Efficient Data Structures for Inductive Logic Programming *

Nuno Fonseca[1], Ricardo Rocha[1], Rui Camacho[2], and Fernando Silva[1]

[1] DCC-FC & LIACC
Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
{nf, fds, ricroc}@ncc.up.pt
[2] Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
rcamacho@fe.up.pt

Abstract. This work aims at improving the scalability of memory usage in Inductive Logic Programming systems. In this context, we propose two efficient data structures: the Trie, used to represent lists and clauses; and the RL-Tree, a novel data structure used to represent the clauses coverage. We evaluate their performance in the April system using well known datasets. Initial results show a substantial reduction in memory usage without incurring extra execution time overheads. Our proposal is applicable in any ILP system.

Keywords: Implementation, Efficiency, Data Structures

1 Introduction

Inductive Logic Programming (ILP) [1,2] is an established and healthy subfield of Machine Learning. ILP has been successfully applied to problems in several application domains [3]. Nevertheless, it is recognized that efficiency and scalability is a major obstacle to the increase usage of ILP systems in complex applications with large hypotheses spaces.

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., [4,5]), or by efficiently testing candidate hypotheses (see, e.g., [6,7,8,9]). Another line of research, recommended by Page [10] and pursued by several researchers [11,12,13,14,15], is the parallelization of ILP systems. Another important issue is memory usage as a result of very large and complex search spaces. In this work, we develop techniques to considerably reduce the memory requirements of ILP systems without incurring in further execution time overheads. We propose and empirically evaluate two data structures that may be applied to any ILP system.

During execution, an ILP system generates many candidate hypotheses which have many similarities among them. Usually, these similarities tend to correspond to common prefixes among the hypotheses. Blockeel et al. [6] defined a new query-pack technique to exploit this pattern and improve the execution time of ILP systems. We propose the use of the Trie data structure (also known as prefix-trees) that inherently and efficiently exploits the similarities among the hypotheses to reduce memory usage. We also noted that systems like Aleph [16], Indlog [9], and April [17] use a considerable quantity of memory to represent clauses' coverage lists[3], i.e., lists of examples covered by an hypothesis. To deal with this issue, we propose a novel data structure, called RL-Tree, specially designed to efficiently store and manipulate coverage lists. An interesting observation is that the proposed data structures address the efficient representation of different types of data. Therefore, these data structures can be used in conjunction to maximize the gains in reducing memory usage by ILP systems.

The remainder of the paper is organized as follows. Sections 2 and 3 introduce the Trie and RL-Tree data structures and describe their implementation. In Section 4 we present an empirical evaluation of the impact in memory usage and execution time of the proposed data structures. Finally, in Section 5, we draw some conclusions and propose further work.

2 Tries

Tries were first proposed by Fredkin [18], the name coming from the central letters of the word *retrieval*. Tries were originally invented to index dictionaries, and has since been generalized to index terms (see [19] for use of tries in tabled logic programs and [20,21,22,23] for automated theorem proving and term rewriting).

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 lookup and possibly insertion to be performed in a single pass through a term.

2.1 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.

In ILP, the hypotheses space is structured as a lattice and hypotheses close to one another in the lattice have a lot of common structure. More specifically, hypotheses in the search space have common prefixes (literals). Not only the

[3] Camacho has observed that Indlog uses around 40% of total memory consumption to represent coverage lists.

hypotheses are similar, but information associated to them 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.

2.2 Description

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term in T. At the entry point 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 labeling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing symbol.

When inserting a new term, we start traversing the trie from the root node. Each child node specifies a symbol to be inspected in the input term when reaching that position. 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 1 presents an example for a trie with three terms.

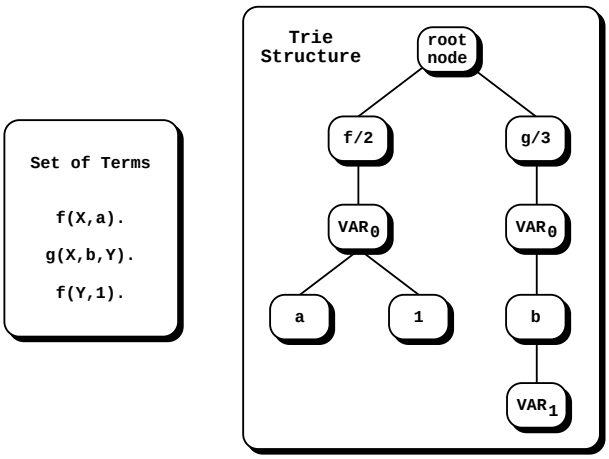

Fig. 1. Using tries to represent terms

An important point when using tries to represent terms is the treatment of variables. We follow the formalism proposed by Bachmair et al. [20], 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 $< \texttt{VAR}_0, \texttt{VAR}_1, ..., \texttt{VAR}_N >$, such that $\texttt{numbervar}_t(\texttt{X}) < \texttt{numbervar}_t(\texttt{Y})$ if X is encountered before Y in the left-to-right traversal of t. For example, in the term g(X, b, Y), numbervar(X) and numbervar(Y) are respectively $\texttt{VAR}_0$ and $\texttt{VAR}_1$. On the other hand, in terms f(X, a) and f(Y, 1), $numbervar(X)$ and $numbervar(Y)$ are both $\texttt{VAR}_0$. This is why the child node $\texttt{VAR}_0$ of f/2 from Fig. 1 is common to both terms.

2.3 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 forth field stores a pointer to the sibling node, in such a way that the outgoing transitions from a node are traced using its first child pointer and by following the list of sibling pointers of this child. Figure 2 illustrates the resulting implementation for the trie presented in Fig. 1.

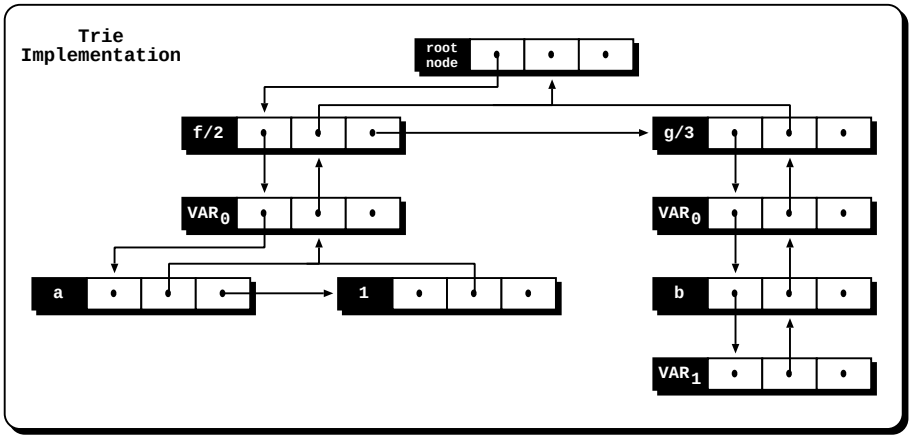

Fig. 2. The implementation of the trie in Fig. 1

At the entry point we have the root node. A root node is allocated when we call a open_trie(−R) predicate[4]. This predicate initializes a new trie structure and returns in R a reference to the root node of the new trie. As it is possible to have more than a trie structure simultaneously, when storing new terms we use R to specify the trie where the new terms should be inserted.

[4] We will use the − symbol to denote output arguments and the + symbol to denote input arguments.

New terms are stored using a `put_trie_entry`(+R, +T, −L) predicate. R is the root node of the trie to be used and T is the term to be inserted. The predicate returns in L the reference to the leaf node of the inserted term. For example, to obtain the structure in Fig. 2, the following code can be used:

```
open_trie(R).
put_trie_entry(R,f(X,a),L1).
put_trie_entry(R,g(X,b,Y),L2).
put_trie_entry(R,f(Y,1),L3).
```

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 is done sequentially. When the chain becomes larger then a threshold value (8 in our implementation), we dynamically index the nodes through a hash table to provide direct node access and therefore optimize the search. Further hash collisions are reduced by dynamically expanding the hash tables.

Recall that variables are standardized using the `numbervar` function. This standardization is performed while a term is being inserted in a trie. First occurrences of variables are replaced by binding the dereferenced variable cell to the constant returned by `numbervar`. Using this single binding, repeated occurrences of the same variable are automatically handled, without the need to check whether the variable has been previously encountered. The bindings are undone as soon as the insertion of the term is complete. In this manner, standardization is performed in a single pass through the input term.

To load a term from a trie, we have defined a `get_trie_entry`(+L, −T) predicate. This predicate returns in T the term whose leaf node is referred by L. For example, to obtain in T the term referred by L1, from the previous code, we should call `get_trie_entry`(L1, T). Starting from the leaf node L1 and following the parent pointers, the symbols in the path from the leaf to the root node are pushed into Prolog's term stack and the term is constructed. On reaching the root node, T is unified with the constructed term ($\mathtt{f}(\mathtt{VAR}_0, \mathtt{a})$).

When loading a term, the trie nodes for the term in hand are traversed in bottom-up order. 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 which limits the efficiency of alternative top-down loading schemes.

Space for a trie can be recovered by invoking a `close_trie`(+R) predicate, where R refers the root node of a particular trie, or by invoking a `close_all_tries`() predicate where all open tries are closed and their space recovered. Current implementation also defines auxiliary predicates to obtain memory consumption statistics and to print tries to the standard output. As a final note we should mention that besides the atoms, integers, variables and compound terms (func-

tors) presented in the examples, our implementation also supports terms with floats.

3 RL-Trees

The RL-Tree (**R**ange**L**ist-Tree) data structure is an adaptation of a generic data structure called quadtree [24] 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. 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 is designed to store integer intervals (e.g. $[1-3] \cup [10-200]$). The goals in the design of the RL-Tree data structure are: efficient data storage; fast insertion and removal; and fast retrieval.

3.1 Applicability

To reduce the time spent on computing clauses coverage some ILP systems, such as Aleph [16], FORTE [25], Indlog [9], and April [17], 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 $Cover(G) = \{all\ e \in E\ such\ that\ B \wedge G \vDash e\}$, where G is a clause, B the background knowledge, and E is the set of positive (E^+) or 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 of $Cover(G)$, thus reducing the coverage computation time. Cussens [26] extended this scheme by proposing a kind of 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 and Aleph are Prolog lists of integers. 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 the problems just mentioned: memory usage and execution time. The RL-Trees can be used to efficiently represent and manipulate coverages lists, and may be implemented in any ILP system (it is not restricted to ILP systems implemented in Prolog).

3.2 Description

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 types 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, and gray; only the root node can be completely black or white.

Consider the RL-Tree with domain [1,65], also denoted as RL-Tree(65). The figures 3, 4, 5, and 6 show some intervals represented in a RL-Tree(65). In these examples the LI and B parameters were set to 16 and 4 respectively. Figure 3 shows the representation of the interval [1]. Each group of four squares represents a range node. Each square in a range node corresponds to a subinterval. A 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 represent 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

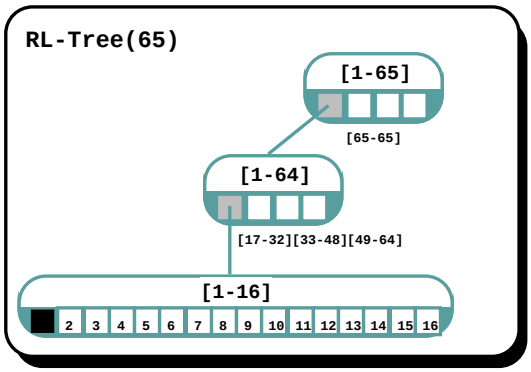

Fig. 3. Interval [1] represented in a RL-Tree(65)

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 4 shows the representation of a more complex list of intervals. Note that the number of nodes is the same as in Fig. 3 even though it represents a more complex list of intervals. Figure 5 and 6 show, respectively, a complete and empty interval representation.

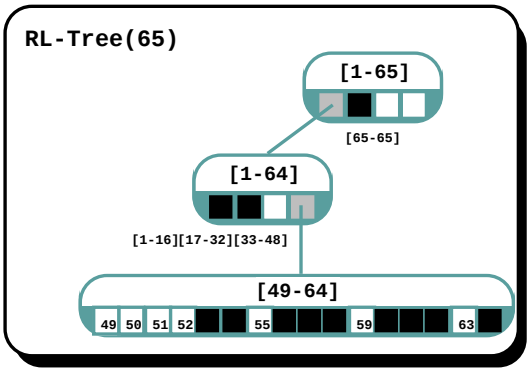

Fig. 4. Intervals $[1, 32] \cup [53, 54] \cup [56, 58] \cup [60, 62] \cup [64, 65]$ represented in a RL-Tree(65)

3.3 Implementation

Like the Trie data structure, 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.

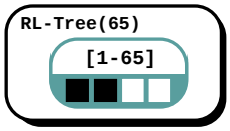

Fig. 5. Interval [1,65] represented in a RL-Tree(65)

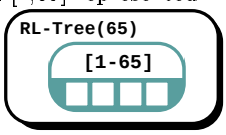

Fig. 6. Interval ∅ represented in a RL-Tree(65)

Like other quadtree data structures [27], a RL-Tree can be implemented with or without pointers. We chose to do a pointerless implementation (using an array) to reduce memory consumption in pointers. 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, or 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 need to traverse the tree to determine the position, in the array, of a given node. The list nodes use 16 bits. Each bit represents a number (that in turn represents an example). The number interval represented by a list node is inferred from its parent range node.

The RL-Tree(N) implemented operations 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: $O(H)$

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 $C = 20$.

4 Experiments and Results

The goal of the experiments was to evaluate the impact of the proposed data structures in the execution time and memory usage when dealing with real application problems.

We adapted the April ILP system [17] so that it could be executed with support for Tries and/or RL-Trees and applied the system to well known datasets. For each dataset the system was executed four times with the following configuration: no Tries and no RL-Trees, Tries and RL-Trees, Tries only, and RL-Trees only.

4.1 Experimental Settings

The experiments were made on an AMD Athlon(tm) MP 2000+ dual-processor PC with 2 GB of memory, running the Linux RedHat (kernel 2.4.20) operating system. We used version 0.5 of the April system and version 4.3.23 of the YAP Prolog [28].

The datasets used were downloaded from the Machine Learning repositories of the Universities of Oxford[5] and York[6]. The *susi* dataset was downloaded from the Science University of Tokyo[7]. Table 1 characterizes the datasets in terms of number of positive and negative examples as well as background knowledge size. Furthermore, it shows the April settings used with each dataset. The parameter *nodes* specifies an upper bound on the number of hypotheses generated during the search for an acceptable hypothesis. The i-depth corresponds to the maximum depth of a literal with respect to the head literal of the hypothesis [29]. *Sample* defines the number of examples used to induce a clause. *Language* parameter specifies the maximum number of occurrences of a predicate symbol in an hypothesis [9]. *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.

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 the dataset. However, since we are comparing the memory consumption when using a data structure with when not using it, the estimate we obtain will still give a good idea of the impact of the data structure in reducing memory usage.

4.2 Tries

When activated in April, the Trie data structure stores 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's head, and a list of free variables in the clause.

Table 2 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 the values obtained when using and

[5] `http://www.comlab.ox.ac.uk/oucl/groups/machlearn/`
[6] `http://www.cs.york.ac.uk/mlg/index.html`
[7] `http://www.ia.noda.sut.ac.jp/ilp`

Dataset	Characterization			April's Settings					
	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$	nodes	i	sample	language	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
susi	252	8979	18	no limit	5	2	1	200	800

Table 1. Settings used in the experiments

when 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, showing a degradation of 25% in memory consumption. This may indicate that the Tries data structure is not adequate for datasets with very small hypothesis spaces. However, memory usage is not a concern for problems with small hypotheses space.

Dataset	$\mid H \mid$	Time (sec.)		Memory (bytes)		on/off(%)	
		off	on	off	on	Time	Memory
amine uptake	66933	357.10	362.40	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**
susi	3344	7995.01	7982.82	3655916	1934640	**99.84**	**52.91**

Table 2. The impact of Tries

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 Tries data structure reduce memory consumption with a minimal execution time overhead.

4.3 RL-Trees

Table 3 presents the impact of using RL-Trees in the April system. It 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 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 (Kb)		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**
susi	3343	8079.51	7927.03	3021356	263098	**98.11**	**8.70**

Table 3. 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 significant reductions in memory usage were obtained with datasets with greater number of examples.

In general, a considerable 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 was obtained.

4.4 Tries and RL-Trees

To evaluate the impact of using Tries and RL-Trees simultaneously we ran April configured to use both data structures. The Table 4 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 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 the 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**
susi	3343	8079.51	7928.98	6677264	2197050	**98.13**	**32.90**

Table 4. The impact of Tries and RL-Trees

The use of both data structures resulted in significant reductions in memory usage. The results indicate that the impact of the data structures tend to be greater in the applications with more examples and with larger search spaces. The *train* was the only dataset that consumed more memory when using Tries and 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 proposed data structures. The time overhead experienced is minimum in the smaller datasets (*train, train128, multiplication*), and non existent in the larger datasets.

Table 5 resumes 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 significantly.

5 Conclusions

This paper contributes to the effort of improving ILP systems efficiency by proposing two data structures: RL-Trees and Tries. The use of these data struc-

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
susi	20	16.66

Table 5. April memory consumption using Tries and RL-Trees

tures reduce memory consumption without an execution time overhead. The RL-Tree is a novel data structure designed to efficiently store and manipulate coverage lists. The Trie data structure inherently and efficiently exploit the similarities among the candidate hypotheses generated by ILP systems to reduce memory usage. The data structures were integrated in the April system, an ILP system implemented in Prolog.

We have empirically evaluated the use of RL-Trees and Tries, both individually and in conjunction, on well known datasets. The RL-Tree data structure alone reduced memory usage in coverage lists to half, in average, and slightly reduced the execution time. The Tries data structure alone reduced memory consumption with a minor overhead (approximately 1%) in 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 and 42%) when using Tries and RL-Trees simultaneously. The results indicate that the benefits from using these data structures tend to increase for datasets with larger search spaces and greater number of examples. Since the data structures are system independent, we believe that they can be also applied to other ILP systems with positive impact.

In the future we plan to implement operations like intersection and subtraction of two RL-Trees in order to compute the coverage intersection of two clauses more efficiently. We also will identify more items collected during the search that may take advantage of the Tries data structure.

Acknowledgments

The authors thank Vitor Santos Costa for making YAP available and for all the support provided. 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. Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.
4. 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.
5. 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.
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. 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.
8. Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 2002.
9. Rui Camacho. *Inductive Logic Programming to Induce Controllers*. PhD thesis, Univerity of Porto, 2000.
10. David Page. ILP: Just do it. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 3–18. Springer-Verlag, 2000.
11. 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.
12. 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.
13. 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.
14. Y. Wang and D. Skillicorn. Parallel inductive logic for data mining. In *In Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*, Boston, 2000. ACM Press.
15. 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.

16. Aleph. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.
17. Nuno Fonseca, Rui Camacho, Fernando Silva, and Vítor Santos Costa. Induction with April: A preliminary report. Technical Report DCC-2003-02, DCC-FC, Universidade do Porto, 2003.
18. E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.
19. 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.
20. 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 Lecture Notes in Computer Science, pages 61–74, Orsay, France, 1993. Springer-Verlag.
21. P. Graf. Term Indexing. Number 1053 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
22. W. W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
23. 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.
24. Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.
25. Bradley L. Richards and Raymond J. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
26. James Cussens. Part-of-speech disambiguation using ilp. Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, 1996.
27. Hanan Samet. Data structures for quadtree approximation and compression. *Communications of the ACM*, 28(9):973–993, 1985.
28. Vítor Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP Prolog User's Manual*. Universidade do Porto, 1989.
29. S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.