to April through the weight/4 predicate (see Example 6).

Example 6 (member.w)

```
weight(1,pos,0,0.59375).
weight(2,pos,1,1.1875).
weight(3,pos,1,1.1875).
```

3.2 Running April

April is executed in the command line through the invocation of the executable `sapril`.

The syntax of `sapril` is: *sapril datasetname* [*settings file*].

April can be used to induce a predicate definition for the `member` relation (described previously) once the background file (**member.b**), positive (**member.f**) and negative (**member.n**) examples files are saved in a directory. Assuming that the referred files are in the current directory, April can be executed to generate a theory by typing `sapril member` on the command line. Note that it is only necessary to pass the name of the background file without any extensions. April will automatically look for all files in the current directory.

The output of executing the above command, including the theory found, is presented in the Appendix A. After inducing a theory, April saves the theory found in a file, together with some extra information.

3.3 Simplified Algorithm

April's algorithm is based on the MDIE (presented in Section 2). The main steps in the April's algorithm are as follows:

1. **Example Selection**: Select an uncovered positive example e to be generalised. If none exists, stop.

2. **Saturation step**
   Construct the most-specific-clause $\perp_{(e,B,C)}$ that entails the example selected, and is within language restrictions provided (C). The most-specific-clause is usually a definite clause with many literals, and is called the "bottom clause" [18].

3. **Reduction Step**
   Find a clause more general than the bottom clause. This is done by performing a general-to-specific search in the subsumption lattice bounded bellow by $\perp_{(e,B,C)}$. The clauses' bodies generated during the search are subsets of the literals from the bottom clause.

4. **Cover Removal**
   The clause with the best score is added to the current theory and all positive examples made redundant are removed.

April has several example selection procedures. The choice of the example selection procedure is done through the parameter ***sat_example*** (see Section 3.4).

Like other systems based on MDIE (eg. Progol [18, 34] or Indlog [7]) April searchs a bounded sub-lattice for each example e_i relative to background knowledge B and constraints C. The sub-lattice has a most general element $\top$ (the empty clause $\Box$) and one least general element $\perp_i$ such that $B \wedge \perp_i \wedge \neg e_i \vdash \Box$.

The top-down search performed by April traverses the generalization lattice starting with the most general clause (having the same predicate symbol of the target concept). The search then proceeds by specializing the current clause by adding literals to its body or binding variables. During this process, the coverage of the generated hypothesis is computed. The goal of the search is to find the best hypothesis, i.e., the hypothesis that covers the maximum number of positive examples as possible and is consistent with the negative examples.

Most of the time spent by April, and other ILP systems, is in the reduction step computing hypothesis coverage. To speed up this step, April uses techniques like lazy evaluation of examples and coverage caching that will be explained later. These optimizations can be turned on and off by using the meta-language.

3.4 Meta-language

The *meta-language* provided by April allows declarative bias specification. The meta-language features include determination declarations, mode and type declarations, background predicates' properties, and facilities to change system parameters.

3.4.1 Determination declarations

Determination declarations [35] specify, for each predicate symbol, which other predicate symbols can appear in its definition. They take the form `determination(`*`TargetName/Arity1`*`,`*`BackgroundName/Arity2`*`).` The first argument is the name and arity of the target predicate (i.e. the predicate that appears in the head of hypothesised clauses). The second argument is the name and arity of a predicate that can appear in the body of such clauses.

Typically there will be many determination declarations for a target predicate, corresponding to the predicates thought to be relevant to the target predicate. April does not construct any clauses if there is no determination declarations. Since April is a single predicate learning system, it can only accept one target predicate at a time. If multiple target determinations are provided by the user, the first one is chosen. An example of determination declaration can be seen in Example 2 on page 8.

3.4.2 Mode declarations

Mode declarations specify the mode of call for predicates that can appear in any clause induced by April. These declarations specify the arguments' types, and also indicate if they are intended to be an input or an output argument. There may be more than one mode declaration for each predicate symbol except to head of the target predicate.

Mode declarations take the form `modeh(1,`*`PredicateMode`*`)` for the head of the target predicate, and `modeb(`*`RecallNumber`*`,`*`PredicateMode`*`)` for the literals that may appear in the body of an hypothesis clause. The number of possible outputs, for each combination of input arguments, is limited by the *`RecallNumber`*. *`RecallNumber`* can either be a number specifying the number of successful calls to the predicate, or * specifying that the predicate has bounded non-determinacy. It is usually simpler to specify *`RecallNumber`* as * with the side effect that the system may become slower.

PredicateMode specifies the arguments mode of a predicate. It has the form:

`predicatename(` *`ModeType`* `,` *`ModeType`*`...).`

Each *`ModeType`* is either a simple or structured. A simple *`ModeType`* is one of the form:

- *`+T`* specifying that when a literal with predicate symbol `predicatename` appears in an hypothesised clause, the corresponding argument should be an "input" variable of type T
- *`-T`* specifying that the argument is an "output" variable of type T
- *`#T`* specifying that it should be a constant of type T

A structured ModeType is of the form *`f(..)`* where *`f`* is a function symbol and each argument is either a simple or structured ModeType.

Types have to be specified for every argument of all predicates to be used in constructing an hypothesis. This specification is done within a `mode(...,...)` statement. By default April does not perform type-checking, but it can be activated through the parameter ***typechecking***. An example of a type structure is given in Example 2.

Here are some examples of mode declarations:

Example 7 (Mode declarations)

```
:- mode(1,dec(+integer,-integer)).
:- mode(1,mult(+integer,+integer,-integer)).
:- mode(1,plus(+integer,+integer,-integer)).
:- mode(1,(+integer)=(#integer)).
:- mode(*,has_car(+train,-car)).
```

3.4.3 Background predicate's properties

April allows the user to indicate some properties (commutative and equivalence) of the predicates, in the background knowledge, that are used in the construction of the bottom clause. By taking advantage of these properties, April is able to reduce the bottom clause size, and thus the size of the search space.

Currently available predicate declarations and their syntax are:

- Equivalence: `equiv(`*`predicatename1(arguments)`*`,`*`predicatename2(arguments)`*`)`

- Commutativity: `commutative(`*predicatename/arity*`)`

The equivalence declaration may be used to indicate April that two terms are equivalent. For instance, `equiv('=<'(A,B),'>='(B,A))` tells April that it does not need to have both terms in the bottom clause.

The commutativity declaration may be used to inform April that the arguments of a term are commutative. Consider for example the declaration `commutative('='/2)`. This declaration informs April that literals of the equality predicate may have the arguments commuted. Commutativity declaration can be seen as a special case of the equivalence declaration where *predicatename1*=*predicatename2* and the arguments variables are switched. For example, the previous commutative declaration could be rewritten has `equiv('='(X,Y),'='(Y,X))`. Although the commutativity declaration can be represented by the `equiv` declaration it is available in April for two reasons: first for simplicity, the commutative declaration is more compact and simple than the equivalence declaration; and secondly, for compatibility with existing systems (e.g. Indlog) and previous versions of April.

3.4.4 Parameters

We consider the system parameters as part of the declarative bias because some of the parameters constraint and change the search performed by April. Hence, by changing the parameters values the user may change the search bias [32] or the restriction bias [32].

The `set/2` predicate is used for setting April's parameters values.

`set(Parameter,Value)`

April's most important parameters are described in Table 1 and Table 2. Table 1 presents the safe parameters. A parameter is considered "safe" if changing its value does not alter the induced theory produced by April. However, the safe parameters may have a significant impact on April's performance. Table 2 describes other parameters that are, or may be in certain conditions, not safe.

3.5 Altering the search

The search performed by April may be user specified by setting some appropriate parameters. This section presents ways to alter the search.

3.5.1 Search strategies

The search for individual clauses is mainly affected by two parameters: ***search*** and ***heuristic***. The ***search*** sets the search strategy that will be used to induce clauses and the ***heuristic*** parameter sets the evaluation function used to measure the quality of the hypothesis generated.

The following search strategies are available in April:

- **bf**: Enumerates shorter clauses before longer ones. At a given clause length, clauses are re-ordered based on their evaluation. This is the default search strategy;

- **best-first**: Enumerates clauses in a best-first manner using the heuristic function defined by parameter ***heuristic*** to evaluate the clauses.

Parameter	***Possible Values***	***Description***
cache	true or false (true)	If set to true then clauses coverage are cached for future use.
cache_storage	list, rl (rl)	Selects the structure used to keep clauses' coverage: interval lists or RL-Trees (see Section 4.7 for more information).
optimise_clauses	true or false (false)	If true performs query optimisations as described in [9].
record	true or false (false)	If true then April's execution output (defined by ***verbose*** parameter) is written to a file. The filename is given by ***recordfile***.
recordfile	Prolog atom (record)	Sets the filename where the execution output is going to be written. Only makes sense if record is set to true.
verbose	Integer >= 0 (2)	Sets the level of verbosity of April's output messages.
trace	Integer >=0 (0)	Level of information added to the generated trace file for VisAll [36] (0 disables trace file creation).
use_tries	yes or no (no)	Defines if the Trie data structure [37] is used to pack/compress data associated to the search space.
clean_tries	yes or no (yes)	Defines if the Tries are cleaned at the end of a reduction step. This parameter is only used if ***use_tries*** is set to yes.
save_theory	yes or no (yes)	If set to yes, April saves the theory found into a file.

Table 1: April's safe parameters

3.5.2 Heuristic functions

The heuristic function is defined by the parameter ***heuristic***. The value given by the heuristic function to a clause is used to order the clauses.

Next we enumerate the available heuristic functions in April, where the constants we use have the following meaning: P is the number of positive examples covered by the clause; N is the number of negative examples covered by the clause; L is the number of literals in the clause; U the number of variables in the clause head that are unbound.

- **positive**: Clause utility is simply P;
- **coverage**: Clause utility is $P - N$;
- **coverage_l**: Clause utility is $P - N + L$. The idea with this function is to give greater score to bigger clauses;
- **compression**: Clause utility is $P - N - L + 1$.
- **compression2**: Clause utility is $P - N - L * (U + 1) + 1$. This is a variant of the compression evaluation function;
- **progol**: Clause utility is $P - L - U$. This function is based on the Progol evaluation function described in [18];
- **laplace**: Clause utility is $(P + 1)/(P + N + 2)$;
- **l**: Clause utility is L;
- **acc**: Clause utility is $P/TP - N/TN$, where TP and TN are, respectively, the number of positive and negative examples;

Parameter	***Possible Values***	***Description***
clauselength	Positive integer (4)	Sets an upper bound on number of literals acceptable in a clause.
h	Positive integer (10)	Sets the maximum depth of the proof tree carried out by the theorem-prover.
i_determinancy	Positive integer (2)	Bounds the number of interactions carried out in the bottom clause construction.
nodes	Positive integer (2000)	Sets an upper bound on the nodes to be explored when searching for an acceptable clause.
noise	Positive integer >= 0 (0)	Sets an upper bound on the number of negative examples allowed to be covered by an acceptable clause.
search	bf, bestfirst (bf)	Sets the search strategy.
explore	true or false (false)	If true then forces search to continue up to the point where all remaining elements in the search space are definitely worse than the current best element. Otherwise, the search stops when it is certain that all remaining elements are no better than the current best.
language	Integer >= 0 (0)	Specifies the maximum number of occurrences of a predicate symbol in any clause. A value of 0 disables this parameter. This parameter is based on Camacho's ILLS [7].
language_init	Integer >= 1 or inf (1)	Specifies the initial language level.
lazy_eval	disabled,pos,neg,all (disabled)	Specifies if and what kind of lazy evaluation is performed when computing clause coverage. The use of lazy evaluation of negatives does not alter the theory generated.
minacc	Floating point number between 0 and 1 (1.0)	Sets a lower bound on the minimum accuracy of an acceptable clause.
targetacc	Floating point number between 0 and 1 (1.0)	Sets a lower bound on the minimum accuracy of an acceptable theory. April stops once theory accuracy reaches ***targetacc***.
mincover	Positive integer (0)	Sets a lower bound on the number of positive examples to be covered by an acceptable clause. The value 0 disables the parameter. This parameter can be used to prevent ground unit clauses to be added to the theory (by setting its value to 2).
minpcover	Floating point number between 0 and 1 (0)	Sets a lower bound on the positive examples covered by an acceptable clause as a fraction of the positive examples covered by the head of that clause. If the best clause has a ratio below this number, then it is not added to the current theory. Note that the use of this parameter together with the ***mincover*** parameter may result in unexpected behavior.
samplesize	Integer >= 0 (0)	Sets the number of examples selected (using the method defined by the parameter ***sat_example***) to be used in the induction step. A value of 0 turns off sampling and all uncovered examples are used.
sat_example	first, random, weight	Example selection strategy for the saturation.

Table 2: April's parameters

- **acc-ul**: Clause utility is $P/TP - N/TN - L*(U+1)$, where TP and TN are, respectively, the number of positive and negative examples.
- **weighted_coverage**: Clause utility is the same as the coverage function with the difference that examples have weights [38]. Initial weights can be defined in a weights file (see Section 3.1).

3.5.3 Pruning

Pruning is used to exclude clauses and their refinements from the search. It is very useful for stating which kinds of clauses should not be considered in the search. The use of pruning greatly improves the efficiency of ILP systems since it leads to a reduction of the size of the search space.

Two types of pruning can be distinguished within April, built-in and user-defined pruning. Built-in, or internal pruning, refers to pruning implemented in April that performs admissible removal of clauses from a search, and is currently available for all evaluation functions. User-defined prune statements can be written to specify the conditions under which a user knows that a clause (and its refinements) could not possibly be an acceptable hypothesis. Such clauses are pruned from the search. The `prune` definitions are written in the background knowledge file using rules of the form `prune((ClauseHead:-ClauseBody)) :- Body`.

The following example is from a pharmaceutical application that states that every extension of a clause representing a "pharmacophore" with six "pieces" is unacceptable, and that the search should be pruned at such a clause.

Example 8 (Pruning declaration)

```
prune((Head:-Body)) :-
        violates_constraints(Body).

violates_constraints(Body) :-
        has_pieces(Body,Pieces),
        violates_constraints(Body,Pieces).
violates_constraints(Body,[_,_,_,_,_,_]).

has_pieces(Body,Pieces):-...
```

3.5.4 User-defined constraints

April accepts the definition of Integrity Constraints (IC) that should not be violated by a hypothesis. Integrity Constraints can be used to impose syntactic or semantic properties on the hypothesized clauses. They are are written in the background knowledge file.

The constraints are defined by rules of the form:

`is_constraint(HypothesisBody):- ConstraintBody.`

or

`constraint(HypothesysHead,HypothesisBody):- ConstraintBody.`

ConstraintBody is a set of literals that specify the condition(s) that should not be violated by hypotheses found by April.

Note that negative examples are a special case of integrity constrains. If April applies a constraint successfully to a hypothesis then it will be considered inconsistent and will not be accepted. Note also that an integrity constraint does not state that the refinement of an hypothesis that violates one or more constraints will also be unacceptable. To achieve this pruning should be used.

The following example is from a pharmaceutical application that states that hypotheses are unacceptable if they have fewer than three "pieces".

Example 9 (Constraint declaration)

```
constraint(Head,Body):-
        has_pieces(Body,Pieces),
        length(Pieces,N),
        N =< 2.
```

3.5.5 User-defined refinement

April allows the user to redefine the refinement operator by specifying a definition for the predicate `refine/2` that states the transitions in the refinement graph traversed during a search. The `refine` definition is written in the background knowledge file in rules of the form:

`refine(Clause1,Clause2):- Body.`

This specifies that *Clause1* is refined to *Clause2*. The definition can be nondeterministic, and the set of refinements for any clause is obtained trough backtracking. *Clause2* may have cuts ("!") in its body. If the parameter ***construct_bottom*** is not set to false then, for any refinement, April ensures that *Clause2* implies the current most specific clause.

The parameter ***refine*** is used to select the refinement mode. April accepts two refinement modes: auto and user. If ***refine*** is set to *auto* then is performed an automatic enumeration of all clauses acceptable by the meta-language. The result is a top-down search starting from the empty clause. If the parameter ***refine*** is set to *user* then the user must specify a domain-specific refinement operator with `refine/2` statements as described previously.

4 Implementation Details

April is implemented mainly in Prolog and runs in the YAP [33] Prolog compiler. By using a Prolog compiler like YAP, April takes advantage of its tested and fast deductive engine. YAP implements some advanced techniques in Logic Programming, such as implicit and explicit parallelism [22], and tabling [23], that could contribute to improve April's response time.

The choice of using Prolog, and Prolog engines, also carries some drawbacks. One of such drawbacks is the inability of efficiently implement complex data structures in Prolog. To circumvent this limitation, some data structures have been implemented in C to improve response time and reduce memory consumption. An example of such data structures are the RL-trees and Tries [37] data structures. RL-Trees are described in Section 4.7.1. The Trie data structure used in April was implemented by Ricardo Rocha and is available in YAP as an external module. An essential property of the Trie structure is that common prefixes are represented only once. April exploits inherently and efficiently the similiarities among the hypotheses by using Tries to store the hypothesis and related information.

Research in improving the efficiency of ILP systems has been focused in reducing their sequential execution time, either by reducing the number of hypotheses generated (see, e.g., [32, 11]), or by efficiently testing candidate hypotheses (see, e.g., [6, 10, 9, 7]). Several techniques to improve the sequential performance have been implemented, such as coverage caching, incremental language-level search [7], clause transformation [10, 9], and a strong declarative bias.

Another way of improving the response time of ILP systems, besides improving their sequential efficiency, is through parallelization. The parallelization strategies used so far in ILP can be divided in four types: parallel exploration of independent hypothesis [14]; parallel exploration of the search space [13, 14, 15], parallel coverage test [14, 16, 17]; parallel execution of an ILP system over a partition of the data [12, 13, 16, 17]. As pointed out by Page [39], and confirmed by research results [12, 13, 14, 15, 16, 17], parallelization of ILP systems is a research direction that must be pursued. This line of research will be followed by us in the next phase of April's development.

In the rest of this section we present April's module architecture, a simplified description of its main algorithms, and its refinement operator. We also provide a descrition of two optimization

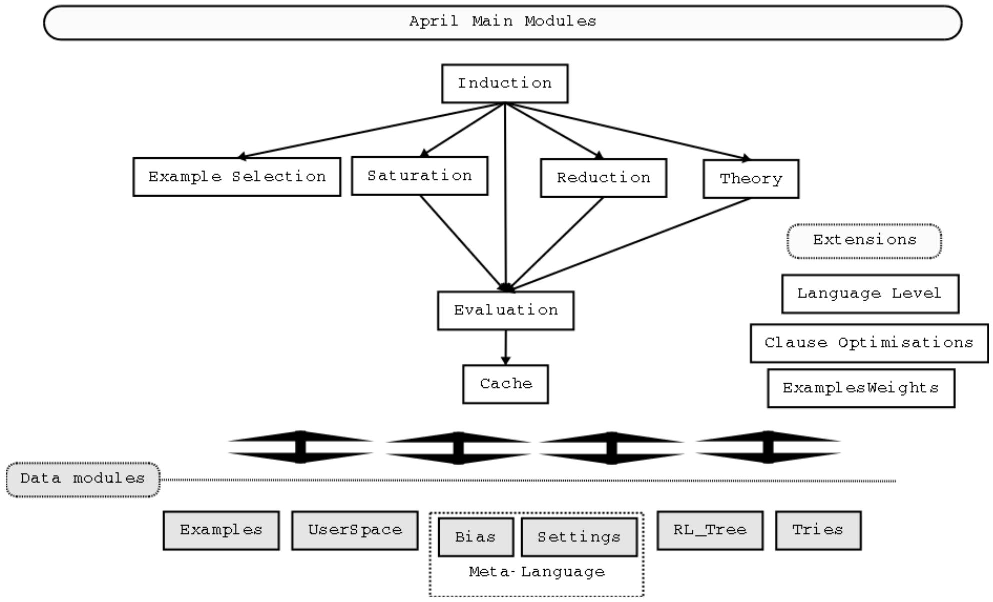

Figure 1: Module Architecture

techniques: lazy coverage evaluation and coverage caching.

4.1 April Module Architecture

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

Figure 1 presents April's module architecture. We divided the modules in two major types: data modules and functional modules. The data modules are used to store data, while the functional modules implement an algorithm or some functionality. A third type of modules, the extension modules, include modules that implement ideas available in other systems, or described in papers. Currently, there are three extension modules: the *language level module*, that implements Camacho's Incremental Language Level Search [11]; the *clause optimisations* module, that implements the optimisations described by Costa et al. [9]; and the *ExamplesWeights* module, that provides functionalities to allow examples to have weights [38].

The *induction* module implements April's main algorithm. The *example selection module* aims to provide the functionalities for selecting an example to be used in the *saturation module*. The *reduction module* performs the search through the hypothesis 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 coverage computation algorithm implemented in this 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. The *examples module* stores the examples provided by the user. The examples may be stored in YAP clausal database or, in the future, in a relational database system. The bias declarations provided by the user are stored in the *bias module* and *settings module*. The *UserSpace module* stores all background knowledge provided by the user and the clauses committed by the theory module to the final theory. The RL-tree and tries data structures are described in Section 4.7.

The main modules are described in more detail in the following subsections.

4.2 April 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 definite clauses; 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 [2].

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 */
   j = 1
   do                                                    /* Clause generation cycle */
      e⁺_i = select_example(E⁺_cur, C, SampleSize, j)
      ⊥ = saturate(B, H, C, e⁺_i)                        /* See Section 4.3 */
      h_i = reduction(⊥, B, H, C, E⁺_cur, E⁻, Best)      /* See Section 4.4 */
      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 */
```

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 are "mutations" of the one presented. The outer cycle contains two inner cycles that we called *clause generation cycle* and *clause consumption cycle*. The idea behind these two cycles is similar to the cautious induction method implemented in CILS [21]. As in CILS, April first generates a set of candidate clauses (clause generation). April then selects the hypothesis with higher quality and adds them to the theory. The difference to CILS is that April performs a more greedy selection, that may result in a slightly worst quality of the final theory. On the other hand April can learn recursive predicates while CILS does not. The outer cycle ends when there are no positive examples left or a stopping condition is 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

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

[2] Consistency and completeness conditions may be relaxed by constraints C

or randomly from E^+_{cur} (the choice is made by the user). The selected example e^+_i is saturated and flattened (see Section 4.3) using B and C, generating the $\perp$ (bottom clause). A clause (hypothesis) is generated by using the reduction Algorithm 2. The *Best* clause is used in the reduction 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. This pool contains the clauses found ordered by the number of positive and negative examples covered.

The clause consumption cycle tries to consume the clauses found previously, i.e. add the clauses to H. The best clause in the pool is added to H. The examples covered by *Best* are removed from E^+_{cur}, and *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.

4.3 Saturation

Saturation aims at constructing of the bottom clause $\perp$ that corresponds to the bottom of generalization lattice. This is the first step to remove useless clauses from consideration when searching for an hypothesis. The task in saturation is to gather all “relevant” ground atoms that can be derived from $B \wedge \neg e^+_i$ and satisfy the constraints C. The bottom clause generated will contain all literals that may be found in the clauses generated during the search.

April uses a slightly modified version of Progol's saturation algorithm to construct the bottom clause [34]. All relevant atoms collected during the construction of the bottom clause are flattened [40] and input or ouput arguments of the literal are transformed into skolemized variables. Then the new atom (with skolem variables) is recorded, to be used in the refinement, together with its i-depth and an identifier (auto-number). The skolemization performed by April is the main difference to the original saturation algorithm and makes unnecessary the use of substitutions lists for the literals in the reduction step (e.g. as performed by Aleph and Indlog), because the substitutions are implicit in the skolemized literals. The identifier number is used, in the reduction step, to impose an ordering to the literals that appear in a clause. Like other ILP systems, April flattens all function symbols by introducing equalities. The equalities are internally represented by April as `sat_eq(Var,Value)`. Flattening is a method to make a theory function-free and was introduced in ILP by Rouveirol [41, 40, 42].

Example 10 shows a bottom clause, as seen by the user and how it is internally represented using literal identifiers.

Example 10 (Bottom Clause)
Example of a bottom clause generated by April (for example `member(3,[2,3])`*):*

```
member(A,B) :-
   sat_eq(A,3), sat_eq(B,[2,3]), sat_eq(B,[C|D]), sat_eq(D,[A|E]),
   member(A,D), sat_eq(F,[A|G]), sat_eq(G,[E|E]), member(A,F), member(C,B).
```

Internal representation of the $\perp$ clause using literals identifiers:

```
0 :- 1, 2, 3, 4, 9, 10, 11, 12, 14.
```

4.4 Reduction

The bottom clause generated in the saturation step is the most specific clause ($\perp_i$) that subsumes e^+_i relative to the background knowledge B. Thus, for the example e^+_i, the search for an acceptable hypothesis is limited to the bounded sub-lattice $\Box \prec H \prec \perp$. The bottom clause is often too specific to be of interest because it sometimes just subsumes e^+_i. For this reason $\perp_i$ must be generalized.

The search in the subsumption lattice is made using a top-down approach, starting with the more general clause (that has the same predicate symbol of the target concept).

Algorithm 2 Reduction algorithm

Input :$\perp_i$ /* Bottom clause */
B /* Background knowledge */
H /* Current theory */
C /* Set of constraints */
E^+_{cur} *and* E^- /* Examples */
$Best$ /* Best clause */
Output:h_i /* Best clause seeded with e^+_i */
$MaxLangLevel = C(language\_level)$
$CurLevel = C(init\_level)$
$searchMethod = C(search)$ /* Search method used (bestfirst or breadth-first) */
$h_i = Nothing$
$Open = \{RootNode(\perp_i)\}$
repeat
  $h_i = search(searchMethod, Open, CurLevel, B, H, C, E^+_{cur}, E^-, Best)$
  if $h_i! = NULL$ **then** *break*
  $CurLevel = CurLevel + 1$
  if $CurLevel > C(MaxLevel)$ **then** *break*
end repeat

The Algorithm 2 generates a clause h_i given $\perp_i$, background knowledge B, a set of constraints C, and the best clause found so far. The search for h_i is done using a best-first or breadth-first strategy. During the search the clauses are generated by the refinement operator described in Section 4.5. For each clause generated the system computes its coverage, unless the clause violates some user constraint. The goal of the search is to find the shortest hypothesis that covers the maximum number of positive examples and is consistent with the negative examples. If a clause coverage is inferior to the coverage of $Best$ clause found so far, then it is pruned. Other safe pruning is made to attempt to reduce the search space. For instance, if a clause positive coverage is lower than the value defined in the parameter ***minpos*** then it is pruned.

April implements the Incremental Language Level Search [11] strategy. Currently, this feature is not efficiently implemented for the reason that we now explain. If a candidate hypothesis h_i is not found on the current language level, all clauses generated in that unsuccessfully search must be generated again and their coverage recomputed (if caching coverages is disabled) in the following language levels. The original implementation found in Indlog avoids regenerating the clauses by keeping a list of clauses that violate the language level. When the search ends in a language level, Indlog expands the clauses kept in the list instead of recomputing all clauses again.

4.5 Refinement Operator

The refinement operator in April is designed with the primarily concern of maintaining the relationship $\square \prec H \prec \perp$ for each clause H. Secondly, the operator avoids the use of siblings lists (for example, as used by Indlog and Aleph). This is achieved by exploring the ordering of the bottom clause literals.

April's refinement operator takes as input a clause (represented as a list of bottom clause literal identifiers), and all variables found in the clause (bound, unbound, and the clause's unbound head variables). The output is a literal that can be added to the clause. Each literal selected must be mode and language level conform, and its identifier bigger than the latest added identifier in the clause. By imposing an ordering on the clauses' literals, based on the order by which the literals were added to the bottom clause, the operator eliminates combinations of literals that would lead to equivalent clauses, although syntactically different.

Example 11 presents some of the clauses generated, by applying repeatedly the refinement operator to the head literal (0) of the bottom clause in Example 10.

Example 11
Clauses that the refinement operator may generate:

```
member(A,B):-A=3
member(A,B):-A=3,B=[2,3]
member(A,B):-B=[2,3]
member(A,B):-B=[C|D]
member(A,B):-B=[C|D],member(A,D)
member(A,B):-B=[C|D],member(A,D),member(C,B)
member(A,B):-B=[C|D],member(C,B)
member(A,B):-B=[C|D],D=[A|E]
```

The ordering imposed on the literals identifiers prevents the operator from generating redundant clauses like:

```
member(A,B):-B=[2,3],A=3
member(A,B):-B=[C|D],B=[2,3]
member(A,B):-B=[C|D],member(C,B),member(A,D)
```

4.6 Coverage computation

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^-.

An efficient coverage computation is crucial for the performance of an ILP system. Several approaches have been proposed to improve coverage computation efficiency. Camacho [7, 43] proposed a technique called *lazy evaluation* that consists in avoiding or postponing the evaluation of each clause against all the examples. Lazy evaluation can be activated on April through the parameter ***lazy_eval*** and it is furher described in the next section. Another approach consists in performing exact transformations in the clauses generated to make them more efficient to be executed by a Prolog engine [10, 9]. This technique is available on April and can be activated using the parameter ***optimise_clauses***. Another technique used to speedup coverage computation consists in maintaining a cache with covered examples of each clause. Coverage caching is described in Section 4.7. A technique called *query pack* [6] exploits the existent great number of shared literals between a clause and its refinements. It groups the clauses in sets that are executed as a pack (single clause). This technique aims at avoiding redundant computation in the coverage computation. One of those approaches consists in performing the coverage test for each example in parallel [17, 14].

4.6.1 Lazy evaluation

The lazy evaluation of examples, originally proposed by Camacho [7], is a technique that avoids or postpones the evaluation of each clause against all the examples. We distinguish three types of lazy coverage computation: negatives; positives; and all. Algorithm 4.6.1 describes how the computation is performed in these cases. It is important to note that performing lazy evaluation limits the use of heuristics and pruning because the coverage computed for a given clause may be inaccurate.

Algorithm 3 Lazy Evaluation algorithm

Input : h /* An hypothesis (clause) */
B /* Background knowledge */
H /* Current theory */
C /* Set of constraints */
E^+_{cur} *and* E^- /* Examples */
$LazyType$ /* Type of lazy evaluation: pos, neg, all */
Output:(Pos, Neg) /* Number of positive and negative examples covered by h*/
$Neg = 0$
$Pos = 0$
$NOISE = C(noise)$
if $LazyType == neg$ **then**
 $Pos = compute\_coverage(h, B, H, E^+_{cur})$
 if $Pos < C(mincover)$ **then** $Neg = inf$
 else $Neg = compute\_lazy\_coverage(h, B, H, E^-, NOISE + 1)$
 endif
elseif $LazyType == pos$ **then**
 $Neg = compute\_lazy_coverage(h, B, H, E^-, NOISE + 1)$
 if $Neg > NOISE$ **then** $Pos = inf$
 else $Pos = compute\_coverage(h, B, H, E^+_{cur})$
 endif
elseif $LazyType == all$ **then**
 $MINCOVER = C(mincover)$
 $Pos = compute\_lazy\_coverage(h, B, H, E^+_{cur}, MINCOVER)$
 if $POS < MINCOVER$ **then** $Neg = inf$
 else $Neg = compute\_lazy\_coverage(h, B, H, E^-, NOISE)$
 if $Neg > NOISE$ **then** $Pos = inf, Neg = inf$
 else $Pos = compute\_coverage(h, B, H, E^+)$
 endif
endif

$compute\_lazy\_coverage(h, B, H, E, Limit)$ auxiliary function
$Covered = 0$
for each e **in** E **do**
 if $h \wedge B \wedge H \vdash e$ **then**
  $Covered = Covered + 1$
  if $Covered == Limit$ **then**
   break
  endif
 endif
end for
return Covered

4.7 Coverage Caching

As said earlier, an efficient coverage computation is crucial for the performance of an ILP system. In order to reduce the number of examples tested, thus minimizing the coverage computation time, April has a coverage caching mechanism that stores clause's coverages.

To reduce the time spent on computing clauses coverage some ILP systems, such as Aleph [20], Indlog [7], and April, maintain lists of examples covered (coverage lists) for each hypothesis that is generated during execution. Coverage lists are used in these systems as follows. An hypothesis S is generated by applying a refinement operator to another hypothesis G. Let $Cover(G) = \{all\ e \in E\ such\ that\ B \wedge G \models e\}$, 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 [8] 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 reduce the effort in coverage computation at the cost of significantly increasing memory consumption. Efficient data structures should be used to represent coverage lists to minimize memory consumption.

The data structure used to maintain coverage lists in systems like Indlog or Aleph 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 represented as a list of intervals corresponds to $[1-2, 5-10]$. Using a list of intervals to represent coverage lists is an improvement to lists of numbers but it still presents some problems. First, the efficiency of performing basic operations on the interval list is linear on the number of intervals and can be improved. Secondly, the representation of lists in Prolog is not very efficient regarding memory usage. The RL-Tree data structure was designed to tackle those problems mentioned: memory usage and execution time. It can be used to efficiently represent and manipulate coverages lists.

Even using lists of intervals the memory consumption is high and complexity of the operations, to access elements in the lists, is linear, i.e. $O(n)$, where n is the number of intervals.

4.7.1 RL-Trees

To tackle the memory consumption problem that results from storing clauses covered examples we designed and implemented a new data structure - **R**ange**L**ist-tree (RL-Tree).

The **R**ange**L**ist-Tree data structure is an adaptation of a generic data structure called quadtree [44] that has been used in areas like image processing, computer graphics, and geographic information systems. Quadtree is a term used to represent a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. Quadtree based data structures are differentiated by the type of data that they represent, the principle guiding the decomposition process, and the number of times the space is decomposed.

The RL-Tree is designed to store integer intervals (e.g. $[1-3] \cup [10-200]$). The goals in the design of the RL-Tree data structure were: efficient data storage; fast insertion and removal; and fast retrieval.

In the design and implementation of the RL-Tree data structure we took the following characteristics into consideration: intervals are disjuncts; intervals are defined by adding or removing numbers; and, the domain (an integer interval) is known at creation time.

RL-Trees are trees with two distinct type of nodes: list and range nodes. A list node represents a fixed interval, of size LI, that is implementation dependent. A range node corresponds to an interval that is subdivided in B subintervals. Each subinterval in a range node can be completely contained (represented in Black) or partially contained in an interval (represented in Gray), or not be within an interval (represented in White).

The basic idea behind RL-Trees is to represent disjunct set of intervals in a domain by recursively partition the domain interval into equal subintervals. The number of subintervals B generated in each partition is implementation dependent. The number of partitions performed depend on B, the size of the domain, and the size of list node interval LI. Since we are using RL-Trees to represent coverage lists, the domain is $[1, NE]$ where NE is the number of positive or negative examples. The RL-Tree whose domain corresponds to the integer interval $[1, N]$ is denoted as RL-Tree(N).

A RL-Tree(N) has the following properties: $LN = ceil(N/LI)$ is the maximum number of list nodes in the tree; $H = ceil(log_B(LN))$ is the maximum height of the tree; all list nodes are at depth H; root node interval range is $RI = B^H * LI$; all range node interval bounds (except the root node) are inferred from its parent node; every range node is colored with black, white, or gray; only the root node can be completely black or white.

The RL-Tree data structure was implemented in C as a shared library. Since the ILP system used in the experiments is implemented in Prolog we developed an interface to RL-Tree as an external Prolog module.

Like other quadtrees data structures [45], a RL-Tree can be implemented with or without pointers. We chose to do a pointerless implementation (using an array) to reduce memory consumption. The LI and B parameters were set to 16 and 4 respectively. The range node is implemented using 16 bits. Since we divide the intervals by a factor of 4, each range node may have 4 subintervals. Each subinterval has a color associated (White, Black, and Gray) that is coded using 2 bits (thus a total of 8 bits are used for the 4 subintervals). The other 8 bits are used to store the number of subnodes of a node. This information is used to improve efficiency by reducing the traversal of the tree to determine the position, in the array, of a given node. The list node uses 16 bits. Each bit represents a number, determined by its position in the tree. The number interval represented by a list node is inferred from its parent range node.

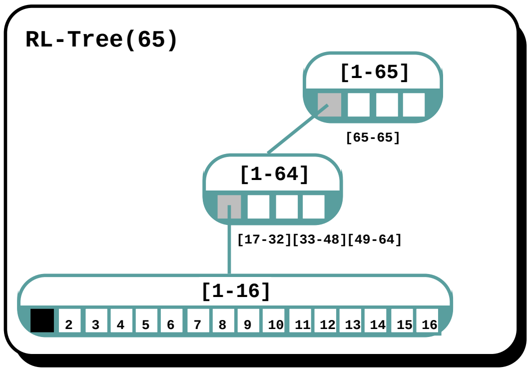

Figure 2: Interval [1] represented in a RL-Tree(65)

Consider the RL-Tree with domain [1,65], also denoted as RL-Tree(65). The Figures 2, 3, and 4 show some intervals represented in a RL-Tree(65). Each group of four squares in Figure 2 represents a range node. Each square in a range node corresponds to a subinterval. The sixteen square group represents a list node. Each square in a list node corresponds to an integer. The top of the tree contains a range node that is associated to the domain ($[1, 65]$). Using the properties of RL-Trees described earlier one knows that the maximum height of the RL-Tree(65) is 2 and the root node range is $[1 - 256]$. Each subinterval (square) of the root interval represents an interval of 64 integers. The first square (counting from the left) with range $[1 - 64]$ contains the interval $[1]$, so it is marked with Gray. The range node corresponding to the range $[1 - 64]$ has all squares painted in White except the first one corresponding to range $[1 - 16]$, because it contains the interval [1]. The list node only has one square marked, the square corresponding to the integer 1. Figure 3 shows the representation of a more complex set of intervals. Note that the number of nodes is the same as in Figure 2 even though it represents a more complex set of intervals.

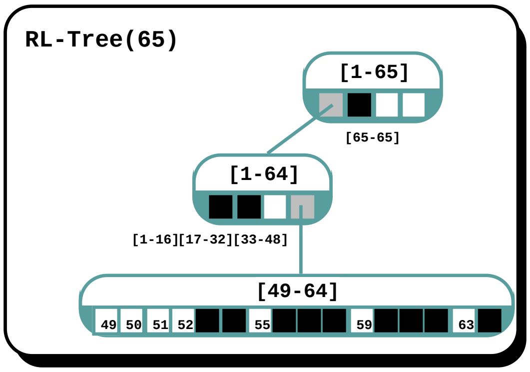

Figure 3: Intervals $[1,32] \cup [53,54] \cup [56,58] \cup [60,62] \cup [64,65]$ represented in a RL-Tree(65)

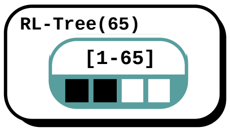

Figure 4: Interval [1,65] represented in a RL-Tree(65)

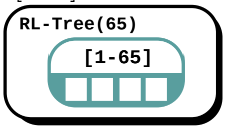

Figure 5: Interval $\varnothing$ represented in a RL-Tree(65)

The RL-Tree(N) operations implemented and their complexity (regarding the number of subintervals considered) are:

- Create a RL-Tree: $O(1)$;
- Delete a RL-Tree: $O(1)$;
- Check if a number is in a RL-Tree: $O(H)$.
- Add a number to a RL-Tree: $O(H)$
- Remove a number from a RL-Tree: $O(H)$

The current implementation of RL-Trees uses, in the worst case, $(4^{H+1}-1)/3$ nodes. The worst case occurs when the tree requires all LN list nodes. Since each node in the tree requires 2 bytes, a RL-Tree(N) will require, in the worst case, approximately $((4^{H+1}-1)/3) * 2 + C$ bytes, where C is the memory needed to store tree header information. In our implementation $C = 20$.

Preliminary results of using RL-trees in April are shown in Section 5.4.

5 Experiments and Results

The goal of the experiments were to empirically evaluate the effects on execution time and memory consumption of some techniques that are available in April. In particular, we evaluate coverage caching, RL-Trees, and Tries.

5.1 Experiment Settings

The experiments were made 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 4.3.23 of the YAP Prolog.

The datasets used were downloaded from the Machine Learning repositories of the Universities of Oxford[3] and York[4]. Table 3 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 [46]. The *sample* defines the number of examples used to induce a clause. *Lang* parameter specifies the maximum number of occurrences of a predicate symbol in an hypothesis [7]. *MinPos* specifies the minimum number of positive examples that an hypothesis must cover in order to be accepted. Finally, the parameter *noise* defines the maximum number of negative examples that an hypothesis may cover in order to be accepted.

Dataset	**Characterization**			**April's Settings**					
	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$	**nodes**	**i**	**sample**	**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	all	-	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
train	5	5	10	no limit	2	all	1	1	0
train128	120	5	10	no limit	2	all	1	1	0

Table 3: Settings used in the experiments

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 needed to process a dataset. However, since we are comparing the memory consumption and execution time when using a feature against not using it, the estimate obtained gives a good idea of the impact of feature.

The theory accuracies obtained by April with the given settings for the several datasets are presented in Table 13 in the Appendix B.

5.2 Coverage Caching

The impact of activating coverage caching is presented in Table 4. The table 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

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

caching and not using coverage caching). The memory values presented correspond only to the total memory used by April. The coverage lists were represented as RL-Trees.

Dataset	$\mid H \mid$	Time (sec.)		Memory (bytes)		on/off(%)	
		off	on	off	on	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**
train	37	0.02	0.01	1743048	1757620	**50.00**	**100.83**
train128	44	0.03	0.05	1806416	1834544	**166.66**	**101.55**

Table 4: The impact of coverage caching

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, in some cases more than 5 times, for larger datasets (i.e. datasets with larger number of examples and $\mid H \mid$). 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.

The overall poor results of coverage caching do not reflect the relevance of cache as can be seen in Table 5. It shows, for each dataset, the number of entries (clauses and their coverage) in the cache, the number of hits and misses [5], and the number of a clause "father"hit (i.e, the number of times that a father clause coverage was found in cache). Finally, the last column presents the ratio of the number of entries with $\mid H \mid$, given an idea of the redundancy in the search space. The values show that there is a lot of redundancy that is being exploited by the cache (as the high number of cache hits indicate). Hence, we conclude that the poor coverage caching results should be a consequence of implementation problems.

In order to identify the causes of the poor caching results we decided to activate YAP's profiling and then rerun April for all datasets previously considered. One first issue that we would like to clarify is whether coverage caching reduced the number of goal invocations executed. Table 6 shows

[5] The number of hits and misses correspond to the number of times that a positive, or negative, coverage of a clause was found in the cache. Thus, if the positive and negative coverage of a clause is found in the cache then it would result in two cache hits.

Dataset	Entries	Hits	Misses	Father Hits	Entries/$\mid H \mid$(%)
amine uptake	7808	90465	11608	11523	11.66
carcinogenesis	6127	242943	9090	8846	4.29
choline	24989	1437532	38424	38289	3.11
krki	123	3098	245	91	4.76
mesh	17091	374344	24144	24046	6.02
multiplication	157	566	294	246	32.84
pyrimidines	18918	639304	29015	28942	5.08
proteins	477	864848	741	530	0.11
train	16	15	26	23	43.24
train128	31	27	62	58	70.45

Table 5: Coverage caching statistics

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.

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 6: Total number of calls and execution time

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 7 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 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. 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 7: Calls distribution among April's modules

The results in Table 7 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 millions operations made by the cache module appear to be more expensive than the 30 billion operations whose execution were avoided by the use of coverage caching.

We further analyzed the profiling logs trying to identify the predicates that were causing the inefficiency problems. Table 8 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 had increased, or operate the Prolog database.

Table 8 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.

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 8: 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.

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 YAP Prolog internal database. Moreover, we find that it would be very much useful the support of an efficient indexing mechanism using multiple keys.

5.3 Tries

When activated in April, the Trie [47] data structure stores some information about each hypothesis generated. More specifically, it stores the hypothesis (Prolog clause), a list of variables in the clause, a list of unbound variables in the clause, and a list of free variables in the clause.

Table 9 shows the total number of hypotheses generated ($|H|$), the execution time, the memory usage and the impact in performance for execution time and memory usage (given as a ratio between the values obtained using and not using Tries). The memory values presented correspond only to the memory used to store information about the hypotheses.

The use of tries resulted in an average reduction of 20% in memory consumption with the datasets considered. The train dataset was the only exception as it shows a degradation of 25% in memory consumption. This may indicate that the Tries data structure is not adequate for datasets with very small hypotheses space. However, memory usage is not a concern for problems with small hypotheses space.

With Tries, the execution time slightly increased but the overhead is not significant. The **krki** and **train128** datasets are exceptions, nevertheless unimportant as the difference in execution time is just a fraction of a second.

In summary, the results suggest that the Trie data structure reduces memory consumption with a minimal execution time overhead.

Dataset	| H |	Time (sec.)		Memory (bytes)		on/off(%)	
		off	on	off	on	Time	Memory
amine uptake	66933	357.1	362.4	739316	553412	**101.48**	**74.85**
carcinogenesis	142714	506.19	517.76	869888	680212	**102.28**	**78.19**
choline	803366	13451.21	13573.24	869736	598344	**100.90**	**68.79**
krki	2579	1.11	1.30	62436	50000	**117.11**	**80.08**
mesh	283552	3241.62	3267.85	607584	506112	**100.80**	**83.29**
multiplication	478	8.91	8.98	164304	105348	**100.78**	**64.11**
pyrimidines	372320	5581.35	5602.96	914520	580852	**100.38**	**63.51**
proteins	433271	794.03	832.83	759440	595928	**104.88**	**78.46**
train	37	0.02	0.02	9260	11612	**100.00**	**125.39**
train128	44	0.05	0.06	22224	21392	**120.00**	**96.25**

Table 9: The impact of Tries

5.4 RL-Trees

Table 10 presents the impact of using RL-Trees in the April system. 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 RL-Trees and Prolog range lists). The memory values presented correspond only to the memory used to store coverage lists.

Dataset	| H |	Time (sec.)		Memory (bytes)		rl/list(%)	
		list	rl	list	rl	Time	Memory
amine uptake	66933	365.74	357.23	5142784	2181658	**97.67**	**42.42**
carcinogenesis	142714	508.41	505.61	2972668	1560180	**99.44**	**52.48**
choline	803366	13778.29	13617.49	17644032	7520744	**98.83**	**42.62**
krki	2579	1.22	1.13	150264	43822	**92.62**	**29.16**
mesh	283552	3394.1	3258.21	8286944	4880746	**95.99**	**58.89**
multiplication	478	8.89	8.91	35808	35412	**100.22**	**98.89**
pyrimidines	372320	5606.97	5460.22	24291608	6568286	**97.38**	**27.03**
proteins	433271	805.97	791.92	693868	146344	**98.25**	**21.09**
train	37	0.02	0.02	3676	3692	**100.00**	**100.43**
train128	44	0.05	0.05	10228	7284	**100.00**	**71.21**

Table 10: The impact of RL-Trees

The use of RL-Trees resulted in an average of 50% reduction in memory usage (when comparing to Prolog range lists). The only exception to the overall reduction was registered by the *train* dataset. This is probably a consequence of the reduced number of examples of the dataset. The results indicate that more significants reductions in memory usage are obtained with datasets with greater number of examples.

In general, a great reduction in memory usage is achieved with no execution time overhead when using RL-Trees. In fact, an average reduction of 2% in the execution time is obtained.

5.5 Tries and RL-Trees

To evaluate the impact of both data structures, we ran April again configured to use both data structures. The Table 11 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 the proposed data structures and not using them). The memory values presented correspond only to the memory used to store coverage lists and information about the hypotheses (stored in Tries).

Dataset	$\mid H \mid$	**Time (sec.)**		**Memory (bytes)**		**on/off(%)**	
		off	**on**	**off**	**on**	**Time**	**Memory**
amine uptake	66933	365.74	362.83	5882100	2728174	**99.20**	**46.38**
carcinogenesis	142714	508.41	516.51	3842556	2223164	**101.59**	**57.85**
choline	803366	13778.29	13651.51	18513768	8090504	**99.07**	**43.69**
krki	2579	1.22	1.21	212700	93978	**100.82**	**44.18**
mesh	283552	3394.1	3284.33	8894528	5376906	**96.76**	**60.45**
multiplication	478	8.91	8.98	200112	140908	**100.78**	**70.41**
pyrimidines	372320	5606.97	5501.65	25206128	7132978	**98.12**	**28.29**
proteins	433271	805.97	834.76	1453308	740904	**103.57**	**50.98**
train	37	0.02	0.02	12936	15264	**100.00**	**117.99**
train128	44	0.05	0.05	32452	28564	**100.00**	**88.01**

Table 11: The impact of Tries and RL-Trees

The use of both data structures resulted in significant reductions in memory usage. The *train* was the only dataset that consumed more memory by using Tries with RL-Trees. This occurred because the dataset has a very small hypothesis space and the number of examples is also small. Nevertheless, the values obtained are useful because they give an idea of the initial overhead of the data structures.

The results indicate that the impact of the data structures tend to be greater in the datasets with more examples and with bigger search spaces.

Dataset	**Memory (MB)**	**Reduction (%)**
amine uptake	11.02	21.64
carcinogenesis	13.14	9.04
choline	31.28	26.57
krki	2.21	4.02
mesh	24.86	23.83
multiplication	4.18	0.44
pyrimidines	26.53	42.15
proteins	26.22	2.01
train	1.68	-1.50
train128	1.75	-1.11

Table 12: April memory consumption using Tries and RL-Trees

Table 12 shows the impact of the data structures proposed in the April system total memory usage. The table shows the April (total) memory usage when using Tries and RL-Trees simultaneously and the reduction ratio when comparing to using Prolog range lists and not using Tries.

The reduction values obtained are good, especially if we take into account that the biggest reductions (42.15 and 26.57) were obtained in the datasets with greatest memory usage. From the reduction values presented we conclude that with small datasets the data structures do not produce major gains, but they also do not introduce significant overheads. On the other hand, the data structures proposed should be used when processing larger datasets since they can reduce memory consumption very significantly.

6 Related Work

Since the initial concept proposal of Inductive Logic Programming [1] many ILP systems have been developed. A compreensive list of ILP systems and their description is provided in [48]. April is specialy related to Indlog [7] and Aleph [20] systems. Like in April, the core algorithm used in these systems is based on Mode Direct Inverse Entailment (MDIE), a technique initially used in the Progol [18] system. April implements most of the features found in Aleph and Indlog. Due to this close relation, April maintains compatibility with the parameters and input files format of Indlog and Aleph.

The main differences between April and Indlog are in the strategy of the reduction algorithm and in bottom clause construction. Indlog uses an iterative deepening search or best-first while April can use a breadth-first or best-first search. When constructing the bottom clause, Indlog constructs a directed-acyclic graph of literals that reduces the search space by not considering some combinations of literals. The Indlog's Incremental Language Level Search [11] strategy can be emulated in April by defining the ***language_level*** parameter, however its implementation is still more inefficient than Indlog's. April differs from those two systems by implementing specific data structures in C, such as RL-trees and tries, in order to reduce memory consumption and improve execution time.

April traverses the generalisation lattice as MIS [28], FOIL [29] and Progol. Like GOLEM [27] and Progol, April generates an initial clause to bound the generalization lattice, thus reducing the search space. Unlike FOIL and GOLEM, April can handle non-ground background knowledge. In the line of Indlog, Skillit [49], CILS [21], and Aleph, April is implemented using the Prolog language and uses YAP [33] compiler.

7 Conclusions and future work

This report described the use and implementation of the April ILP system. Two novel data structures were introduced in April: Tries and RL-Trees. The Trie data structure inherently and efficiently exploit the similarities among the candidate hypotheses generated by ILP systems to reduce memory usage. The RL-Tree is a novel data structure designed to efficiently store and manipulate coverage lists. The data structures were integrated in the April system. Both data structures were implemented in C and are available to a Prolog engine as an external Prolog module.

We have empirically evaluated the use of several features on April, namely the use of coverage caching, RL-Trees, Tries, RL-Trees and Tries (simultaneously), on well known datasets. Coverage caching was evaluated and the results indicate that the technique when applied to the majority of the datasets results in a significant increase of memory usage and execution time. We advocate that the problem is a consequence of the poor performance of the Prolog internal database, and we suggest that Prolog implementors should try to improve it. The proposed Tries data structure reduced memory consumption with a minor overhead (approximately 1%) in the execution time. The RL-Tree data structure reduced memory usage in coverage lists to half, in average, and slightly reduced the execution time. The use of both data structures simultaneously resulted in a overall reduction in memory usage without degrading the execution time. In some datasets, the April system registered very substantial memory reductions (between $20-42\%$) by using both Tries and RL-Trees simultaneously. The results indicated that the benefits from using these data structures tend to increase for datasets with larger search spaces and larger number of examples. Since the data structures are system independent, we believe that they can be also applied to other ILP systems (even to other paradigms of machine learning) with positive impact.

There is still a lot to improve in April. In the near future we plan to improve the implementation of the language level extension, and continue the development of the RL-Trees by implementing operations like intersection or subtraction of two RL-Trees in order to compute the coverage intersection of two clauses more efficiently. We will also try to identify more data stored during the search that may take advantage of the Tries data structure.

The search spaces in ILP systems contain many redundancies. We plan to investigate the types of clause redundancy that ILP systems may have in their search space and try to point solutions to efficiently decrease the redundancy.

We also plan to parallelize April and complete the integration of an RDBMS interface (so learning can be done directly from a database). While the Prolog database is not improved, we will tackle the problem of implementing coverage caching efficiently by designing and implementing in C an efficient data structure.

Acknowledgments

The authors thank Ricardo Rocha for implementing the Tries module and Ins Dutra for her helpfull comments during the initial development of April. 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] Luc De Raedt. A perspective on inductive logic programming. In *The logic programming paradigm - a 25 year perspective*, pages 335,346. Springer-Verlag, 1999.

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

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

[6] 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.

[7] Rui Camacho. *Inducting models of human control skills unsing ML algorithms.* PhD thesis, Univerity of Porto, 2000.

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

[9] Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 2002.

[10] Vítor Santos Costa, Ashwin Srinivasan, and Rui Camacho. A note on two simple transformations for improving the efficiency of an ILP system. *Lecture Notes in Computer Science*, 1866, 2000.

[11] 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.

[12] 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.

[13] 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.

[14] Hayato Ohwada and Fumio Mizoguchi. Parallel execution for speeding up inductive logic programming systems. In *Lecture Notes in Artificial Intelligence*, number 1721, pages 277–286. Springer-Verlag, 1999.

[15] Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. 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 165–173. Springer-Verlag, 2000.

[16] Mahesh Joshi Eui-Hong. Parallel algorithms in data mining, 2000.

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

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

[19] 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.

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

[21] 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.

[22] 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.

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

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

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

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

[27] 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.

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

[29] 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.

[30] P.R.J. van der Laag. *An analysis of refinement operators in inductive logic programming.* PhD thesis, Erasmus Universiteit, Rotterdam, the Netherlands, 1995.

[31] Tom M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, New Brunswick, New Jersey, 1980.

[32] 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.

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

[34] Stephen Muggleton and John Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, September 2001.

[35] T. Davies and Stuart Russell. A logical approach to reasoning by analogy. In *IJCAI87*, pages 264–270, Los Altos, California, 1987.

[36] Nuno Fonseca, Vitor Santos Costa, and Ines de Castro Dutra. Visall: A universal tool to visualise parallel execution of logic programs. In *IJCSLP*, pages 100–114, 1998.

[37] 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.

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

[39] 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.

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

[41] 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.

[42] C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14(2):219–232, 1994.

[43] Rui Camacho. As lazy as it can be. In P. Doherty B. Tassen, P. Ala-Siuru and B. Mayoh, editors, *The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03)*, pages 47–58. Bergen, Norway, November 2003.

[44] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[45] Hanan Samet. Data structures for quadtree approximation and compression. *Communications of the ACM*, 28(9):973–993, 1985.

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

[47] Nuno Fonseca, Rui Camacho, Fernando Silva, and Vítor S. Costa. Induction with April: A preliminary report. Technical Report DCC-2003-02, DCC-FC, Universidade do Porto, 2003.

[48] Ilp systems. http://www.cs.bris.ac.uk/ ILPnet2/Systems/.

[49] A. Jorge and P. Brazdil. Architecture for iterative learning of recursive definitions. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 95–108. Department of Computer Science, Katholieke Universiteit Leuven, 1995.

A April output

The ouput of executing April with the input files described in Section 3 follows.

```
[ Initializing... ]
[ Initialization complete. ]

[ Files found in examples/member/ ]
[ Consulting examples/member/member.f. ]
[ Consulting examples/member/member.n. ]
[ Examples(+,-): (5,7) ]

*******************************************************************
* Settings:
verbose         = 2 ([0,1,2,3,4,5,6,7,8,9,10])
trace           = 0 ([0,1])
trace_y         = event_number ([event_number,time])
record          = false ([true,false])
recordfile      = record
example_storage = yapcdb ([yapcdb,rdbms])
rdbms_connection = conf(mysql,localhost,0,_3923,_3924,_3925)
pos_examples_table = pos_examples,[]
neg_examples_table = neg_examples,[]
use_tries       = no ([yes,no])
clean_tries     = yes ([yes,no])
h               = 10 (integer)
sat_example     = first ([first,random,weight])
pos_only        = false ([true,false])
selector        = 2 (integer)
typechecking    = no ([yes,no])
sampling        = 0 (integer)
i_determinancy  = 1 (integer)
backtracking_limit = 10 (integer)
construct_bottom = saturation ([saturation,reduction,false])
sat             = system ([user,system])
lazy_bottom     = false ([false,true])
low_level_refine = false ([false,true])
cache           = yes ([yes,no])
cache_storage   = rl ([list,rl])
refine          = auto ([auto,user])
noise           = 0 (integer)
pnoise          = 0
minpcover       = 0
mincover        = 0 (integer)
clause_length   = 8 (integer)
min_clause_length = 1 (integer)
nodes           = 2000 (integer)
search          = bf ([bestfirst,bf])
heuristic       = coverage ([pos,coverage,progol,l,acc,acc-ul,compression,compression2,weighted_coverage,laplace,coverage_l])
auto_settings   = disabled ([disabled,lazy,lazy_neg,lazy_pos])
bg_file         = examples/member/member.b
conf_file       =
unseen_neg_file = examples/member/
unseen_pos_file = examples/member/
train_neg_file  = examples/member/member.n
train_pos_file  = examples/member/member.f
weights_file    =
save_theory     = yes ([yes,no])
use_best_clause = yes ([yes,no])
greedy_use_best_clause = yes ([yes,no])
lazy_eval       = disabled ([disabled,pos,neg,all])
language        = 0 (integer)
language_init   = 1 (integer)
minacc          = 0.0
targetacc       = 1.0
explore         = false ([true,false])
max_theory_size = 0 (integer)
reduce_theory   = no ([yes,no])
optimise_clauses = no ([yes,no])
*******************************************************************
[ >Example Selection: Generating sample with 5 examples... ]
1 2 3 4 5
[ <Example Selection. ]
[ >Starting saturation... ]
[ Example picked (1):member(0,[0]) ]
[ Bottom Clause: ]
member(A,B) :-
   sat_eq(A,0),
   sat_eq(B,[0]),
```

```
   sat_eq(B,[A|C]),
   sat_eq(C,[]).

[ Number of Literals: 7]
[ Saturation Time: 0.0 seconds ]
[ <Saturation complete ]


[ >Starting reduction/search... ]
[ member(A,B):-A=0 ]
[ member(A,B):-B=[0] ]
[ member(A,B):-B=[A|C] ]
[ Best clause so far: member(A,B):-B=[A|C] = (s(2,0,2,1,1,0)) ]



[ Clause found:member(A,B):-B=[A|C] ]
[ Clauses: generated= 4 ; Evaluated=3 ; System Pruned=1 ; User Pruned=0; User Constrained: 0 ]
[ <Reduction/search complete. ]

[ >Starting saturation... ]
[ Example picked (3):member(3,[2,3]) ]
[ Bottom Clause: ]
member(A,B) :-
   sat_eq(A,3),
   sat_eq(B,[2,3]),
   sat_eq(B,[C|D]),
   sat_eq(D,[A|E]).

[ Number of Literals: 7]
[ Saturation Time: 0.0 seconds ]
[ <Saturation complete ]


[ >Starting reduction/search... ]
[ member(A,B):-A=3 ]
[ member(A,B):-B=[2,3] ]
[ member(A,B):-B=[C|D] ]



[ Clause found:member(A,B):-B=[A|C] ]
[ Clauses: generated= 4 ; Evaluated=3 ; System Pruned=0 ; User Pruned=0; User Constrained: 0 ]
[ <Reduction/search complete. ]

[ >Starting saturation... ]
[ Example picked (4):member(3,[4,2,3]) ]
[ Bottom Clause: ]
member(A,B) :-
   sat_eq(A,3),
   sat_eq(B,[4,2,3]),
   sat_eq(B,[C|D]),
   sat_eq(D,[E|F]),
   member(A,D).

[ Number of Literals: 9]
[ Saturation Time: 0.0 seconds ]
[ <Saturation complete ]


[ >Starting reduction/search... ]
[ member(A,B):-A=3 ]
[ member(A,B):-B=[4,2,3] ]
[ member(A,B):-B=[C|D] ]



[ Clause found:member(A,B):-B=[A|C] ]
[ Clauses: generated= 4 ; Evaluated=3 ; System Pruned=0 ; User Pruned=0; User Constrained: 0 ]
[ <Reduction/search complete. ]

[ >Starting saturation... ]
[ Example picked (5):member(5,[4,2,3,5]) ]
[ Bottom Clause: ]
member(A,B) :-
   sat_eq(A,5),
   sat_eq(B,[4,2,3,5]),
   sat_eq(B,[C|D]),
   sat_eq(D,[E|F]).
```

```
[ Number of Literals: 7]
[ Saturation Time: 0.00999999999999995 seconds ]
[ <Saturation complete ]


[ >Starting reduction/search... ]
[ member(A,B):-A=5 ]
[ member(A,B):-B=[4,2,3,5] ]
[ member(A,B):-B=[C|D] ]
[ member(A,B):-B=[C|D],D=[E|F] ]



[ Clause found:member(A,B):-B=[A|C] ]
[ Clauses: generated= 5 ; Evaluated=4 ; System Pruned=0 ; User Pruned=0; User Constrained: 0 ]
[ <Reduction/search complete. ]

[ Best clause: member(_3963,_3964):-_3964=[_3963|_3969] ]
[ (+,-,v,fv,hfv,time)=(s(2,0,2,1,1,0)) ]
[ >Cover Removal ]
[ Marked 2 examples: member(_3963,_3964):-_3964=[_3963|_3969] ]
[ <done. ]
[ Current theory accuracy: 0.75/1.0 PosCovered: 2/5 NegCovered: 0/7  ]
[ >Example Selection: Generating sample with 3 examples... ]
3 4 5
[ <Example Selection. ]
[ >Starting saturation... ]
[ Example picked (3):member(3,[2,3]) ]
[ Bottom Clause: ]
member(A,B) :-
   sat_eq(A,3),
   sat_eq(B,[2,3]),
   sat_eq(B,[C|D]),
   sat_eq(D,[A|E]),
   member(A,D),
   sat_eq(F,[A|G]),
   sat_eq(G,[E|E]),
   member(A,F),
   member(C,B).

[ Number of Literals: 14]
[ Saturation Time: 0.0 seconds ]
[ <Saturation complete ]


[ >Starting reduction/search... ]
[ member(A,B):-A=3 ]
[ member(A,B):-B=[2,3] ]
[ member(A,B):-B=[C|D] ]
[ member(A,B):-B=[C|D],member(A,D) ]
[ Best clause so far: member(A,B):-B=[C|D],member(A,D) = (s(3,0,3,2,1,0)) ]
[ member(A,B):-B=[C|D],member(C,B) ]
[ member(A,B):-B=[C|D],member(C,B) ]
[ member(A,B):-B=[C|D],D=[A|E] ]
[ member(A,B):-B=[C|D],member(A,D) ]



[ Clause found:member(A,B):-B=[C|D],member(A,D) ]
[ Clauses: generated= 9 ; Evaluated=8 ; System Pruned=0 ; User Pruned=0; User Constrained: 0 ]
[ <Reduction/search complete. ]

[ Best clause: member(_3965,_3966):-_3966=[_3973|_3974],member(_3965,_3974) ]
[ (+,-,v,fv,hfv,time)=(s(3,0,3,2,1,0)) ]
[ >Cover Removal ]
[ Marked 3 examples: member(_3965,_3966):-_3966=[_3973|_3974],member(_3965,_3974) ]
[ <done. ]




[ Theory found... ]
[ Predicate:member/2 ]
[ 2 clause(s) ]
      Clause
      (PosCovered,NegCovered,Value,ClauseLength,NFreeVars,HeadFreeVars)

[1]       member(A,B):-B=[A|C]
          s(2,0,2,1,1,0)
```

```
[2]        member(A,B):-B=[C|D],member(A,D)
           s(3,0,3,2,1,0)


Training set performance

Pred\Actual    |       +       |       -       |
       +       |       5       |       0       | 5
       -       |       0       |       7       | 7
               |       5       |       7       | 12
Accuracy = 100.0

[ Total Time: 0.03 seconds ]
[ Clauses: generated= 21 ; Evaluated=22 ; System Pruned=1 ; User Pruned=0; User Constrained: 0 ]
```

B April Accuracy

Dataset	Theory Size	Train Accuracy(%)	Test Accuracy(%)	Time (sec.)
amine uptake	3	58.37	-	58.37
carcinogenesis	3	64.76	61.53	506.65
choline	3	67.72	-	1840.25
krki	4	99.6	-	1.15
mesh	18	77.43	-	637.34
multiplication	2	100.0	100.0	10.54
pyrimidines	3	72.134	65.789	915.95
proteins	3	52.295	50.962	794.4
train	1	100.0	-	0.02
train128	1	100.0	-	0.05

Table 13: April's execution time and accuracy