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, Portugal
rcamacho@fe.up.pt
[2] Instituto de Biologia Molecular e Celular (IBMC), Universidade do Porto, Portugal
nf@ibmc.up.pt
[3] DCC-FC, Universidade do Porto, Portugal
{ricroc,vsc}@ncc.up.pt

Abstract. Despite the considerable success of Inductive Logic Programming (ILP), 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 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 using 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] has been successfully applied to problems in several application domains [2]. Nevertheless, it is recognised that efficiency and scalability are major obstacles to the increased usage of ILP systems in complex applications with large hypothesis spaces. Research on 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., [3,4]), or by efficiently testing candidate hypotheses (see, e.g., [5,6,7,8]). Another line of research pursued by several researchers is the parallelization of ILP systems [9].

During execution, an ILP system generates many candidate hypotheses which have a lot of similarities among them. Usually, these similarities tend to correspond to common prefixes among the hypotheses. Blockeel et al. [5] 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 clauses. 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)[10].

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 [10] 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 will therefore be replaced by a simple inspection of this trie to retrieve the best clause.

Trieing MDIE can be implemented in MDIE-based ILP systems such as Progol [11], Aleph [12], Indlog [8], and April [13]. It 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 known strategies to parallelise ILP systems [9], such as, *parallel exploration of independent hypotheses* and *data parallelism*. Notice that tries have already been proposed previously [14] 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 FARMER [15] to overcome efficiency issues of the Warmr system [16] 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 a 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 [18,19,20] 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. This

naturally applies to ILP, as the hypothesis 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.

Using Tries to Represent Hypotheses

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. 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. Figure 1 presents an example for a trie with three clauses.

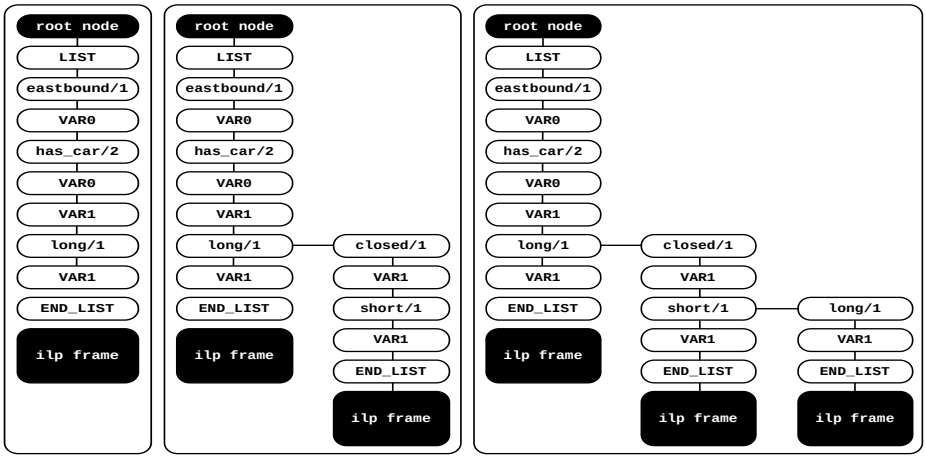

Fig. 1. Using tries to represent:
(a) *C = eastbound(T) :- has_car(T,C), long(C).*
(b) *C and D = eastbound(T) :- has_car(T,C), closed(C), short(C).*
(c) *C, D and E = eastbound(T) :- has_car(T,C), closed(C), long(C).*

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)).

Each path in the trie terminates at a leaf data structure, the *ilp frame* data structure, that we used to extend the original trie structure to store information associated with the clause, namely info concerning the number of positive and negative examples covered by the clause (the use of this information is discussed in more detail next). Another important point when using tries to represent clauses is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [18], 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.

3 Trieing MDIE

MDIE-based systems use bottom-clauses to generate sets of clauses. Given a bottom-clause $\bot_e$, the refinement operator generates clauses from $\bot_e$ that will cover at least the example e. Let us call this set $\mathcal{S}$. The clauses in $\mathcal{S}$ share e, so we can say that e *forms* $\mathcal{S}$. Note that, in general, $\mathcal{S}$ will be arbitrarily large, and we will need to impose some further restrictions, such as clause length restrictions. Moreover, note that even if complete, $\mathcal{S}$ does not correspond to all clauses that cover e. Indeed, it is well known that the bottom-clause is not complete: we can generate clauses that cover an example e which cannot be refined from $\bot_e$ [21].

Still, it is interesting to try to understand the meaning of $\mathcal{S}$. An important question in this regard is: if a clause c generated for example e covers example x, will c or, to be more precise, a variant of c, be in x's bottom clause, $\bot_x$? We would expect this to be true for ground clauses. Indeed, if this was not the case there must be at least a ground clause $h \leftarrow g_1, \ldots, g_{i-1}, g_i$ not refined from $\bot_x$, such that $h \leftarrow g_1, \ldots, g_{i-1}$ can be refined from $\bot_x$. Moreover, g_i must be in $\bot_e$ but not in $\bot_x$. On the other hand, if g_i was in $h \leftarrow g_1, \ldots, g_{i-1}, g_i$ it can be reached from $h, g_1, \ldots, g_{i-1}$, so it must also be in $\bot_x$.

Consider, for example, the following bottom-clause for an example e:

$$\bot_e = l(A) \leftarrow h\_c(A, B), h\_c(A, C), d(B), o\_c(B), f(C).$$

and the following clause c:

$$c = l(A) \leftarrow h\_c(A, B), h\_c(A, C), d(C), o\_c(B).$$

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' 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 is

indeed covered by the clause. This suggests the following approach: given an example e construct the corresponding bottom clause $\bot_e$ and generate a set $\mathcal{S}$ with all legal clauses c such that c θ-subsumes $\bot_e$. 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 $\{\mathcal{S}_1^+, \mathcal{S}_2^+, \ldots, \mathcal{S}_n^+, \mathcal{S}_1^-, \mathcal{S}_2^-, \ldots, \mathcal{S}_m^-\}$. Finding the best clauses should be just a question of searching for clauses that appear in most $\mathcal{S}_i^+$ and not in $\mathcal{S}_i^-$. More precisely, if we allow no noise, then we would like to find the clause with the largest coverage from $\cup_i \mathcal{S}_i^+ \setminus \cup_j \mathcal{S}_j^-$. Note that we are not interested in the examples, but in the set of all clauses of interest, which would to a first approximation be close to $\cup_i \mathcal{S}_i^+$. 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 in Section 2.

Our Algorithm

To *estimate* the coverage of all clauses we will walk over all examples $e \in E$ as follows. Visit positive examples first, and assume that we do not care about clauses that only cover negative examples:

- If $e \in E^+$ and $c \notin \mathcal{S}$, add c to $\mathcal{S}$ and state that c covers one positive example.
- If $e \in E^+$ and $c \in \mathcal{S}$, state that c covers one more positive example.
- If $e \in E^-$ and $c \in \mathcal{S}$, state that c covers one more negative example.
- If $e \in E^-$ and $c \notin \mathcal{S}$, do nothing.

We therefore need to define an abstract set, that we call *decorated set* $\mathcal{S}$, with all clauses and their coverage. A *decorated set* is a set whose elements are clauses, and attached to each element are counters (one counter for each class of the learning problem). With this abstraction we can easily implement any theory construction algorithm as shown in Figure 2.

$generaliseTrieingMDIE(B, E^+, E^-, C)$:
Given: background knowledge B, finite training set $E = E^+ \cup E^-$, constraints C.
Return: a hypothesis H that explains E^+ and satisfies C.

1. $H = \emptyset$
2. **while** $E^+ \neq \emptyset$ **do**
3. $\quad h = learnTrieingMDIE(B, E^+, E^-, C)$
4. $\quad E^+ = E^+ \setminus covered(h)$
5. $\quad H = H \cup h$
6. $\quad B = B \cup h$
7. **endwhile**
8. **return** H

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

The main difference with systems like Progol or Aleph concerns the inner procedure $learnTrieingMDIE()$. We next describe how clauses are learned in the *Trieing MDIE* algorithm (see Figure 3). The *Trieing MDIE* algorithm has two basic stages. First a decorated set $\mathcal{S}$ is constructed (lines 1 to 9) and then the best

clause (according to some metric) is found by inspection of the set (line 10). The decorated set $\mathcal{S}$ is constructed as described above. First, all positive examples are processed and then a pruning procedure, *prune*(), is invoked to remove useless clauses from $\mathcal{S}$ (e.g., clauses with coverage lower than some predefined minimum number of examples). Next, all negative examples are also processed and then the set is pruned again.

$learnTrieingMDIE(B, E^+, E^-, C)$:
Given: background knowledge B, finite training set $E = E^+ \cup E^-$, constraints C.
Return: the *best* hypothesis that explains some of the E^+ and satisfies C.

1. $\mathcal{S} = \emptyset$
2. **foreach** $e \in E^+$ **do**
3. $\quad fillSet(\mathcal{S}, B, e, C)$
4. **endforeach**
5. $\mathcal{S} = prune(\mathcal{S}, C)$
6. **foreach** $e \in E^-$ **do**
7. $\quad fillSet(\mathcal{S}, B, e, C)$
8. **endforeach**
9. $\mathcal{S} = prune(\mathcal{S}, C)$
10. **return** $bestClauseInTrie(\mathcal{S}, C)$

Fig. 3. The learning algorithm of *Trieing MDIE*.

Figure 4 shows how the set $\mathcal{S}$ is filled for each example. First we generate the bottom clause (line 2). Then, using the bottom clause, we generate all valid clauses[4] (line 4), normalise them (line 5), and insert them in the set (line 6). Normalisation orders the literals according to the Prolog "@ <" order relation. We generate all renaming of existential variables to check if a variant already exists in the trie, therefore guaranteeing a unique representation for each clause. The *insertUpdateInSet*() procedure works as follows. If the example class is positive, the clause is inserted into $\mathcal{S}$ and the positive counter is updated. If the example class is negative, the clause is not added to $\mathcal{S}$, only the negative counter of the clause is updated. This means that the clauses generated from the negative examples that are not in $\mathcal{S}$ are discarded.

$fillSet(\mathcal{S}, B, e, C)$:
Given: decorated set $\mathcal{S}$, background knowledge B, example e, constraints C.

1. $class = getExampleClass(e)$
2. $bottom = saturate(e, B, C)$
3. **do**
4. $\quad clause = findNewValidClause(bottom, C)$
5. $\quad clause = normalise(clause)$
6. $\quad insertUpdateInSet(clause, \mathcal{S}, class)$
7. **while** $clause \; ! = \emptyset$

Fig. 4. From an example to a set of clauses.

The algorithm is shown to be complete when compared to the traditional Prolog resolution approach of computing the coverage. Therefore, the coverage

[4] Clauses satisfying the language and bias constraints.

calculated for a clause by the algorithm should be interpreted as an estimate since it may not be the exact (*correct*) value.

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 Factor

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 that 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 factor* 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 a 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 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 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 representation in the trie.

5 Experiments and Results

To evaluate the impact of the proposed approach we adapted the April ILP system [13] so that it could be executed with support for tries and applied the system to well known data sets. 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 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.

For each data set, the system was executed for a standard MDIE implementation using a deterministic top-down breadth-first search procedure (**dtd-bf**) and for MDIE using tries (**Trieing**). We varied the maximum depth of the clauses from 2 (one literal in the body) to 4 (3 literals in the body). Table 2 compares the execution times and the number of clauses generated by both approaches.

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

Table 2. Execution time and number of clauses generated.

The results confirm that *Trieing MDIE* generates considerably more clauses (ranging from two up to five fold) than the *dtd-bf MDIE* 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 for *dtd-bf MDIE*, *Trieing MDIE* will also compare favorably.

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 very good results, both in terms of accuracy and memory usage.

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

Data set	Accuracy		Memory	
	dtd-bf	Trieing	dtd-bf	Trieing
Carcinogenesis	72 / 48 / 51	72 / 62 / 69	19 / 19 / 122	19 / 21 / 59
Mutagenesis	65 / 71 / 74	65 / 94 / 82	10 / 19 / 99	13 / 13 / 22

Table 3. Accuracy and memory usage (in each cell the 3 values represent clause length of 2/3/4).

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 three 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 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

This work has been partially supported by Fundação para a Ciência e Tecnologia and by project Myddas (POSC/EIA/59154/2004). Nuno A. Fonseca is funded by FCT grant SFRH/BPD/26737/2005.

References

1. S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–317, 1991.
2. Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.
3. 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.

4. 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.
5. Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
6. 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.
7. 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.
8. Rui Camacho. *Inductive Logic Programming to Induce Controllers*. PhD thesis, Univerity of Porto, 2000.
9. 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.
10. S. Muggleton *Inverse Entailment and Progol*, New Generation Computing, Special issue on Inductive Logic Programming", vol 13, N. 3-4, pp 245-286, 1995.
11. S. Muggleton, *Inverse Entailment and Progol*, New Generation Computing, Special issue on Inductive Logic Programming. 245-286, vol 13, N. 3-4, 1995.
12. Aleph. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.
13. 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.
14. 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.
15. 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
16. 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
17. E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.
18. 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.
19. P. Graf. Term Indexing. Number 1053 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
20. 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.
21. Akihiro Yamamoto *Which Hypotheses Can Be Found with Inverse Entailment?* ILP '97: Proceedings of the 7th International Workshop on Inductive Logic Programming, pp 296–308, 1997.