ILP :- Just Trie It

Rui Camacho[1], Nuno A. Fonseca[2], Ricardo Rocha[3], and Vítor Santos Costa[3]

[1] Faculdade de Engenharia & LIAAD
Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
rcamacho@fe.up.pt
[2] IBMC & LIACC
Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
nf@ibmc.up.pt
[3] DCC-FC & LIACC
Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
{ricroc, vsc}@ncc.up.pt

Abstract. Despite the considerable success of Inductive Logic Programming, deployed ILP systems still have efficiency problems when applied to complex problems. Several techniques have been proposed to address the efficiency issue. Such proposals include query transformations, query packs, lazy evaluation and parallel execution of ILP systems, to mention just a few.
We propose a novel technique to improve the execution time of an ILP system that avoids the procedure of *deducing* each example to evaluate each constructed clause. The technique takes advantage of the two stage procedure of Mode Directed Inverse Entailment (MDIE) systems. In the first stage of a MDIE system, where the bottom clause is constructed, we store not only the bottom clause but also valuable additional information. The information stored is sufficient to evaluate the clauses constructed in the second stage without the need for a theorem prover. We used a data structure called Trie to efficiently store all bottom clauses produced using all examples (positive and negative) as seeds.
The technique was implemented and evaluated in two well known data sets from the ILP literature. The results are promising both in terms of execution time and accuracy.

Keywords: Mode Directed Inverse Entailment, Efficiency, Data Structures

1 Introduction

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

Research in improving the efficiency of ILP systems has focused on 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. A survey on parallel execution of ILP systems can be found in [30].

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 query-packs as a technique to exploit this pattern and improve the execution time of ILP systems. Inspired by their work, we focus on how to reduce the amount of theorem proving to a minimum. We call our novel approach *trieing MDIE*. The key idea is to use a single *Trie* data structure (also known as a *Prefix-Tree*) to inherently and efficiently exploit the similarities among the hypotheses, hence reducing memory usage and allowing us to store useful information about causes. But this is as close we get to query-packs, which can be considered as a form of trie designed to improve execution speed. Instead we follow a different approach based on Mode Directed Inverse Entailment (MDIE)[32].

To explain our approach, *trieing MDIE*, let us recall that a "traditional" MDIE-based procedure is performed in two stages. In the first stage an example is chosen and the bottom clause[32] is constructed (saturation stage). In the second stage a search is performed using the bottom clause as the lower bound of the search space. During the second stage clauses are constructed and evaluated using the examples. In the *trieing MDIE* approach we saturate all examples (positive and negative). From each bottom clause we generate valid clauses and insert them in an *unique* Trie, such that the Trie contains counters describing clause coverage. The search procedure of the second stage of a "traditional" MDIE approach will therefore be replaced by a simple inspection of this Trie to retrieve the best clause. For efficiency sake, in our approach we first process the positive examples so that only clauses generated by the positives are inserted in the Trie. The clauses generated using the negatives are not considered interesting, so negative clauses just update coverage counters.

Trieing MDIE is usable in positive only data sets and is not applicable to learn recursive theories. A further improvement in speedup can be achieved by combining *Trieing MDIE* with some of the strategies to parallelise ILP [30], such as *Parallel Exploration of Independent Hypotheses* and; *Data Parallelism.* It can be implemented in MDIE-based ILP systems such as Progol [29], Aleph [16], Indlog [9], and April [33]. Notice that tries had already been proposed previously ([31]) as a technique to reduce the amount of memory storage. In that study Tries were used to store the clauses constructed during the second stage of the MDIE method. They have also been used in the FARMER system [35] to overcome efficiency issues of the Warmr system [34] for learning Association Rules.

The remainder of the paper is organised as follows. Section 2 introduces the Trie data structure and describes its implementation. In Section 3 we present the algorithm to use Tries in MDIE-based ILP systems. Section 4 presents some

limitations of the technique when using background knowledge containing predicates that are not pure Logic Programs. In Section 5 we present an empirical evaluation of the impact in execution time of the proposed data structure. Finally, in Section 6, we draw some conclusions and propose further work.

2 The Trie Data Structure

Tries were first proposed by Fredkin [17], the original name inspired by the central letters of the word *retrieval*. Tries were originally invented to index dictionaries, and have since been generalised to index recursive data structures such as terms. Please refer to [22,21,19,20,18] for the use of tries in automated theorem proving, term rewriting and tabled logic programs. An essential property of the trie data structure is that common prefixes are represented only once. The efficiency and memory consumption of a particular trie thus largely depends on the percentage of terms that have common prefixes. This naturally applies to ILP, as the hypotheses space is structured as a lattice and hypotheses close to one another in the lattice have common prefixes (literals). Hence, it clearly matches the common prefix property of tries. We thus argue that, for ILP systems, this is an interesting property that we should be able to take advantage of for storing hypotheses and associated information.

2.1 Description

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. At the entry point we have the root node. Internal nodes 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. When inserting a new term, we start 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 node. On reaching the last symbol in the input term, we reach a leaf node in the trie.

An important point when using tries to represent terms is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [19], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR_0, ..., VAR_N$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t. For example, in the term $[eastbound(T), has_car(T, C), long(C)]$, $numbervar(T)$ and $numbervar(C)$ are respectively VAR_0 and VAR_1.

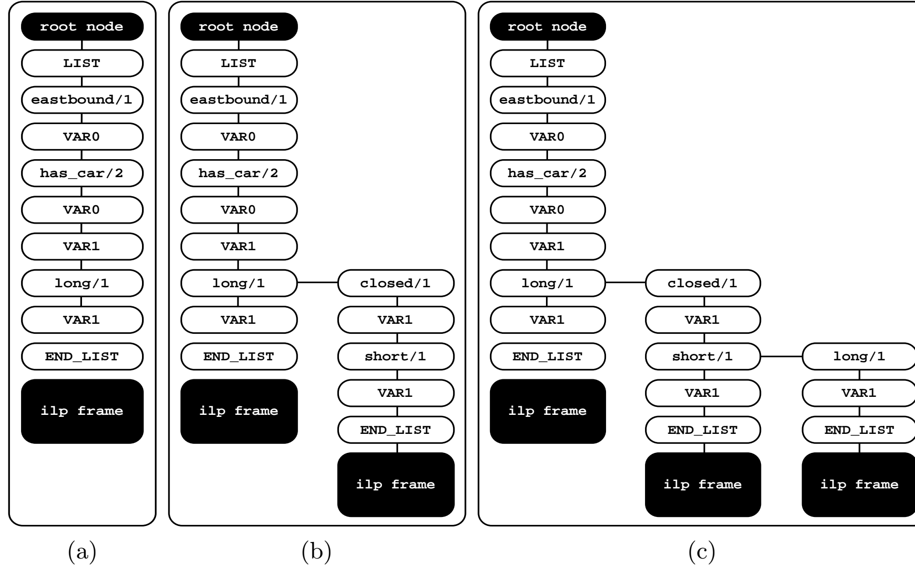

Fig. 1. Using tries to represent:
(a) C = eastbound(Train) :- has_car(Train,Car),long(Car);
(b) C and D = eastbound(Train) :- has_car(Train,Car),closed(Car),short(Car);
(c) C, D and E = eastbound(Train) :- has_car(Train,Car),closed(Car),long(Car).

2.2 Using Tries to Represent Hypotheses

In order to maximise the number of common trie nodes when storing hypotheses in a trie, we used Prolog lists to represent the clauses corresponding to hypotheses. A clause of the form $Head : -Body_1, ..., Body_n$ is stored in the trie structure as an unique path corresponding to the list $[Head, Body_1, ..., Body_n]$. Such a path always starts from the root node in the trie, follows a sequence of trie nodes and terminates at a leaf data structure, the *ilp frame* data structure, that we used to extend the original trie structure to store associated info with the clause, namely info regarding the number of positive and negative examples covered by the clause (the use of this info is discussed in more detail in the following sections). Figure 1 presents an example for a trie with three clauses.

Initially, the trie contains the root node only. Next, we insert the clause $[eastbound(T), has_car(T, C), long(C)]$ and nine nodes are added to represent it (Figure 1(a)). The clause $[eastbound(T), has_car(T, C), closed(C), short(C)]$ is then inserted which requires eleven nodes. As it shares a common prefix with the previous clause, we save the six initial nodes common to both representations (Figure 1(b)). The clause $[eastbound(T), has_car(T, C), closed(C), long(C)]$ is next inserted and we save more eight nodes, the same six nodes as before plus two more nodes common with the second inserted clause (Figure 1(c)).

3 Trieing MDIE

The first insight in our approach is that if a legal clause C covers an example e then an instance of the clause $C\sigma$ must be in the most specific clause for e, $\bot_e$.

Indeed, imagine that C covers e. Then, there is a θ such that $C\theta$ satisfies example e. Hence, from the completeness of saturation, a variant of $C\theta$ must be in e's most specific clause. Hence, there must be a σ such that $C\sigma$ is in e's most specific clause.

If this is the case, can we just look for the clause in e's bottom clause? Unfortunately, the answer is not quite: generalising $C\theta$ will not always lead to a *variant* of the clause of interest, C. To understand the problem, consider the following clause C:

$$C \;=\; l(A) \leftarrow h_c(A,B), h_c(A,C), d(C), o_c(B).$$

and the following bottom-clause for an example e:

$$\bot_e \;=\; l(A) \leftarrow h_c(A,B), h_c(A,C), o_c(B), d(B), f(C).$$

Careful examination shows that $\bot_e$ is entailed by clause C. On the other hand, the closest clause C' that can be generated from the bottom-clause is:

$$C' \;=\; l(A) \leftarrow h_c(A,B), h_c(A,C), d(B), o_c(B).$$

Although $C' = C\theta$, C' is a more *specific* version of the original clause, it is not a variant. In this case, we cannot find a variant, even though the example indeed covers the clause.

This suggests the following approach: given an example e construct the corresponding bottom clause $\bot_e$. Next, generate a set $\mathcal{C}$ with all legal clauses C such that C θ-subsumes $\bot_e$. Then, we can state the following theorem:

Theorem 1. *A clause covers e iff it is in* $\mathcal{C}$

If a clause covers e then according to the completeness of the bottom-clauses it must θ-subsume the bottom-clause. The reverse immediately follows from the properties of the bottom-clause. Therefore we have a set $\mathcal{C}$ with all legal clauses that cover the example.

Next, given a set of examples $\{e_1^+, e_2^+, \ldots, e_n^+\, e_1^-, e_2^-, \ldots, e_m^-\}$ construct the corresponding sets of clauses $\{\mathcal{C}_1^+, \mathcal{C}_2^+, \ldots, \mathcal{C}_n^+\, \mathcal{C}_1^-, \mathcal{C}_2^-, \ldots, \mathcal{C}_m^-\}$: finding the best clauses should be just a question of searching for clauses that appear in most $\mathcal{C}_i^+$ and not in $\mathcal{C}_i^-$. More precisely, if we allow no noise, then we would like to find the clause with best coverage from $\cup_i \mathcal{C}_i^+ \setminus \cup_j \mathcal{C}_j^-$.

The second insight of our approach is that we are not interested in examples, but in the set of all clauses of interest, $\mathcal{C}$ (which would to a first approximation be close to $\cup_i \mathcal{C}_i^+$. Now, this set may grow quickly, and therefore needs a compact and fast representation. It makes sense to represent sets of clauses by structures optimised for quick access and sharing, such as the *tries* discussed previously.

Assuming this representation works, one approach is just to walk over all examples, generate all clauses subsuming the bottom-clause, and:

– If $C \in \mathcal{C}$, somehow state that C covers this example.
– If $C \notin \mathcal{C}$, add C to $\mathcal{C}$ and state that C covers this example.

This basic algorithm can be optimised if we visit positive examples first, and assume we care not about clauses that only cover negative examples:

– If the example is positive and $C \in \mathcal{C}$, we state that C covers this positive example.
– If the example is positive and $C \notin \mathcal{C}$, we add C to $\mathcal{C}$ and state that C covers one positive example.
– If the example is negative and $C \in \mathcal{C}$, we state that C covers this negative example.
– If the example is negative and $C \notin \mathcal{C}$, do nothing.

Given a set $\mathcal{C}$ with all clauses and their coverage, we can easily implement any theory construction algorithm.

The algorithm

The *Trieing MDIE* approach uses a greedy coverage main cycle as shown in Figure 2. The main difference with systems like Progol or Aleph concern the inner procedure *learn_MDIE_Trie()*. We now explain how clauses are being learn in the *trieing MDIE* approach.

```
generalise_MDIE_Trie(B,E,C):
  Given:
  background knowledge B;
  a finite training set E = E+ ∪ E−;
  a set of constraints C;
  Return: a hypothesis H that explains the E and satisfies C

  H = ∅
  while E+ ≠ ∅ do
     h = learn_MDIE_Trie(E+, E−, B, C)
     E+ = E+ \ covered(h)
     H = H ∪ h
     B = B ∪ h
  endwhile
  return H
```

Fig. 2. The greedy cover algorithm of a MDIE system implementation.

In what follows we refer the reader to Figure 3. The *Trieing MDIE* algorithm has two basic stages. First a Trie is constructed (lines 1-19) and then the best clause is found by inspection of the Trie (line 20). In the first stage the Trie is constructed in three steps. We first use each positive example (lines 2-10) and

```
learn_MDIE_Trie(E+, E−, B, C):
   Given:
   background knowledge B; a finite training set E = E+ ∪ E−; constraints C;
   Return: the best hypothesis that explains some of the E+ and satisfies C.

1.   trie = ∅
2.   foreach e ∈ E+ do
3.       bot = Saturate(e, B, C)
4.       repeat
5.           clause = FindNewValidClause(bot)
6.           clause = Normalise(clause)
7.           if NotInTrie(clause, trie) then trie = InsertInTrie(clause, trie)
8.           trie = UpdatePosCounters(clause, trie)
9.       until clause == ∅
10.  endforeach
11.  trie = PruneTrie(trie)
12.  foreach e ∈ E− do
13.      bot = Saturate(e, B, C)
14.      repeat
15.          clause = FindNewValidClause(bot)
16.          clause = Normalise(clause)
17.          trie = UpdateNegCounters(clause, trie)
18.      until clause == ∅
19.  endforeach
20.  return bestClauseInTrie(trie)
```

Fig. 3. The learning algorithm of Trieing MDIE.

for each we generate a bottom clause. Using the bottom clause we generate all valid clauses[4] (lines 4-9) and insert them in the Trie (line 7). To avoid syntactic redundancy we normalise each clause (line 6). Normalisation orders the literals according to the Prolog "@ <" order relation. The user has no choice on the ordering to use. Since variables are turned into constants in the Trie ($VAR_0, \ldots VAR_N$) we have to generate all renaming of existential variables to get the right countings in the Trie. All generate valid clauses update the counters in the Trie (line 8). After the Trie is constructed we call a pruning procedure (PruneTrie() line 11) that removes useless nodes like nodes with countings less than minimum number of examples covered by a clause to be accepted (mincover parameter). In the third step we process all the negative examples. With each negative example we construct a bottom clause and from that bottom clause construct valid clauses. Those clauses are discarded if they are not in the Trie or used to update the counters if there is a copy of them.

The last stage of the algorithm (line 20 - bestClauseInTrie()) returns the best clause stored in the Trie.

4 Trie the Real World

We have presented our algorithm in the context of an ideal world, where the Background Knowledge is a pure logic program, the saturated clause is generated to its completion, and all clauses subsuming the saturated clause are enumerated. Next we discuss how our algorithm can cope with two major issues we found in practise: completion of the saturated clause and syntactic redundancy.

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

The i constraint is a syntactic constraint on the maximum length of variable chains. This constraint is applied uniformly to every goal while generating the bottom-clause. By induction, it should be clear that if a variable chain respects the i constraint in a saturated clause, it will respect the same constraint on every other saturated clause.

The recall-number parameter indicates how many solutions to a goal can be introduced in the bottom clause. If set to *, it will include every answer. On the other hand, if set to a lower threshold than the actual number of different answers a goal can generate, this parameter becomes a source of incompleteness. As the answer order will be different with different examples, using low-values of this parameter is not recommended when using the proposed algorithm.

Syntactically Redundant Clauses The *switching lemma* tells us that if conjunction of goals $G_1, \ldots, G_n$ is satisfiable, then any permutation of these goals is also satisfiable. ILP systems often take advantage of this principle to reduce the

[4] Clauses satisfying the language and bias constraints.

number of clauses they actually need to generate: if one generates $a(X), b(X)$ there is no point in also generating $b(X), a(X)$.

On the other hand, traditional ILP systems cannot use any ordering of goals, as they must respect an ordering that is efficient for Prolog execution. As our algorithm does not actually evaluate goals, this is is unnecessary: we can choose any ordering between goals when checking for redundant goals. In this vein, we try to simplify all syntactically redundant clauses into a normalised clauses, so that all syntactically equivalent clauses will have a canonical representative in the trie.

5 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 [33] so that it could be executed with support for Tries and applied the system to well known data sets. For each data set the system was executed twice with the following configuration: standard MDIE implementation (no Tries), and MDIE using Tries.

5.1 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.

The data sets used were downloaded from the Machine Learning repositories of the Universities of Oxford[5] and York[6]. Table 5.1 characterises the data sets in terms of number of positive and negative examples as well as background knowledge size.

Data set	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$
carcinogenesis	202	174	44
mutagenesis	136	69	21

Table 1. Data sets

5.2 Results

For each data set, we ran a vanilla ILP system to generate a theory using a deterministic top-down breadth-first search procedure (*dtd-bf*). We varied the maximum depth of the clauses ($\mathcal{S}$) from 2 (one literal in the body) to 4 (3 literals in the body).

[5] `http://www.comlab.ox.ac.uk/oucl/groups/machlearn/`

[6] `http://www.cs.york.ac.uk/mlg/index.html`

Data set	$\mathcal{S}$	Execution Time		Clauses generated	
		dtd-bf MDIE	**Trieing MDIE**	**dtd-bf MDIE**	**Trieing MDIE**
Carcinogenesis	2	4	6	8,012	17,352
Carcinogenesis	3	56	82	233,860	684,855
Carcinogenesis	4	2,205	4,049	5,827,459	26,613,734
Mutagenesis	2	2,130	3,442	8,991	18,308
Mutagenesis	3	13,809	5,343	339,591	834,023
Mutagenesis	4	21,600	7,115	9,261,589	20,445,957

Table 2. Execution time and number of clauses generated

Table 2 compares the execution times and the number of clauses generated by both approaches. The results confirm that Trieing MDIE generates considerably more clauses (ranging from two up to five fold) than the other approach. In spite of considering more clauses in the search, Trieing MDIE outperforms dtd-bf MDIE in the mutagenesis data set. However, it is around 50% slower than dtd-bf MDIE in the carcinogenesis data set. Naturally, if the same number of clauses is generated, Trieing MDIE reduces the execution time. Although the impact in execution time of Trieing MDIE is inconclusive, the impact in accuracy is promising. In Table 3 we can observe that Trieing MDIE achieves good results. In terms of memory usage Trieing MDIE is efficient as shown in Table 4.

Data set	$\mathcal{S}$	dtd-bf MDIE	Trieing MDIE
Carcinogenesis	2	72	72
Carcinogenesis	3	48	62
Carcinogenesis	4	51	69
Mutagenesis	2	65	65
Mutagenesis	3	71	94
Mutagenesis	4	74	82

Table 3. Accuracy

Data set	$\mathcal{S}$	dtd-bf MDIE	Trieing MDIE
Carcinogenesis	2	19	19
Carcinogenesis	3	19	21
Carcinogenesis	4	122	59
Mutagenesis	2	10	13
Mutagenesis	3	19	13
Mutagenesis	4	99	22

Table 4. Memory Usage

6 Conclusions

This paper is a novel contribution to the effort of improving ILP systems efficiency. A novel technique was put forward to reduce execution time of MDIE-based ILP systems. This improvement is achieved by avoiding the theorem proving of all clauses constructed during the search stage of a MDIE system. This was possible by using a Trie data structure to store all generated clauses, and their coverage. Tries take advantage of common pre-fixes in clauses which leads to a quite small memory requirements for the ILP system. Coverage information allows the system to estimate efficiently the value of clauses.

The proposed technique was integrated in an ILP system implemented in Prolog and empirically evaluated on two well known data sets. The results indicate a significant reduction in execution time (for the same number of clauses evaluated) in all data sets used. The results also indicate an increase in accuracy since the system is performs wider searches. Overall the amount of memory used to analyse the data sets was very small.

In the future we plan to extend the evaluation process. We will first determine the degree of non-determinism of the background knowledge of each data set. We expect the result to improve with an increase of non-determinism of the predicates in the background knowledge (more effort in theorem proving).

To show further the advantage in memory savings we intend to use much larger data sets. Data from the ILP challenge from 2005, for example, will be considered.

Acknowledgements

Nuno Fonseca is funded by the FCT grant SFRH / BPD / 26737 / 2005. This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

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. E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.
18. 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.
19. 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.
20. P. Graf. Term Indexing. Number 1053 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
21. W. W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.
22. 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.
23. Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

24. Bradley L. Richards and Raymond J. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
25. James Cussens. Part-of-speech disambiguation using ilp. Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, 1996.
26. Hanan Samet. Data structures for quadtree approximation and compression. *Communications of the ACM*, 28(9):973–993, 1985.
27. Vítor Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP Prolog User's Manual*. Universidade do Porto, 1989.
28. S. Muggleton and C. Feng. *Efficient induction in logic programs*, In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.
29. S. Muggleton, *Inverse Entailment and Progol*, New Generation Computing, Special issue on Inductive Logic Programming. 245-286, vol 13, N. 3-4, 1995.
30. Nuno A. Fonseca and Fernando Silva and Rui Camacho, *Strategies to Parallelize ILP Systems*, Proceedings of the 15th International Conference on Inductive Logic Programming (ILP 2005), Lecture Notes in Artificial Intelligence, vol 3625, pp 136–153, 2005.
31. Nuno A. Fonseca and Ricardo Rocha and Rui Camacho and Fernando Silva *Efficient Data Structures for Inductive Logic Programming*, Proceedings of the 13th International Conference on Inductive Logic Programming Lecture Notes in Artificial Intelligence, vol 2835, edts T. Horváth and A. Yamamoto, pp 130–145, 2003.
32. S. Muggleton *Inverse Entailment and Progol*, New Generation Computing, Special issue on Inductive Logic Programming", vol 13, N. 3-4, pp 245-286, 1995.
33. Nuno A. Fonseca and Fernando Silva and Rui Camacho *April - An Inductive Logic Programming System*, Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA06), Lecture Notes in Artificial Intelligence, Springer-Verlag, pp. 481-484, vol 4160, 2006.
34. L. Dehaspe and L. De Raedt *Mining Association Rules in Multiple Relations*, Proceedings of the 7th International Workshop on Inductive Logic Programming, LNAI 1297, pp 125–132, 1997
35. Siegfried Nijssen and Joost N. Kok *Faster Association Rules for Multiple Relations*, Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01), pp. 891-896. 2001