SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Improving the Efficiency of Inductive Logic Programming Systems

Nuno A. Fonseca[1,*], Vítor Santos Costa[2,†], Ricardo Rocha[2,†], Rui Camacho[3,‡,*] and Fernando Silva[2,†]

[1] *Instituto de Biologia Molecular e Celular (IBMC), Universidade do Porto, Rua do Campo Alegre, 823, 4169-007 Porto, Portugal*
[2] *CRACS & Faculdade de Ciências, Universidade do Porto Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal*
[3] *LIAAD & Faculdade de Engenharia, Universidade do Porto Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal*

SUMMARY

Inductive Logic Programming (ILP) is a sub-field of Machine Learning that provides an excellent framework for Multi-Relational Data Mining applications. The advantages of ILP have been successfully demonstrated in complex and relevant industrial and scientific problems. However, to produce valuable models, ILP systems often require long running times and large amounts of memory. In this article we address fundamental issues that have direct impact on the efficiency of ILP systems. Namely, we discuss how improvements in the indexing mechanisms of an underlying Logic Programming system benefit ILP performance. Furthermore, we propose novel data structures to reduce memory requirements and we suggest a new lazy evaluation technique to search the hypothesis space more efficiently. These proposals have been implemented in the April ILP system and evaluated using several well known data sets. The results observed show significant improvements in running time without compromising the accuracy of the models generated. Indeed, the combined techniques achieve several order of magnitudes speedup in some data sets. Moreover, memory requirements are reduced in nearly half of the data sets.

KEY WORDS: Inductive Logic Programming, Efficiency, Data Structures, Indexing

*E-mail: nf@ibmc.up.pt
†E-mail: {vsc, ricroc, fds}@dcc.fc.up.pt
‡E-mail: rcamacho@fe.up.pt
*Correspondence to: Faculdade de Engenharia, Universidade do Porto
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal

Received 03-Jan-2008
Revised 02-May-2008
Accepted

1. Introduction

One of the main areas of research in the Machine Learning and Data Mining communities is learning from structured data and relational databases. Arguably, most developments in this area originate from Inductive Logic Programming (ILP) [13], a sub-field of Machine Learning where one learns (induces) first-order logical theories from *examples* and, additionally, from background knowledge. ILP provides an excellent, well-studied framework for learning in multi-relational domains. Moreover, the theories learned by general purpose ILP systems are in a high-level formalism often understandable and meaningful for the domain experts. ILP systems have therefore been successful in both industrially and scientifically relevant problems [51, 12, 27, 50] (see [1, 13, 20] for a detailed survey on ILP applications).

Most often, ILP systems learn by searching the best hypothesis over a very large space of hypotheses. As the search space tends to grow rather quickly, ILP systems can take several hours, if not days, to return a theory for complex applications. Improving efficiency of ILP systems has thus been recognized as one of the main issues to be addressed by the ILP community [33]. Initial research focused on reducing the search space [28, 7], and on efficiently testing candidate hypotheses [4, 44]. Although such strategies have been largely successful, ILP algorithms are sufficiently complex to offer a large scope for research on further improvements. Opportunities range from improving the ILP algorithm down to low-level engine optimizations. Arguably, improving the ILP algorithms may offer the most opportunities, but on the other hand doing so may affect the learning quality. In contrast, it was not always clear whether low-level optimization of an inference engine can have a substantial impact.

In this paper we study the performance of a number of optimizations that operate at different levels in the search space: **(i)** we relax the coverage mechanism, which is used by ILP to quantify the goodness of clauses; **(ii)** we study the performance of specialized storage algorithms that fit ILP search; and, **(iii)** we study whether performance of the inference engine can be improved. Next, we motivate each optimization.

Most ILP systems follow a generate-and-test approach. Hypotheses (or clauses) are generated, usually by extending a previous clause, and then evaluated by computing how many examples they cover, or *coverage*. Often, generating a new clause is straightforward. On the other hand, the evaluation of individual hypothesis (theorem proving effort) can be hard [5], or it can just become expensive if the number of examples is large, say in Data Mining applications. Techniques such as query transformations [43, 44] and query-packs [4] have been designed to speed-up query answering. Even though they significantly improve performance, further gains can be obtained by estimating or predicting clause coverage [47, 48]. Unfortunately, such efforts may result in incorrect measures of an hypothesis' value. We propose a new technique, *lazy evaluation of examples*, such that we *only evaluate the examples we have to*.

ILP systems search large spaces of hypotheses. In larger applications this space can quickly grow to thousands and even millions of different clauses. Efficiently storing and searching in this space can become a major overhead in ILP systems.

Our first concern is storing the search space itself. We start from the observation that the ILP search space is highly redundant [29]: the same clause will be proposed over and over again. Moreover, quite often clauses are very similar. This is partly because several clauses

may result from expanding a common ancestor, and will therefore share a common prefix. The query-packs technique first exploited this similarity to reduce computation [4] by compiling the search-space *before* computing coverage. On the other hand, query-packs were designed for a specific ILP approach. In contrast, our proposal is geared towards obtaining a compact representation of the search space. Our goal is to improve memory usage, and to make it efficient to verify whether some clause has already been explored. Note that to do so we benefit from previous work in using tries to support Logic Programming [38]. Our proposal is in fact closer to the work of Nijssen and Kok, where tries are used to represent item-sets in a multi-relational setting [30], with significant performance improvements. However, the tries proposed in this paper are also more fine-grained (they work at sub-term level) than those of Nijssen and Kok, and are used to compactly store more information than just clauses.

ILP systems often keep an *open list* of clauses that deserve to be refined, or further explored. For each such clause it is quite convenient to keep a list of examples covered, the *coverage list*. This helps when we compute coverage for the new clauses. Experience with ILP systems such as Aleph [52], IndLog [6], and April [15] suggests that these lists are a major source of space overhead. As coverage lists represent sets of examples they do not have the nice incremental property of clauses. On the other hand, these sets can become quite sparse as clauses cover less examples. A similar problem is addressed by the quad-tree representation which has been so successful at compactly representing sparse spaces. In this vein, we propose the *RL-tree data structure*.

Ideally, ILP systems should spend most of their running time performing inference. To do so, they often use the inference mechanism of a Prolog engine to do theorem proving. As it turns out, it often becomes easier to implement the whole ILP system in Prolog. As ILP systems address larger and larger applications, they challenge traditional Prolog engines. We show that *indexing* is critical for good ILP performance [45]. Aggressive indexing is both required for static data structures, representing the databases, and for the dynamic data structures that represent the search space.

The proposed techniques have been implemented in the April [15][†] ILP system, which runs on top of the Yap Prolog engine [42]. We evaluated the different techniques through a large set of ILP applications, that we believe represent the main application domains for ILP. The results obtained show that our techniques achieve both significant speedups and memory usage improvements. Most important, the results preserve the accuracy and quality of the models found. The exception is that the quality of the model may be affected when adopting lazy evaluation, as we lose information that could be used to help in navigating the search space.

The remainder of this article is organized as follows. Section 2 provides some ILP background, including a description of an ILP procedure. Section 3 presents and discusses the lazy evaluation technique. Next, in Sections 4 and 5, we present, respectively, the trie, and RL-tree data structures. In Section 6 we discuss how improvements in the indexing algorithm of a Prolog engine may benefit Prolog-based ILP systems. Section 7 presents and discusses the experimental results. We draw some conclusions in Section 8 and point out future work.

[†]APRIL is currently available for academic purposes upon request to Nuno Fonseca (nf@ibmc.up.pt).

2. Background

This section briefly presents some concepts and terminology of Inductive Logic Programming. For a gentle introduction to the field, please refer to [25, 29].

2.1. The ILP Problem

Given a set of positive and negative examples of the concept to learn, and some prior knowledge, or *background knowledge*, the fundamental goal of an ILP system is to induce a logic program that entails all positive examples and none of the negative ones.

More formally, let E^+ be the set of positive examples, E^- the set of negative examples, $E = E^+ \cup E^-$, and B the background knowledge. In general, B and E can be arbitrary logic programs and have to satisfy:

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

The aim of an ILP system is to find a set of hypotheses (also referred to as a theory) H, in the form of a logic program, such that the following conditions hold:

- **Posterior Satisfiability**: $B \wedge E^- \wedge H \not\models \square$
- **Posterior Sufficiency**: $B \wedge H \models E^+$
- **Posterior Necessity**: $B \wedge h_i \models e_1^+ \vee e_2^+ \vee \ldots \vee e_n^+ \quad (\forall h_i \in H, e_j \in E^+)$

The posterior sufficiency condition is known as *completeness* with regard to positive evidence, and the posterior satisfiability condition is sometimes referred as *consistency* with the negative evidence. The posterior *necessity* avoids the induction of clauses that do not entail any of the positive examples.

In short, the problem that an ILP system must solve is to find a consistent and complete theory, i.e., find a set of hypotheses that *explains* all the given positive examples, while being consistent with the given negative examples. Since it is not usually obvious which set of hypotheses should be retrieved as the theory, an ILP system must search through the permitted hypotheses to find a set with the desired properties.

The states in the search space, the *hypothesis space*, are concept descriptions, or hypotheses, and the goal is to find one or more states satisfying some quality criterion. To evaluate the quality of the hypotheses generated during the search, there are several evaluation measures that can be applied [21, 18]. The two most widely used measures are *accuracy* and *coverage*. Accuracy is the percentage of examples correctly classified by an hypothesis. Coverage of an hypothesis h is the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from $B \wedge h$. The time needed to compute the coverage of an hypothesis depends primarily on the cardinality of E (i.e., E^+ and E^-) and on the theorem proving effort required to evaluate the background knowledge.

induce(B,C,E)
Input: Background knowledge B, hypotheses constraints C, and a finite training set $E = E^+ \cup E^-$.
Output: A set of complete and consistent hypotheses H.
1. $i = 0$
2. $H_i = \varnothing$
3. **if** $E^+ = \varnothing$ **then return** H_i
4. $i = i + 1$
5. $Train = E^+ \cup E^-$
6. $e_i^+ = select\ an\ example\ from\ E^+$
7. $\bot_i = saturate(B, C, H_{i-1}, e_i^+)$
8. $h_i = search(B, C, Train, H_{i-1}, e_i^+, \bot_i)$
9. $H_i = H_{i-1} \cup h_i$
10. $E_{covered} = \{e \mid e \in E^+ \wedge B \cup H_i \vDash e\}$
11. $E^+ = E^+ \setminus E_{covered}$
12. **goto** *step* 3

Figure 1. A general induction procedure based on MDIE.

2.2. The Mode-Directed Inverse Entailment approach to ILP

From the point of view of an ILP system, induction can be seen as an iterative process of repeatedly applying an algorithm to traverse the hypothesis space searching for a satisfactory clause. Next, we discuss a widely implemented algorithm, *Mode-Directed Inverse Entailment* (MDIE), a procedure that uses inverse entailment together with mode restrictions to find new clauses [24]. Figure 1 shows a general implementation of an induction procedure based on MDIE.

Starting from a set of examples E, a background knowledge B, and some constraints C, the `induce()` procedure follows a greedy cover set approach to induce a theory H. The most interesting steps in the algorithm are *saturation* (step 7), *reduction* (step 8) and *cover removal* (steps 10 and 11). Saturation constructs the most specific clause $\bot_i$ (B, C, H_{i-1}, e_i^+) a definite clause that is used to constrain the search space. The `search()` procedure does most of the work. MDIE systems work top-down, by adding literals from the bottom-clause to an existing clause. Bottom-clauses can grow to thousands of literals: one of the constraints in C imposes a limit on the number of hypotheses generated, thus ensuring that `search()` terminates. If no hypothesis is found by `search()` then example e_i is returned. The cycle then continues by *cover removal*: it removes the examples covered by the new rule and searches for a clause that covers the remaining examples.

2.3. Coverage Lists and Coverage Caching

Coverage computation is performed for each generated hypothesis during the search procedure (step 8 in Figure 1). To reduce the time spent on computing hypotheses' coverage, ILP systems such as Aleph [52], FORTE [37], IndLog [6], and April [15], 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 h_i is generated by applying a refinement operator to another hypothesis h_j. Let $Cover(h_j) = \{all\ e \in E\ such\ that\ B \wedge h_j \vDash e\}$, where B is the background knowledge and E is the set examples. Since h_j is more general than h_i then $Cover(h_i) \subseteq Cover(h_j)$. Taking this into account, when testing the coverage of h_i it is only necessary to consider examples of $Cover(h_j)$, thus reducing the coverage computation time.

Cussens [9] extended this scheme by proposing a kind of *coverage caching*. The coverage lists are permanently stored and reused whenever necessary, thus avoiding the need to recompute the coverage of some hypotheses. Coverage caching reduces 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.

Another approach is taken in the FOIL [35] system. FOIL keeps all possible bindings for the variables appearing in the hypothesis, thus reducing the theorem proving effort in coverage computation when new literals are added to it. This approach is limited in two ways: first, it requires large amounts of memory, as in the case of coverage lists; second, it is restricted to algorithms that greedily explore the search space due to the amount of memory required.

3. Lazy Evaluation of Examples

A general ILP system spends most of its time generating and evaluating hypotheses. Generating hypotheses is computationally very cheap. A significant percentage of the overall time of an ILP system is spent in the evaluation of hypotheses, either because the number of examples is large or because testing each example is computationally hard [5]. *Lazy evaluation of examples* [8] is a technique that aims at speeding up the evaluation of hypotheses by avoiding the unnecessary use of examples in the coverage computations. It is usable in any general purpose ILP system and does not affect the completeness of the hypotheses search. The term *lazy evaluation* is used in the sense of making the minimal computation to obtain useful information.

We distinguish between lazy evaluation of positive examples, lazy evaluation of negative examples and total laziness. Figure 2 presents the complete lazy evaluation algorithm. We next describe in more detail each form of lazy evaluation.

3.1. Lazy Evaluation of Negatives

One can observe that, in any case, an hypothesis will only be accepted if it is consistent with the negative examples. We will designate a clause that is inconsistent with the negative examples (the hypothesis being too general) a *partial clause*. Lazy evaluation takes advantage of this observation.

lazy_evaluation(h,B,H,C,E^+_{cur},E^-,LazyType)
Input: Hypothesis h, Background knowledge B, current theory H, hypotheses constraints C, set of positive examples E^+_{cur}, set of negative examples E^-, and type of evaluation $LazyType$.
Output: The number of positive (Pos) and negative (Neg) examples covered by h.
$Neg = 0$
$Pos = 0$
$Noise = C(NOISE)$
$MinCover = C(MINCOVER)$
$MaxNegs =\mid E^- \mid$
$MaxPos =\mid E^+ \mid$
if $LazyType = NEGATIVES$ **then**
  $Pos = compute\_coverage(h, B, H, E^+_{cur})$
  if $Pos < MinCover$ **then** $Neg = MaxNegs$
  else $Neg = compute\_lazy\_coverage(h, B, H, E^-, Noise + 1)$
elseif $LazyType = POSITIVES$ **then**
  $Pos = compute\_lazy\_coverage(h, B, H, E^+_{cur}, MinCover)$
  if $Pos < MinCover$ **then** $Neg = MaxNegs$
  else
    $Neg = compute\_lazy\_coverage(h, B, H, E^-, Noise + 1)$
    if $Neg > Noise$ **then** $Pos = MaxPos, Neg = MaxNegs$
    else $Pos = compute\_coverage(h, B, H, E^+)$
elseif $LazyType = TOTAL$ **then**
  $Neg = compute\_lazy\_coverage(h, B, H, E^-, Noise + 1)$
  if $Neg > Noise$ **then** $Pos = MaxPos$
  else $Pos = compute\_coverage(h, B, H, E^+_{cur})$
return (Pos, Neg)

compute_lazy_coverage(h,B,H,E,Limit)
$Counter = 0$
for each e **in** E **do**
  if $h \wedge B \wedge H \vdash e$ **then**
    $Counter = Counter + 1$
    if $Counter = Limit$ **then break**
return $Counter$

Figure 2. Lazy evaluation algorithm. The type of laziness (lazy evaluation of *positive* examples, lazy evaluation of *negative* examples and *total* laziness) is defined by the parameter ***LazyType***.

In some applications, an hypothesis is allowed to cover a small number of negative examples (the *noise* level). If a clause covers more than the allowed number of negative examples it must be specialized. Otherwise, the search for further refinements of the partial clause terminates there. In these circumstances lazy evaluation of the negative examples should be useful. When using the lazy evaluation of negatives, we are only interested in knowing if the hypothesis covers more than the allowed number of negative examples or not. One is not really interested to know how much more negative examples a hypothesis covers than allowed. Hence, testing stops as soon as the number of negative examples covered exceeds the allowed noise level or when there are no more negative examples to be tested. The noise level is quite often very close to zero and therefore the number of negative examples tested in each clause may be very small. It is also common that the negative examples largely outnumber the positives. If the heuristic does not use the negative counting then this produces exactly the same results (clauses and accuracy) as the non-lazy approach but with a reduction on the number of negative examples tested.

3.2. Lazy Evaluation of Positives

One may also allow the positive cover to be evaluated lazily. A partial clause must be either specialized (if it covers more positives than the best consistent clause found so far) or justifiably pruned away otherwise. When using lazy evaluation of positives we start by determining if an hypothesis covers more positives than the current best consistent hypothesis. If it does, we then evaluate the positive examples just until we exceed the best cover so far. If the best cover is exceeded we retain the hypothesis (either accept it as final if consistent or refine it otherwise), otherwise we may justifiably discard it. We need to evaluate its exact positive cover only when accepting a consistent hypothesis. In the event of this latter case we do not need to restart the positive coverage computation from scratch, we may simply continue the test at the point where we left it before the negative coverage computation. Also in this later case we update the best positive counting.

3.3. Total Laziness

As pointed out earlier, generating hypotheses is computationally very cheap. We may thus consider the possibility of generating more hypotheses if that contributes to the reduction of the effort of evaluating them. We may therefore go a bit further with the lazy evaluation proposal and simply do not evaluate the positive cover at all for partial clauses. We may design the evaluation of an hypothesis in a two step process. In the first step we perform a lazy evaluation of negatives examples. If the clause is inconsistent then we are done, no extra evaluation effort is required and the clause is retained for specialization. On the other hand if we find a consistent clause then an exact positive coverage is carried out. The advantage of the total laziness is that for each partial clause we only test it on the negatives until it covers a number of negative examples equal to the noise level plus one.

One should note that in systems that constrain the number of hypotheses generated, it is necessary to relax the nodes limit (i.e., increase the limit of generated hypotheses). Although we may generate more hypotheses, we may still gain by the increase in speed of their evaluation

process since computational cost of generating hypotheses is usually much smaller than that of evaluating them.

3.4. Lazy Evaluation Considerations

When performing lazy evaluation of positive and negative examples we may use a breadth-first search strategy. This seems a good choice given that, in most applications, one looks for short clauses, that are very close to the top of the subsumption lattice.

Some systems like Progol [24], Aleph [52] or IndLog [6] rely on heuristics to guide the search for an hypothesis. If the search is complete then the role of the heuristic is just to improve speed, the final hypothesis should be independent of the heuristic used. Progol, for example, uses a *best first* algorithm in which the heuristic value of each clause is computed on the basis of the number of positive and negative examples covered and length of the clause. It is important for such heuristic to determine the exact number of examples covered. Note that performing lazy evaluation limits the use of heuristics and pruning, because the computed coverage for a given clause may be inaccurate.

Lazy evaluation may be used together with other ILP speedup techniques to further increase the efficiency of the system. Lazy evaluation of negative examples may be used with the coverage caching technique described in Section 2.3. Lazy evaluation of either negative or positive examples cannot be used simultaneously with the technique called *lazy evaluation of literals* [49]. To compute the constant values of the lazily evaluated literals, all of the positive and negative covered examples have to be computed exactly. Furthermore, lazy evaluation of negatives is not applicable in data sets with positive examples only, or when a compression measure [24] or an user defined cost function[49], as available in Aleph and IndLog, is used.

The lazy evaluation technique is also directly applicable to stochastic search [48] since this type of search does not rely on heuristics. Stochastic search has been shown to be adequate for very large search spaces. Hence, combining lazy evaluation with stochastic search may be useful to search very large (intractable) search spaces.

4. Tries

Tries were originally invented by Fredkin [17] to index dictionaries and have since been generalized to index recursive data structures such as terms. Please refer to [31, 23, 3, 19, 36] for the use of tries in automated theorem proving, term rewriting and tabled logic programs.

The basic idea behind the trie data structure is to partition a set T of terms based upon their structure so that looking up and inserting these terms will be efficiently done. The trie data structure provides complete discrimination for terms and permits look up and possibly insertion to be performed in a single pass through a term.

4.1. Applicability

An essential property of the trie structure is that common prefixes are represented only once. The efficiency and memory consumption of a particular trie data structure largely depends on

the percentage of terms in T that have common prefixes. For ILP systems, this is an interesting property that we can take advantage of because the hypothesis space is structured as a lattice and hypotheses close to one another in the lattice have a largely common structure. More specifically, hypotheses in the search space have common prefixes (literals), and some related information is also similar (e.g. the list of variables in an hypothesis is similar to other lists of variables of nearby hypotheses). This clearly matches the common prefix property of tries. We thus argue that tries form a promising alternative for storing hypotheses and some associated information.

4.2. Description

At the entry point of a trie we have the root node. Internal nodes represent symbols in terms and leaf nodes specify the end of terms. Each root-to-leaf path represents a term described by the symbols labelling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing symbol. Inserting a new term requires traversing the trie starting at the root node. Each child node specifies the next symbol to be inspected in the input term. A transition is taken if the symbol in the input term at a given position matches a symbol on a child node. Otherwise, a new child node representing the current symbol is added and an outgoing transition from the current node is made to point to the new child node. On reaching the last symbol in the input term, we reach a leaf node in the trie.

Figure 3 presents an example for a trie with three terms. Initially, the trie contains the root node only (not shown in the figure). Next, we insert $f(X, a)$. As a result, we create three nodes: one for the functor $f/2$, next for the variable X, and last for the constant a (Figure 3(a)). The second step is to insert $g(X, b, Y)$. The two terms differ on the main functor, so tries bring no benefit here (Figure 3(b)). In the last step, we insert $f(Y, 1)$ and we save the two nodes common with term $f(X, a)$ (Figure 3(c)).

Notice the way term tries represent variables. We follow the formalism proposed by Bachmair et al. [3], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $<\text{VAR}_0, ...,\text{VAR}_N>$, such that $numbervar_t(X) < numbervar_t(Y)$ if X is encountered before Y in the left-to-right traversal of t.

4.3. Implementation

The trie data structure was implemented in `C` as a shared library. Since the ILP system we used for testing is implemented in Prolog, we developed an interface to tries as an external Prolog module.

Tries are implemented by representing each trie node by a data structure with four fields each. The first field stores the symbol for the node. The second and third fields store pointers respectively to the first child node and to the parent node. The fourth field stores a pointer to the sibling node, in such a way that the outgoing transitions from a node can be collected by following its first child pointer and next the list of sibling pointers of this child. Figure 4 illustrates the actual implementation for the trie presented in Figure 3. Observe that we can collect all children of a node, and that we can always reach the root from a leaf node.

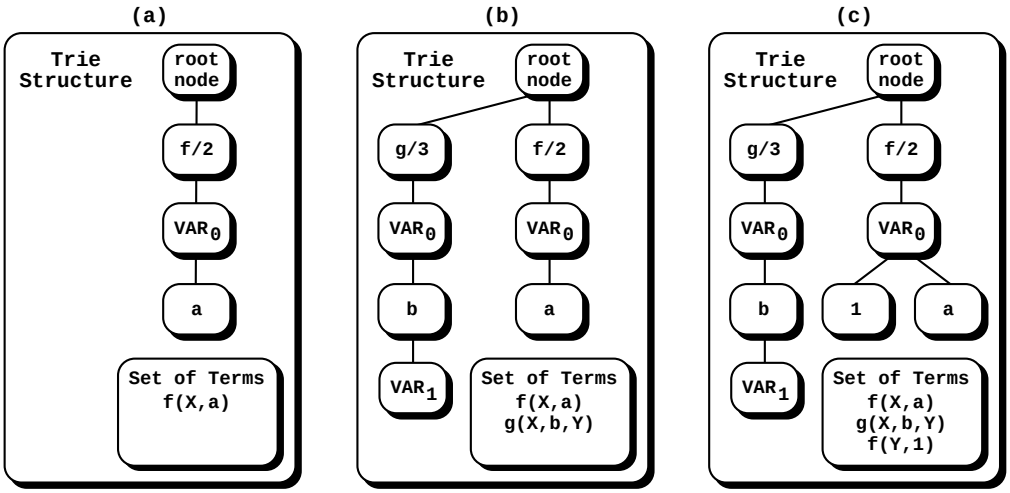


Figure 3. Using tries to represent terms.

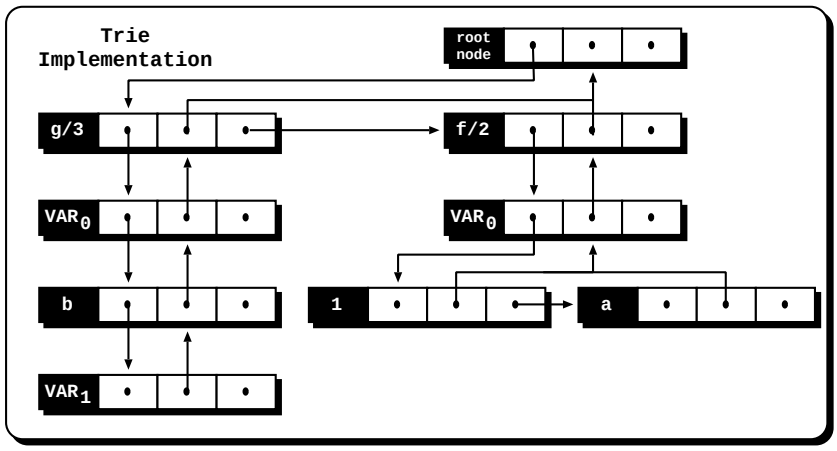


Figure 4. The implementation of the trie in Figure 3(c).

Inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

Searching through a chain of sibling nodes that represent alternative paths could be too expensive if we have a large number of nodes. To avoid this problem we extended the trie data structure with an hashing mechanism. A threshold value (8 in our implementation) controls whether to dynamically index the sibling nodes through a hash table, hence providing direct node access and optimizing search. Further, hash collisions are reduced by dynamically expanding the hash tables.

When reconstructing a term back as a Prolog term, the trie nodes for the term in hand are traversed in bottom-up order, starting from its leaf node until reaching the root node. The trie structure is not traversed in a top-down manner because the insertion and retrieval of terms is an asynchronous process, new trie nodes may be inserted at *anytime* and *anywhere* in the trie structure. This induces complex dependencies that limit the efficiency of alternative top-down reconstructing schemes.

As a final note we should mention that besides atoms, integers, variables and compound terms (functors) presented in the examples, our implementation also supports terms with floats and pairs (lists).

5. RL-Trees

The RL-tree (**R**ange**L**ist-Tree) data structure is based on a generic data structure called quadtree [39] that has been largely used in application areas such as Image Processing, Computer Graphics, or 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. Quadtrees 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 data structure is designed to store lists of intervals representing integer numbers. For example, the list $[1, 2, 5, 6, 7, 8, 9, 10]$ is represented as the list of intervals $[1 - 2, 5 - 10]$.

5.1. Applicability

To reduce the time spent on computing clauses coverage some ILP systems, such as Aleph [52], FORTE [37], IndLog [6], and April [15], maintain lists of examples covered (coverage lists) for each hypothesis that is generated during execution.

The data structure used to maintain coverage lists in systems like IndLog and Aleph are Prolog lists of integers. Two lists are kept for each clause: the list of positive examples covered (numbered from 1 to $\mid E^+ \mid$) and the list of negative examples covered (numbered from 1 to $\mid E^- \mid$). Each example is represented by a number in the list. The above mentioned systems reduce the size of the coverage lists by transforming a list of numbers into a list of intervals. Using a list of intervals to represent coverage lists is an improvement over 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. Second, the representation of lists in Prolog is not very efficient regarding memory usage. To tackle these problems, we propose the RL-tree data structure to achieve efficient storage and manipulation of coverage lists, including fast insertion, removal and retrieval of data. The RL-trees can be implemented in any ILP system be it Prolog based or not.

5.2. Description

In the design and implementation of RL-trees, the following assumptions were made: intervals are disjoint; updates consist of adding or removing numbers; and, the domain (an integer interval) is known at creation time.

RL-trees have two distinct types of nodes: list nodes (represent a fixed interval of size LI) and range nodes (represent an interval that is subdivided into sub-intervals). The number of sub-intervals in each *range node* is B (an implementation parameter).

In the vein of quadtrees, each "cell" (square) in a node is painted black if the example is covered (list nodes) or the interval is completely covered (range nodes). It is painted gray if the interval is partially covered (range nodes) and is white if the example is not covered (list nodes) or the interval is not covered at all (range nodes).

Again, as in quadtrees, the underlying idea of RL-trees is to represent a disjoint set of intervals in a domain by recursively partitioning the domain interval into equal sub-intervals. 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 $[1, NE]$ (denoted as RL-tree(NE)) where NE is the number of positive or negative examples.

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.

Figure 5 represents different intervals in a RL-tree(65) with the LI and B parameters set to 16 and 4, respectively. Each group of four squares represents a range node and each square in a range node corresponds to a sub-interval. A sixteen square group represents a list node and each square in a list node corresponds to an integer. By applying the properties described above one knows that the maximum height of an RL-tree(65) is 2 and that the maximum root node range is $[1 - 256]$. Thus, each sub-interval of the root node represents an interval of 64 integers. Figure 5(a) and 5(b) show, respectively, the empty ([ ]) and complete ($[1 - 65]$) intervals. Figure 5(c) presents a more elaborated interval ($[1 - 32, 53 - 54, 56 - 58, 60 - 65]$). Note however that, even though it seems more complex, the number of nodes is the same as for interval [1] .

5.3. Implementation

As the trie data structure, the RL-tree was implemented in `C` as a shared library. We have also developed an interface to RL-tree as an external Prolog module for the ILP system used

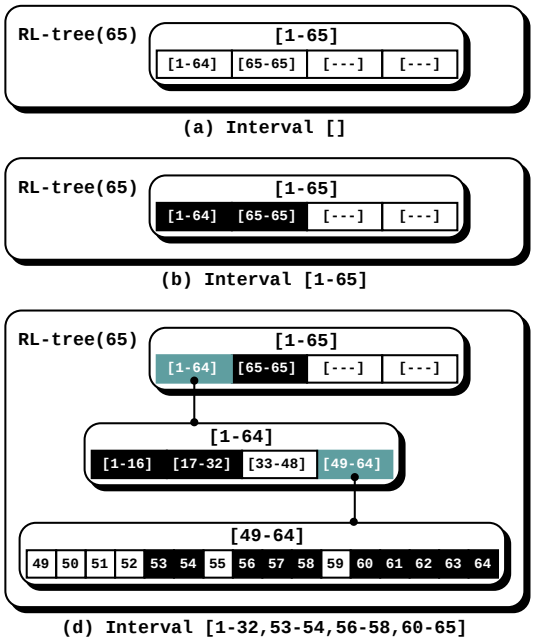


Figure 5. Representing intervals in a RL-tree(65).

in the experiments. Both Tries and RL-trees are now publicly available and integrated in the Yap Prolog system[‡].

Like other quadtree data structures [40], 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 sub-intervals. Each sub-interval has a color associated (white, black, or gray) that is coded using 2 bits (thus a total of 8 bits are used for the 4 sub-intervals). The other 8 bits are used to store the number of sub-nodes of a node. This information is used to improve efficiency by reducing the need to traverse the tree to determine the position, in the array, of a given node. The list nodes use 16 bits, one bit for each number (example). The number interval represented by a list node is inferred from its parent range node.

[‡]`http://www.dcc.fc.up.pt/~vsc/Yap`

The operations implemented for a RL-tree(N) and their complexity (regarding the number of sub-intervals 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: $O(H)$

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 $C=20$.

There are other data structures that may be used to achieve the same purpose of RL-Trees, namely bit-vectors and C-lists. In the worst case scenario, a RL-Tree consumes more memory than the bit-vectors and substantially less memory than C-lists. However, this worst case scenario should not occur, or occur very rarely. On average, a RL-tree consumes less memory than bit-vectors or C-lists but is slightly slower than bit-vectors. Note that in the best case scenario a RL-Tree only requires a single byte compared to a constant size for the bit-vectors, which is specially important for large data sets.

6. Indexing Logic Programs

Many ILP systems use logic programming technology in their implementation. It is therefore legitimate to ask whether inefficiencies in logic programming impact ILP system performance, and whether logic programming systems can be improved towards achieving greater scalability in ILP.

Most of the applications of ILP we have seen largely learn from extensional data, that is, from large *static* tables of symbols or numbers. Computation of numerical values or manipulation of data structures seems to be less frequent. Relations often have large arities (although some practitioners do prefer binarised relations) and queries follow a variety of instantiation patterns. This suggests the need for a sophisticated indexing mechanism.

ILP systems further need to maintain a large search space. This is a database maintenance problem, but now over a dynamic data structure which suffers frequent updates. This creates a rather more complex situation where the effort in building indices may be more than the actual advantages gained from their usage.

In the following discussion we give a brief overview of how these issues have been studied in the Yap Prolog system [42, 45].

6.1. Static Indexing

Many modern Prolog systems follow David H. D. Warren's work on the Warren Abstract Machine (WAM) [53] and implement a simple form of indexing which relies on the main

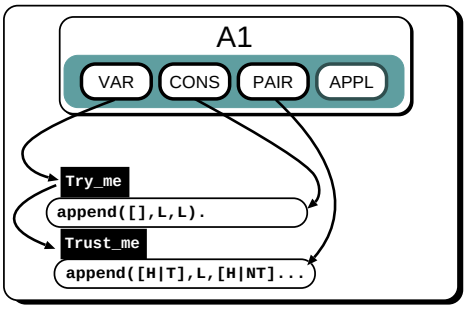


Figure 6. WAM indexing for `append/3`.

functor of the *first* argument to select a subset of all clauses. If this subset has several clauses, Prolog systems try clauses one by one. As an example, consider a well-known list concatenation predicate that says that the third argument is the concatenation of the first two arguments:

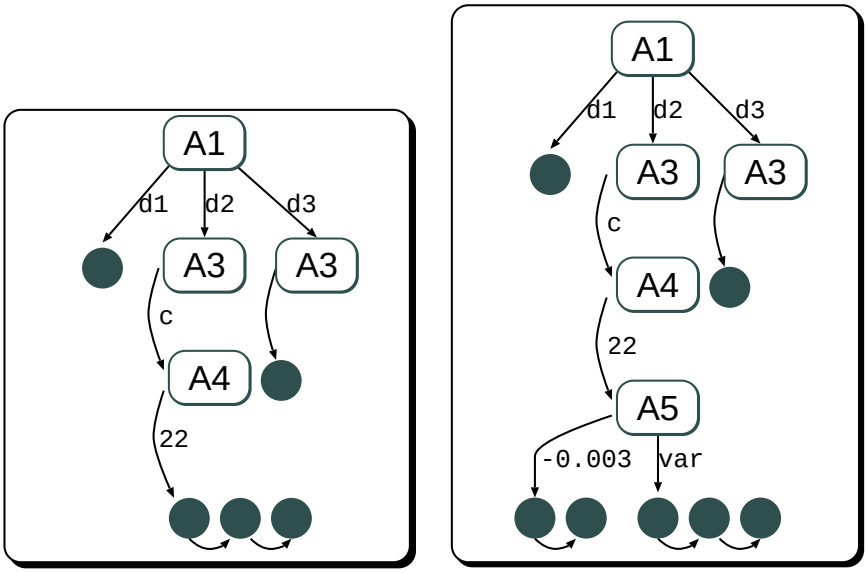
```
append([], L, L).
append([H|T], L, [H|NT]) :- append(T, L, NT).
```

Figure 6 shows the basic structure of the WAM-based indexing code for the predicate. The top node is known as a *type-switch node*: it dispatches according to the first argument's type. If the first argument, A_1 is unbound, all clauses will be tried in sequence. If A_1 is a non-empty list, only the second clause needs to be considered. If A_1 is a constant the first clause will be tried. Last, if A_1 is bound to something else the procedure is guaranteed to fail. Note that, in this code, each clause can be accessed in two different ways: one is when A_1 is unbound, the other when A_1 is bound.

Using traditional WAM indexing on ILP applications shows several limitations. First, ILP systems do not always have the first argument bound. It is quite possible that the second argument is the only one bound. It is also possible that the first and the second would be bound together. Or maybe only the third. In such cases, the WAM indexing code may result in having to try every clause. On the other hand, generating indexing code for every argument would be extremely space-expensive [34, 46].

We propose *just-in-time indexing*: our idea is to apply the Just-In-Time compilation strategy [2] so that we will generate all indices required, and only the indices required. We build the tree from similar building blocks to the WAM but we **(i)** generate indices based on the instantiation on the current goal; and **(ii)** expand indices given different instantiations for the same goal.

Figure 7(a) shows a compact representation of an indexing tree for the `atm/5` predicate that is part of the background knowledge in the *Carcinogenesis* data set. In what follows we assume that the background knowledge contains definitions of the `atm/5` predicate for 3 molecules: `d1`, `d2` and `d3` (the A_1 argument). The tree is built given the initial query:



(a) (b)

Figure 7. JIT indexing for `atm/5`: (a) after the initial query `?- atm(d2,B,c,22,X)`; (b) after the query`?- atm(d2,B,c,22,-0.003)`.

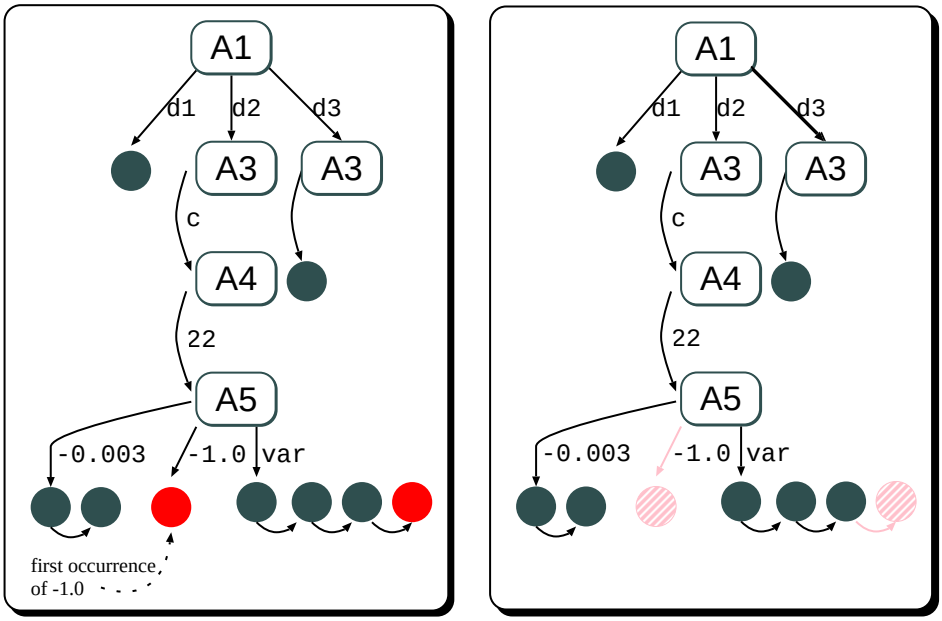
```
?- atm(d2,B,c,22,X).
```

Through query observation we find three bound arguments: A_1, A_3, and A_4. In its first invocation, the algorithm builds all indices for A_1. If A_1 does not determine a clause, the algorithm builds a sub-tree for $A3$, and last it uses A_4. As an example, if A_1 is `d1`, we can commit to a clause immediately, so the algorithm stops there. If A_1 is `d2`, we have to walk through the other two arguments. In the case $A_3 =$ `c` and $A_4 = 22$ we still have three matching clauses, so we need to setup a chain of clauses. Last, if A_1 is bound to `d3` it is sufficient use A_3 to commit. The full tree gives us perfect indexing for the cases where A_1, A_3, A_4 are bound, and that A_2 and A_5 are unbound.

As the ILP search generates more hypotheses, we may again call `atm/5`, for example with more instantiated queries, say:

```
?- atm(d2,B,c,22,-0.003).
```

Figure 7(b) shows the result. The algorithm expand the existing tree using the extra information that A_5 is also bound. The algorithm follows a lazy heuristic and only expands the path $A_1 =$ `d2`, $A_3 =$ `c`, $A_4 = 22$. Other paths may be expanded later, if they benefit from more instantiated arguments.



(a) (b)

Figure 8. (a) Assert on `atm/5`. (b) Retract on `atm/5`.

The tree is thus expanded to support checking A_5. Notice that we do not throw the previous code away: the switch node for A_4 forks depending on whether A_5 is bound or not. The new code supports the case when A_5 is bound, the old case corresponds to the case A_5 is unbound. Actual implementation is achieved by leaving *hooks* in the switching tree whenever we skip originally unbound arguments. Although such hooks can spend a significant amount of space and somewhat delay execution, they do bring flexibility.

6.2. Dynamic Indexing

Consider now that we have update operations such as `assert` or `retract`. One alternative would be simply to destroy the indexing tree every time we change the database. Unfortunately, in the worst case doing so might result in generating an entire tree every time we query. A less expensive solution is to to manipulate the indexing tree so that it is always consistent with the database.

Figure 8 (a) shows an example where we insert a new clause for `atm/5`:

```
?- assert(atm(d2,B,c,22,-1.0)).
```

The clause differs from a previous one only in the last argument, so we only actually have to update the value switch node for A_5 and insert the new clause in the unbound chain for A_5. Our algorithm walks the index tree in order to find the minimal set of expansions to a tree. If we cannot find such a set, the algorithm kills the tree and restarts from scratch.

The algorithm for retracting follows the same lines. Consider:

```
?- retract(atm(d2,B,c,22,-1.0)).
```

That is, we now decide to remove the clause we just added. The necessary operations are the reverse of the insert operations: we remove the branch for $A_5 = -1.0$ and we contract the chain for $var(A_5)$, as shown in Figure 8 (b).

7. Experiments and Results

In order to assess the impact of our proposals on memory usage and execution time, we next present a detailed study for a diverse set of ILP applications. Most of these techniques do not affect accuracy, but regarding the lazy evaluation technique, we also study the impact on the accuracy of the models found.

7.1. Experimental Settings

Our experiments were performed on an AMD Athlon(tm) MP 2000+ dual-processor PC with 2 GB of memory, running the Linux RedHat (kernel 2.4.25) operating system. We used the Yap Prolog system versions 4.4 (last release with the traditional indexing scheme) and 4.5 (first release with the new indexing scheme), and the April ILP system version 0.9 [15]. We chose Yap because it is one of the fastest available Prolog systems [41, 11] and April due to our knowledge regarding its implementation. However, we should point out that our proposals are also applicable to other ILP systems. We performed 3-fold cross-validation. The values presented throughout this section are the average of the values obtained on the three folds.

The April system uses Prolog. It implements a greedy covering algorithm similar to the one described in Section 2. April was configured to perform breadth-first search to find each hypothesis. The hypotheses generated during a search are evaluated using a heuristic that relies on the number of positive and negative examples. Recall that with lazy evaluation we do not have the exact coverage value for an hypothesis, thus potentially leading to a different ordering of hypotheses and, therefore, different final theories.

We used a total of thirteen data sets in our experiments. The data sets were downloaded from repositories at the Universities of Oxford[§], York[¶], Tokyo[‖], Torino [**], and from the MLnet Online Information Service[††].

[§] `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn`
[¶] `http://www.cs.york.ac.uk/mlg`
[‖] `http://www.ia.noda.sut.ac.jp/ilp`
[**] `http://www.di.unito.it/ mluser/challenge`
[††] `http://www.mlnet.org`

Table I. Data sets and settings used in the experiments.

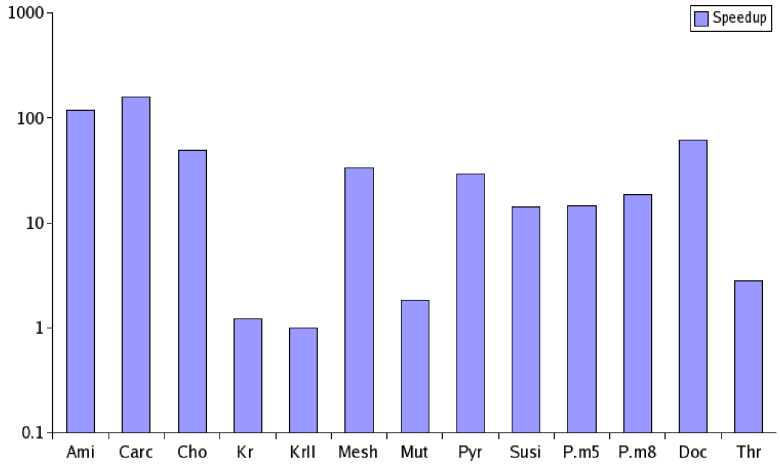
Data Set	Characterisation			Settings			
	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$	*nodes*	*i-depth*	*noise*	*minpos*
Amine uptake	343	343	32	2,500	2	10	20
Carcinogenesis	182	155	38	5,000	2	5%	20%
Choline	663	663	32	4,000	3	5%	22
Krki	342	658	1	2,000	1	10	2
Krki ***II***	3,241	6,760	1	2,000	1	10	2
Mesh	2841	290	29	2,000	3	25	100
Mutagenesis	125	63	21	2,000	2	2%	20%
Pyrimidines	1394	1394	244	2,500	2	25	50
Susi	252	8,979	18	2,000	7	4%	75%
P.m5.l15	200	200	5	30,000	1	0	100%
P.m8.l12	200	200	8	8,000	1	0	100%
Doc understanding	177	708	57	3,000	3	0	2
Throughdoor	21	135	6	10,000	2	0	10%

Table I characterizes the data sets in terms of number of examples (positive and negative) as well as background knowledge size. Furthermore, it shows the main settings used on each data set. The parameter *nodes* specifies an upper bound on the number of hypotheses generated during the search for an acceptable hypothesis. In order to reduce the run time of the experiments we limited the search space using relatively small search spaces, through a low value for the *nodes* parameter. This has the effect of reducing the total memory usage and execution time needed to process a data set, but at the cost of finding possible worse theories. The *i-depth* [26] corresponds to the maximum depth of a literal with respect to the head literal of the hypothesis. The *noise* parameter defines the maximum number or percentage of negative examples that an hypothesis may cover in order to be accepted. Finally, *minpos* specifies the minimum number or percentage of positive examples that an hypothesis must cover in order to be accepted.

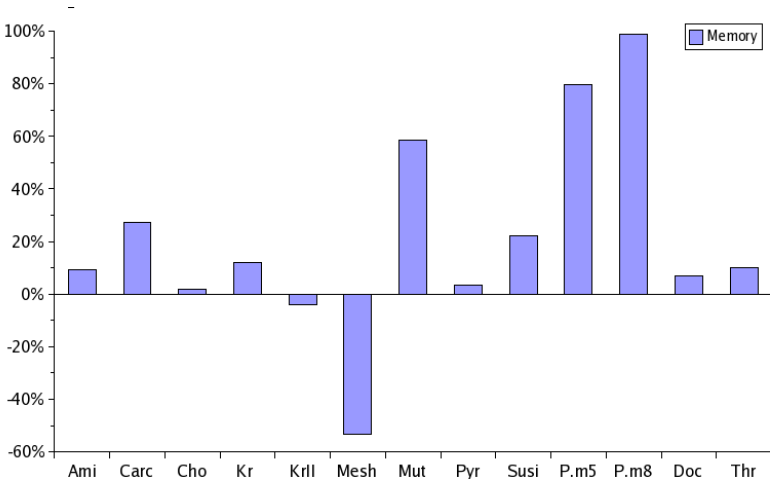
7.2. JIT Indexing

We start by studying the impact of the new indexing scheme. Figure 9 shows the impact in execution time (a) and memory usage (b) that resulted from using April with Yap 4.5 compared to Yap 4.4. Remember that Yap 4.4 performs indexing on the first argument of static predicates and no indexing for dynamic predicates, while Yap 4.5 implements the new indexing scheme. A more complete set can be found in Table AI in the Appendix.

The results clearly show that Yap 4.5 is significantly faster than Yap 4.4. Yap 4.5 achieves best improvements for data sets with long running times such as ***Ami*** and ***Carc***, with superb



a. Speedup



b. Variation in memory usage

Figure 9. Impact of the JIT indexing scheme.

speedups up to 100 times faster. The worst results are observed in ***Kr*** and ***KrII***, where the two versions of Yap achieve almost the same performance. These are also the two data sets that take less time to run. The ***Mut*** data set is also interesting in that it has long running times but a relatively small speedup (only twofold). This happens because the most time is spent on recursive procedures where the more sophisticated indexing does not help as much.

Building the intermediate indices may result in significant increases in memory usage. This is indeed the case for ***Mut***, ***P.m5***, and ***P.m8***. Note that the ***P.m*** data sets have a 10 fold increase in performance at the cost of increasing memory usage. It is interesting to observe that the newer version of Yap can actually require less space for one specific data set, ***Mesh***. The newer implementation of indexing simplifies the implementation of logical update semantics, thus allowing recovering space more easily.

The results demonstrate that indexing is indeed a major factor in the performance of ILP applications. Although further work should be done to improve performance in data sets such as ***Mut*** and memory usage in ***P.m5*** and ***P.m8***, we believe that the results show that we are following the correct path. Therefore, in the remainder of the study we use Yap 4.5.

7.3. Lazy Evaluation

We next study the impact of using lazy evaluation. Table II shows the impact of using the three different forms of lazy evaluation: on the negatives, on the positives, and on both. We measure the variation in: the number of hypotheses generated (|S|); accuracy on unseen examples (*Acc*); and execution time (*Time*). The references for the values in the tables of this section are the values obtained by April using no optimization and the same breadth-first search strategy.

The results presented reflect the impact of using lazy evaluation over coverage caching. Both techniques, lazy evaluation and coverage caching, aim at reducing the number of examples used during hypotheses evaluation, hence reducing the execution time. Thus we believe it is paramount to see how these two techniques behave together. For completeness, Table AII in the Appendix presents results of using only lazy evaluation.

Best results were obtained with lazy evaluation of negatives and total laziness, both with an average reduction of 20% in the execution time. Accuracy and search space were largely unchanged. Total laziness does not always improve execution time over laziness in negatives. An example is ***P.m5***, where using total laziness results in increased search space, and, in turn, in longer execution times. The worst results were observed with the lazy evaluation of positives, with an average increase in the execution time of 146%. The impact on accuracy is practically nonexistent.

Overall, lazy evaluation of negatives reduced execution time and had a small effect on search space and accuracy. To explain these results it is important to recall how cover removal proceeds in MDIE systems. Theory construction in MDIE systems use a greedy cover algorithm that proceeds seed by seed. In general, the algorithm prefers focusing the search on clauses with high positive cover. On the other hand, the number of clauses constructed per seed is bound by the *nodes* parameter. Most of the time, in our experiments, this limit is actually reached. When search stops because the limit is reached, only a part of the search space is explored. Thus, the algorithm may terminate without finding a clause or a with a clause that only covers few

Data Set	Negative			Positive			Total Laziness		
	$\mid S \mid$	Acc	Time	$\mid S \mid$	Acc	Time	$\mid S \mid$	Acc	Time
Ami	-1%	0%	-24%	20%	-1%	31%	3%	0%	-26%
Carc	0%	0%	-10%	6%	7%	-5%	4%	4%	-3%
Cho	4%	0%	-12%	66%	0%	204%	-5%	0%	21%
Kr	0%	0%	-20%	38%	0%	141%	0%	0%	-41%
KrII	0%	0%	-43%	276%	0%	-75%	6%	0%	-75%
Mesh	0%	0%	-1%	30%	-4%	132%	3%	-1%	-22%
Mut	-4%	-1%	-1%	0%	0%	1460%	-5%	-1%	-49%
Pyr	2%	0%	-33%	11%	2%	-44%	-1%	0%	-78%
Susi	0%	0%	-88%	191%	0%	-86%	0%	0%	-84%
P.m5	0%	0%	2%	39%	0%	95%	39%	0%	92%
P.m8	0%	0%	2%	3%	0%	-2%	3%	0%	8%
Doc	2%	0%	-35%	12%	0%	-25%	11%	0%	-32%
Thr	0%	0%	-3%	4162%	0%	70%	26%	0%	24%

Table II. Impact of lazy evaluation.

examples. In either case, the cover removal step will only remove a small number of examples. This leads to more seeds being considered (more iterations of the algorithm).

Lazy evaluation of negatives preserves "correct" counting for the positives and uses that information to, while searching, first expand clauses with high positive cover. Therefore we would not expect significant differences from search without lazy evaluation of negatives. However, if positives are counted "incorrectly" (lazy evaluation of positives and total laziness) the clauses with high positive cover may not be chosen first and the search may terminate (due to the *nodes* limit) before they are generated. In this later case, the number of constructed clauses may differ significantly from a search that does not use lazy evaluation since the algorithm will need more iterations to complete, i.e., to cover all the positive examples.

We next present a more detailed analysis of the results for each lazy evaluation approach.

7.3.1. Lazy Evaluation of Negatives

Lazy evaluation of negatives reduces the execution time in almost all data sets considered, and it has a small effect on search space and accuracy.

Figure 10 gives more information about the behavior of lazy evaluation of negatives. It shows that the improvements in the execution time tend to be better on the data sets with greater percentage of negative examples. This is supported by the results observed in the ***Kr*** and ***KrII*** data sets where, in spite of differing only on the number of examples, the reduction in the second data set was twice the reduction observed in the first. The ***Susi*** data set, the data set with the largest number of negative examples, has the best results with a reduction in the execution time of 88%.

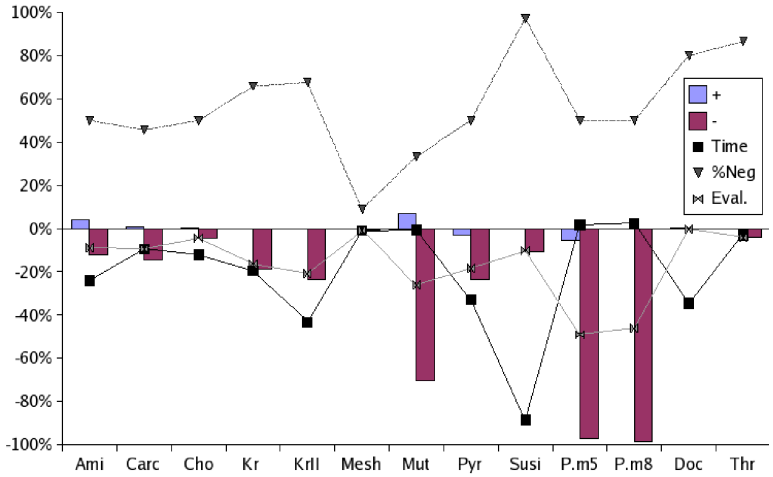


Figure 10. Impact of lazy evaluation of negatives. It presents the variation in the number of positive (+) and negative (−) examples evaluated, the variation in the execution time, the weight of the negative examples on the total number of examples ($\%Neg$), and the variation in the number of examples evaluated ($Eval$).

One can also observe that, in general, the reduction in the execution time follows the variation on the number of examples evaluated. The exception is observed in the ***P.m8*** and ***P.m5*** data sets, where the decrease in the total number of examples evaluated lead to an increase in the execution time of 2%. We know, from observing Table AII (in the Appendix), that lazy evaluation of negatives reduces the execution time when not used together with coverage caching. Therefore, the cause of the 2% increase in the execution time is the overhead of using coverage caching with lazy evaluation.

In summary, lazy evaluation of negatives reduced considerably the execution time without affecting negatively the accuracy of the models found. The reduction on the execution time tends to increase as the number of negative examples increases.

7.3.2. Lazy Evaluation of Positives

The results observed using lazy evaluation of positives are, in general, considerable worse compared to those of lazy evaluation of negatives. In most cases, lazy evaluation of positives increased considerably the execution time and search space. The exceptions were only observed in ***KrII***, ***Pyr***, ***Susi***, ***P.m8***, and ***Doc*** data sets, where the execution time decreased (ranging from 2% to 86%). We show in Figure 11 a more detailed analysis using the same parameters as in the previous approach.

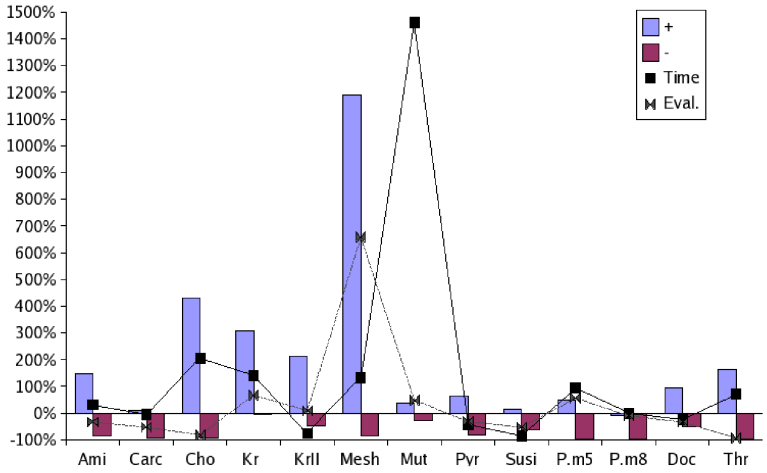


Figure 11. Impact of lazy evaluation of positives.

The good results on the data sets mentioned above can be explained by the reduction on the number of examples evaluated (*Eval*), and, simultaneously, by the small increase in the search space. The bad results follow from an increase of the search space, together with an increase in the number of positive examples evaluated.

It is curious to note that, for ***Thr***, the execution time increased, notwithstanding of the reduction on the number of examples evaluated. This indicates that, at least in this case, the time to evaluate a positive example is higher than the time to evaluate a negative example. The figure also makes it clear that the cost of evaluating examples varies significantly among the data sets. For instance, an increase of 50% in the number of examples evaluated in ***Mut*** resulted in a increase of more than 1400% in the execution time, while in ***Mesh*** an increase of 700% on the number of examples evaluated *only* increased the execution time by 100%. This clearly shows that the cost of evaluating an example in ***Mut*** is significantly higher than in ***Mesh***.

In summary, lazy evaluation of positives can lead to significant increases in the search space. As it turns out, this can actually result in evaluating more positive examples, thus leading to longer runtimes.

7.3.3. Total Laziness

The results observed with total laziness show large variations depending on the data set. For nine data sets the execution time decreased (46% on average), and for the other four it increased (36% on average). Figure 12 shows more details. It presents the variation in the number of positive (+) and negative (−) examples evaluated, the variation in the execution

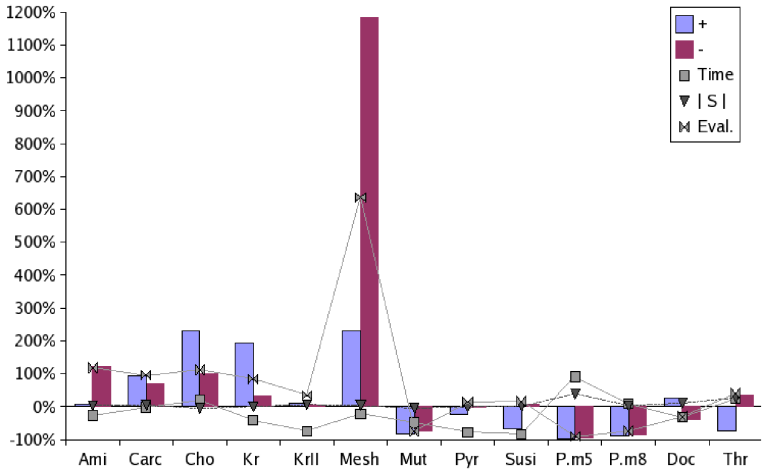


Figure 12. Impact of total laziness.

time, the variation in the number of hypotheses generated (| S |), and the variation in the number of examples evaluated (*Eval*).

The major reduction in the execution time was observed on the ***Mut*** data set, mainly because this data set pays a high cost of evaluating the examples against the hypotheses generated. Curiously, in the ***P.m5*** and ***Thr*** data sets there was an increase in the execution time, in spite of the decrease in the number of examples evaluated. In these data sets, the cost of evaluating an example against an hypothesis is very low, even when compared to the cost of generating one hypothesis. Thus, the increase on the number of hypotheses generated results in an increase in execution time.

In summary, for total laziness, the gains on the execution time depend, mainly, on the cost of evaluating the hypotheses against the examples. Thus, this technique seems to be well suited for data sets where the cost of evaluating an hypothesis against all examples is high.

7.4. Tries

We have used the trie data structure in the April system to store information about the hypotheses. In fact, the trie data structure was used as a repository of Prolog terms (hypotheses and lists), i.e., it was a substitute for Prolog's internal database. It was used to store the hypotheses themselves (Prolog clauses), the lists of variables in the hypotheses, the lists of unbound variables in the hypotheses heads, and the lists of free variables in the hypotheses. For each hypothesis, the quartet of references that represents this information in the trie, is grouped together and stored by the April system in the Prolog internal database in an unique

record associated with the hypothesis. When a record is fetched, the four references are used to reconstruct the terms previously stored in the trie.

We next show, in Table III, the variation in the execution time and memory usage when using tries compared to not using them. Note that the *Memory* parameter only includes the memory used to store the information described above. The table shows that tries significantly reduced memory consumption on all data sets. This reduction on memory usage increased 30% when compared to initial results [14]. This improvement was a consequence of modifications on April's code that reduced the data stored outside the trie for the hypotheses generated, thus increasing the weight of the terms stored in the trie. Regarding execution time, we expected a slightly overhead as a result of using tries. This overhead is, on average, 5%.

Figure 13 shows, in more detail, how the variation in the execution time and memory usage of each data set relates with the total number of examples ($\%E$), total number of hypotheses ($\mid S \mid$), total execution time ($\%Time$), and total memory usage ($\%Mem$) on all data sets. More precisely, we present the percentage in the total for each data set. In general, the overhead in the execution time is more reduced in the data sets with longest execution times. This suggests that the overhead of using tries tends to decrease as the execution time increases. Further experiments are required to confirm this.

In conclusion, the tries data structure significantly reduces memory consumption with minor costs in the execution time.

7.5. RL-Trees

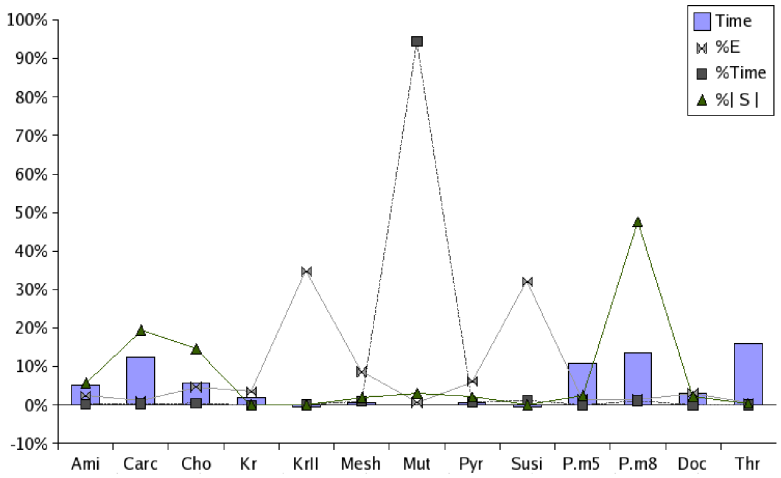
Table III shows the impact on execution time and memory usage of using RL-trees. The *Memory* parameter only accounts for the memory used to store coverage lists.

RL-trees accomplish our goal on reducing memory consumption. Compared to Prolog range lists, the use of RL-tree reduced memory usage by 36%, on average. In general, these reductions are achieved almost with no execution time overhead. In fact, we observed an average reduction of 39% in the execution time. This is a substantial improvement when compared to the initially reported results on RL-trees [14]. This improvement is mainly due to optimizations in the RL-trees implementation.

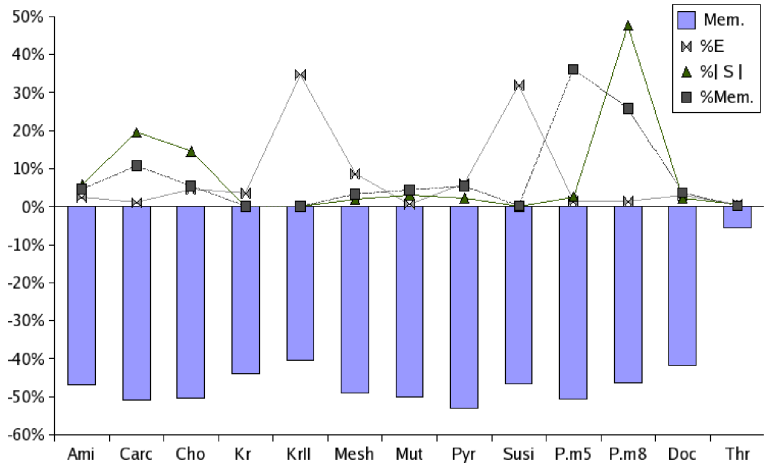
The worst results were observed in the two artificial data sets, ***P.m8*** and ***P.m5***. In these data sets the use of RL-trees lead to an increase in execution time and memory usage. The behaviour, in these cases, results from the fact that most of the hypotheses in the search space cover all positive and almost all of the negative examples. Thus, for these data sets, RL-trees are being compared with the best case for Prolog range lists.

Figure 14 shows more details about the variation in the execution time and memory usage for each data set. By observing the figure, it is clear that the reductions on the execution time and memory usage are directly related with the total number of examples (parameter $\%E$). Therefore, better performance is achieved in data sets with greater number of examples.

In conclusion, the RL-tree data structure clearly reduces memory usage without degrading the execution time. In fact, for almost all data sets, it also significantly reduces execution time.

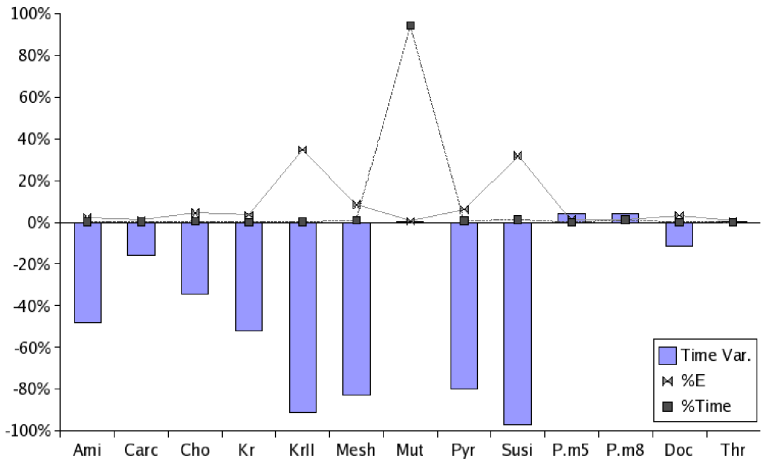


a. Time analysis

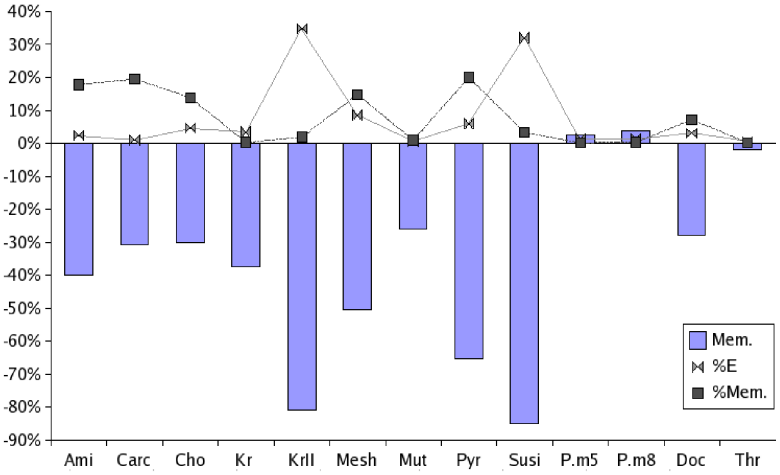


b. Memory analysis

Figure 13. Impact of tries in more detail.



a. Time analysis



b. Memory analysis

Figure 14. Impact of RL-trees in more detail.

Data Set	Tries		RL-Trees		Tries and RL-Trees	
	Time	Mem	Time	Mem	Time	Total Mem
Ami	+5%	-47%	-48%	-40%	-43%	-14%
Carc	+13%	-51%	-16%	-31%	-4%	-6%
Cho	+6%	-50%	-34%	-30%	-29%	-8%
Kr	+2%	-44%	-52%	-37%	-50%	-1%
KrII	0%	-40%	-91%	-81%	-91%	-12%
Mesh	+1%	-49%	-83%	-50%	-82%	-22%
Mut	0%	-50%	0%	-26%	0%	0%
Pyr	+1%	-53%	-80%	-65%	-79%	-32%
Susi	0%	-47%	-97%	-85%	-97%	-4%
P.m5	+11%	-51%	+4%	+3%	+10%	-4%
P.m8	+14%	-46%	+4%	+4%	+11%	-1%
Doc	+3%	-42%	-11%	-28%	-13%	-5%
Thr	+16%	-6%	0%	-2%	+18%	-5%

Table III. Impact of Tries and RL-trees alone and combined.

7.6. Tries and RL-Trees

We next evaluate the impact of using simultaneously tries and RL-trees. Table III shows the variation in April's execution time and total memory usage when using both data structures. The impact of using both data structures is clearly positive.

In general, all data sets show less memory usage, with an average reduction of 9% on April's total memory usage. This reduction is more significant in the data sets that required more memory, like ***Ami***, ***Carc***, ***Mesh*** and ***Pyr***. We should point out that, on average, the data stored in tries and RL-trees are, respectively, 11% and 13% of April's total memory. These weights increase as number of hypotheses generated and/or the number of examples of a data set also increases. Obviously, these weights should also vary from ILP system to ILP system.

Execution time was also significantly reduced, 35% on average, when using tries and RL-trees simultaneously. The only exceptions were observed in the the two artificial data sets, ***p.m5*** and ***p.m8***, and the ***Thr*** data set. This is closely related to the fact that the search space or/and the number of examples in these data sets are smaller than for others.

In summary, by using tries and RL-trees we are able to improve April's efficiency significantly, both in the execution time and memory usage.

7.7. Lazy Evaluation of Negatives with Tries and RL-Trees

At last, we show in Table IV the impact of using simultaneously both data structures, together with lazy evaluation of negatives, the globally best approach of the three presented. The table shows the variation in April's execution time and total memory usage.

Data Set	Time	Total Mem
Ami	-50%	-14%
Carc	-8%	-6%
Cho	-32%	-9%
Kr	-54%	-1%
KrII	-93%	-12%
Mesh	-82%	-22%
Mut	-1%	-1%
Pyr	-84%	-34%
Susi	-98%	-4%
p.m5	+6%	-9%
p.m8	+27%	-9%
Doc	-32%	-6%
Thr	+13%	-5%

Table IV. Impact of lazy evaluation of negatives with tries and RL-trees.

The results observed when combining the techniques are, in general, slightly better than the ones obtained using tries and RL-trees simultaneously.

7.8. Global Results

We conclude our study by presenting, in Table V, the impact of using all proposals: JIT indexing, lazy evaluation of negatives, tries and RL-trees. The time values are in seconds and memory values are in Kb.

The results show a speedup in all data sets, ranging from 1.85 to 600. From the previous analysis we may conclude that these results are a consequence, primarily, of the new indexing algorithm and, secondly, of the use of the proposed data structures together with lazy evaluation. The overall results in memory usage are not so good. In half of data sets there is a reduction in memory usage, substantial in some cases (like in ***Mesh***), while in the remaining data sets there is an increase. The increase was expected once we take into consideration the increase in memory usage that results of JIT indexing. However, the proposed data structures attenuated the increase, and in half of the cases, it was possible to reduce memory usage.

In short, the results demonstrate that our proposals significantly reduce the execution time by several orders of magnitude on all data sets. In terms of memory usage, there is a reduction in some cases and an increase in others. Thus, additional work should be done to further reduce memory consumption.

Data Set	Reference		All Together			
	Time	Mem	Time	Mem	Speedup	Mem (%)
Ami	4,628.92	20,593	19.58	19,272	236.45	-6.4
Carc	7,887.36	31,726	46.14	38,038	170.96	19.9
Cho	4,484.50	19,986	62.38	18,543	71.89	-7.2
Kr	0.62	2,791	0.24	3,084	2.63	10.5
KrII	39.77	5,807	2.99	4,874	13.29	-16.1
Mesh	5,740.76	35,425	30.73	12,822	186.83	-63.8
Mut	30,072.32	7,064	16,268.26	11,140	1.85	57.7
Pyr	3,731.91	18,080	20.47	12,287	182.31	-32.0
Susi	3,144.35	18,773	5.24	21,902	600.07	16.7
P.m5	159.16	36,739	11.43	59,891	13.93	63.0
P.m8	3,596.87	47,259	244.46	85,929	14.71	81.8
Doc	929.70	12,338	10.27	12,328	90.5	-0.1
Thr	3.26	5,228	1.31	5,482	2.49	4.9

Table V. Combined effect of all techniques.

8. Discussion and Conclusions

Inductive Logic Programming is a well established field of Machine Learning that addresses learning from structured and multi-relational data in a logical framework. The usefulness of ILP largely depends on its ability to scale up over larger and more complex applications. We study three strategies to improve the scalability of ILP: at the lower-level we address theorem proving performance in a very general and ILP independent fashion; at the mid-level we present novel data structures that allow current ILP algorithms to perform better; at the high-level, we propose alterations to the ILP algorithms themselves.

Each solution has advantadges and drawbacks. Prolog engines were developed for a very different style of programming, so optimizing ILP queries, in this case via indexing, brings substancial benefits at relatively little cost. On the other hand, improving a Prolog engine requires the ability to understand and improve the engine's implementation, a difficult task for most ILP implementors. At the opposite extreme, we can relax the ILP algorithm for better performance. This is the easiest solution for the ILP implementor, but the effects on system accuracy may be hard to gauge. In fact, the significant amount of previous work on ILP research makes it hard to propose novel solutions that are always effective. Our experiment with lazy evaluation illustrates this point: we do achieve significant benefits, but not always.

A different answer is to focus on data structures that better support current ILP algorithms. Tries are a pre-existing data structure that naturally fits the ILP search space. In practice, ILP systems do not spend much time searching this space, so time benefits are not impressive; but space usage can almost halve. The critical data structure is the open list: RL-trees address this

problem by improving the most expensive component of the coverage-list, and they achieve significant speedups and space compaction.

Overall, our results indicate that ILP performance can be substantially improved, mainly through understanding the main data structures involved in representing the search-space and by doing fast inference. Moreover, the results suggest a real need for collaboration between the ILP researchers and Prolog system designers, as applications grow ever larger and other points in the algorithm become bottlenecks. In a different vein, further opportunities for progress will stem from the parallelization of ILP systems [22, 32, 10, 16].

ACKNOWLEDGEMENTS

We are thankful to the anonymous referees for their valuable comments. This work has been partially supported by projects ILP-Web-Service (PTDC/EIA/70841/2006), JEDI (PTDC/EIA/66924/2006) and STAMPA (PTDC/EIA/67738/2006) and by Fundação para a Ciência e Tecnologia. Nuno Fonseca is funded by FCT grant SFRH/BPD/26737/2006.

REFERENCES

1. Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.
2. John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.
3. L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-Commutative Discrimination Nets. In *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74, Orsay, France, 1993. Springer-Verlag.
4. 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.
5. M. Botta, A. Giordana, L. Saitta, and M. Sebag. Relational learning: hard problems and phase transitions. In *Proc. of the 6th Congress AI*IA, LNAI 1792*, pages 178–189. Springer-Verlag, 1999.
6. R. Camacho. *Inducing Models of Human Control Skills using Machine Learning Algorithms*. PhD thesis, Department of Electrical Engineering and Computation, Universidade do Porto, 2000.
7. 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.
8. 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.
9. James Cussens. Part-of-speech disambiguation using ilp. Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, 1996.
10. 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.
11. Bart Demoen and Phuong-Lan Nguyen. So Many WAM Variations, So Little Time. In *LNAI 1861, Proceedings Computational Logic - CL 2000*, pages 1240–1254. Springer-Verlag, July 2000.
12. B. Dolsak, I. Bratko, and A. Jezernik. *Machine Learning, Data Mining and Knowledge Discovery: Methods and Applications*, chapter Application of machine learning in finite element computation. John Wiley and Sons, 1997.
13. Sašo Džeroski and Nada Lavrač, editors. *Relational Data Mining*. Springer-Verlag, September 2001.
14. Nuno A. Fonseca, Ricardo Rocha, Rui Camacho, and Fernando Silva. Efficient data structures for inductive logic programming. In *13th International Conference on Inductive Logic Programming*, Szeged, Hungary, 2003. Springer-Verlag.
15. Nuno A. Fonseca, Fernando Silva, and Rui Camacho. April - An Inductive Logic Programming System. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA06)*, volume 4160 of *Lecture Notes in Artificial Intelligence*, pages 481–484, Liverpool, September 2006. Springer-Verlag.

16. Nuno A. Fonseca, Fernando Silva, Vitor Santos Costa, and Rui Camacho. A pipelined data-parallel algorithm for ILP. In *Proceedings of 2005 IEEE International Conference on Cluster Computing*, pages 253–262, Boston, Massachusetts, USA, September 2005. IEEE.
17. E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.
18. Johannes Fürnkranz and Peter Flach. An analysis of rule evaluation metrics. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, Washington, 2003. Morgan Kaufmann.
19. P. Graf. Term Indexing. Number 1053 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
20. Stasinos Konstantopoulos, Rui Camacho, Nuno A. Fonseca, and Vitor Santos Costa. *Artificial Intelligence for Advanced Problem Solving Techniques*, chapter Induction as a Search Procedure. IGI Global, 2008.
21. N. Lavrač, P. Flach, and B. Zupan. Rule evaluation measures: A unifying view. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 174–185. Springer-Verlag, 1999.
22. Tohgoroh Matsui, Nobuhiro Inuzuka, Hirohisa Seki, and Hidenori Itoh. Parallel induction algorithms for large samples. In S. Arikawa and H. Motoda, editors, *Proceedings of the First International Conference on Discovery Science*, volume 1532 of *Lecture Notes in Artificial Intelligence*, pages 397–398. Springer-Verlag, December 1998.
23. W. W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
24. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
25. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
26. S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.
27. S. Muggleton, R.D. King, and M.J.E.Sternberg. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5(7):647–657, 1992.
28. 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.
29. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, February 1997.
30. S. Nijssen and J. N. Kok. Faster association rules for multiple relations. In *In* Proceedings of IJCAI 2001, pages 891–896, 2001.
31. H. J. Ohlbach. Abstraction Tree Indexing for Terms. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 479–484, Stockholm, Sweden, 1990. Pitman Publishing.
32. 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.
33. David Page and Ashwin Srinivasan. Ilp: A short look back and a longer look forward. *Journal of Machine Learning Research*, 4:415–430, 2003.
34. Doug Palmer and L. Naish. NUA-Prolog: an Extension to the WAM for Parallel Andorra. In Koishi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*. MIT Press, 1991.
35. 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.
36. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.
37. Bradley L. Richards and Raymond J. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
38. R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proceedings of the 2nd Conference on Tabulation in Parsing and Deduction*, pages 77–87, Vigo, Spain, 2000.
39. Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.
40. Hanan Samet. Data structures for quadtree approximation and compression. *Communications of the ACM*, 28(9):973–993, 1985.
41. V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 261–267, Paris,

France, 1999. Springer-Verlag.
42. V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*, 2000. Available from `http://www.ncc.up.pt/~vsc/Yap`.
43. V. Santos Costa, A. Srinivasan, and R. Camacho. A note on two simple transformations for improving the efficiency of an ILP system. 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 225–242. Springer-Verlag, 2000.
44. V. Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, , Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 2002.
45. Vítor Santos Costa, Kostis Sagonas, and Ricardo Lopes. Demand-driven indexing of prolog clauses. In Veronica Dahl and Ilkka Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 305–409. Springer, 2007.
46. Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I Compilation. *New Generation Computing*, 14(1):3–30, 1996.
47. M. Sebag and Rouveirol C. Tractable induction and classification in first order logic via stochastic matching. In *15th Int. Join Conf. on Artificial Intelligence (IJCAI'97)*, pages 888–893. Morgan Kaufmann, 1997.
48. A. Srinivasan. A study of two probabilistic methods for searching large spaces with ilp. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, 2000.
49. A. Srinivasan and R.C. Camacho. Numerical reasoning with an ILP program capable of lazy evaluation and customised search. *Journal of Logic Programming*, 40(2,3):185–214, 1999.
50. A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287. Springer-Verlag, 1997.
51. A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: Ilp experiments in a non-determinate biological domain. In S. Wrobel, editor, *4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 217–232, 1994.
52. Ashwin Srinivasan. Aleph manual, 2003.
53. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

APPENDIX A. Results

Data Set	Yap 4.4				Yap 4.5	
	|S|	Acc	Time	Mem	ΔTime	ΔMem
Ami	79,065	78.13	4,628.92	20,593	-4,590.05	+1,883
Carc	272,258	53.36	7,887.36	31,726	-7,836.98	+8,618
Cho	203,663	72.17	4,484.50	19,986	-4,392.35	+339
Kr	273	99.00	0.62	2,791	-0.11	+338
KrII	952	99.43	39.77	5,807	+0.05	-242
Mesh	27,901	80.76	5,740.76	35,425	-5,569.99	-18,936
Mut	42,624	82.45	30,072.32	7,064	-13,669.11	+4,150
Pyr	30,109	75.54	3,731.91	18,080	-3,605.35	+631
Susi	722	95.81	3,144.35	18,773	-2,924.57	+4,138
P.m5	35,763	99.49	159.16	36,739	-148.33	+29,210
P.m8	664,487	66.18	3,596.87	47,259	-3,404.08	+46,665
Doc	31,674	97.74	929.70	12,338	-914.66	+841
Thr	6,981	92.95	3.26	5,228	-2.10	+518

Table AI. Results obtained without JIT indexing (Yap 4.4) and with JIT indexing (Yap 4.5). Time is measured in seconds and memory in Kb. Accuracy did not change between Yap4.4 and Yap 4.5 runs.

Data Set	Negative			Positive			Total Laziness		
	$\mid S \mid$	Acc	Time	$\mid S \mid$	Acc	Time	$\mid S \mid$	Acc	Time
Ami	-1%	0%	-23%	20%	-1%	1%	3%	0%	1%
Carc	0%	0%	-19%	6%	7%	-22%	4%	4%	-28%
Cho	0%	0%	-19%	66%	0%	33%	-5%	0%	42%
Kr	-4%	0%	-11%	38%	0%	20%	0%	0%	13%
KrII	0%	0%	-32%	276%	0%	-32%	6%	0%	-35%
Mesh	0%	0%	-1%	30%	-4%	938%	3%	-1%	598%
Mut	8%	0%	13%	9%	1%	43%	9%	1%	-92%
Pyr	-2%	0%	-29%	11%	2%	-22%	-1%	0%	-45%
Susi	0%	0%	-52%	191%	0%	-55%	0%	0%	-53%
P.m5	0%	0%	-18%	39%	0%	59%	39%	0%	45%
P.m8	0%	0%	-30%	3%	0%	-45%	3%	0%	-58%
Doc	0%	0%	-41%	12%	0%	-46%	11%	0%	-54%
Thr	0%	0%	-7%	4162%	0%	58%	26%	0%	19%

Table AII. Impact of lazy evaluation without coverage caching compared to not using optimizations.