Experimental Evaluation of a Caching Technique for ILP [⋆]

Nuno Fonseca[1], Vitor Santos Costa[3], Fernando Silva[1], and Rui Camacho[2]

[1] DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
`{nf,fds}@ncc.up.pt`
[2] COPPE/Sistemas, Universidade Federal do Rio de Janeiro
Centro de Tecnologia, Bloco H-319, Cx. Postal 68511 Rio de Janeiro, Brasil
`vitor@cos.ufrj.br`
[3] Faculdade de Engenharia & LIACC, Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
`rcamacho@fe.up.pt`

Keywords: Inductive Logic Programming, Coverage Caching

1 Introduction

Inductive Logic Programming (ILP) is a Machine Learning technique that has been quite successful in knowledge discovery for relational domains. ILP systems implemented in Prolog challenge the limits of Prolog systems due to heavy usage of resources such as database accesses and memory usage, and to very long execution times. The major reason to implement ILP systems in Prolog is that the inference mechanism implemented by the Prolog engine is fundamental to most ILP learning algorithms. ILP systems can therefore benefit from the extensive performance improvement work that has taken place for Prolog. On the other hand, ILP is a non-classical Prolog application because it uses large sets of ground facts and requires storing a large search tree.

One major criticism of ILP systems is that they often have long running times. A technique that tries to tackle this problem is coverage caching [**?**]. Coverage caching stores previous results in order to avoid recomputation. Naturally, this technique uses the Prolog internal database to store results. The question is: does coverage caching successfully reduce the ILP systems running time? To obtain an answer to this question we evaluated the impact of the coverage caching technique using the April [**?**] ILP system with the YAP Prolog system. To understand the results obtained we profiled April's execution and present initial results. The contribution of this paper is twofold: to an ILP researcher it provides an evaluation of the coverage caching technique implemented in Prolog using well known datasets; to a Prolog implementation researcher it shows the need of efficient internal database indexing mechanisms.

[⋆] 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.

To a brief introduction to some concepts and terminology used in ILP we refer to [?,?]. An extended version of this paper available as a technical report [?].

2 Coverage Caching

The objective of an ILP system is the induction of logic programs. As input an ILP system receives a set of examples E, divided in positive (E^+) and negative examples (E^-), of the concept to learn, and some prior knowledge B, or *background knowledge*. Both examples and background knowledge are usually represented as logic programs. An ILP system tries to produce a theory (logic program) where positive examples succeed and the negative examples fail. To find a satisfactory theory, an ILP system searches through a search space of the permitted clauses.

The coverage of a clause h_i is computed by testing the clause against the positive and negative examples. This is done by verifying for each example e in E if $B \wedge h_i \vdash e$. Coverage caching aims at reducing the computation time spent in coverage tests by storing the coverage lists (the set of positive and negative examples covered by the clause) for each clause generated.

The coverage lists are used as follows. An hypothesis S is generated by adding a literal to a hypothesis G. Let $Cover(G) = \{all\ e \in E\ such\ that\ B \wedge G \vDash e\}$. 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 [?] extended this scheme by proposing what is designated as coverage caching. The coverage lists are permanently stored and reused whenever necessary, thus coverage computation of a particular clause is performed only once. Coverage lists reduce the effort in coverage computation at the cost of significantly increasing memory consumption.

In order to reduce execution time the cache must be very efficient, by this we mean that insertions and retrievals of elements in the cache should be done very fast. The April system uses the YAP Prolog internal and clausal database to store clauses's coverage, the only solution available within the Prolog language.

3 Experiments and Results

To analyze the impact of the coverage caching technique on both memory usage and execution time, we conducted a series of experiments using datasets from the Machine Learning repositories of the Universities of Oxford[4] and York[5]. The experiments were made on an AMD Athlon(tm) MP 2000+ dual-processor PC with 2GB of memory, running the Linux RedHat (kernel 2.4.20) operating system. We used version 0.5 of the April ILP system and version 4.3.24 of the YAP Prolog. A more complete description of the experiments can be found in [?].

[4] `http://www.comlab.ox.ac.uk/oucl/areas/machlearn/applications.html`

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

Note that in order to speedup the experiments we limited the search space on some datasets. This reduces the total memory usage and execution time needed to process the dataset at the cost of finding possible worst theories. However, since we are comparing the memory consumption and execution time when using coverage caching or not using it, the estimate we obtain will still give a good idea of the impact of the cache.

Table **??** presents the impact of activating coverage caching in April. It shows the total number of hypotheses generated ($\mid H \mid$), the execution time, the memory usage, and the impact in performance for execution time and memory usage (given as a ratio between using coverage caching and not using coverage caching). The memory values presented correspond to the total memory used by April.

Dataset	$\mid H \mid$	**Time (sec.)**		**Memory (bytes)**		**yes/no(%)**	
		no	**yes**	**no**	**yes**	`Time`	`Memory`
amine uptake	66933	58.37	357.4	3027460	11255228	`612.30`	`371.77`
carcinogenesis	142714	616.38	506.65	7541316	13542528	`82.19`	`179.57`
choline	803366	1840.25	13596.07	5327052	32537788	`738.81`	`610.80`
krki	2579	3.78	1.15	2225176	2318084	`30.42`	`104.17`
mesh	283552	637.34	3241.73	7255884	25733376	`508.63`	`354.65`
multiplication	478	8.87	8.93	4261768	4422080	`100.67`	`103.76`
pyrimidines	372320	915.95	5581.91	5659544	27856496	`609.41`	`492.20`
proteins	433271	7837.96	794.4	27075788	27495636	`10.13`	`101.55`

Table 1. Impact of coverage caching

As expected, the results indicate a significant increase in memory usage when coverage caching is activated. However, unexpectedly the use of coverage caching also increased the execution time, in some cases more than 5 times, for larger datasets (i.e. datasets with larger number of examples and $\mid H \mid$). The `proteins` dataset shows a reduction of around 90% in the execution time which is what one would like to observe when employing a caching mechanism.

The overheads in execution time were somehow unexpected and prompted us to further investigate the reasons for this behavior. We decided to activate YAP's profiling and then rerun the April system for all datasets previously considered.

One first issue that we would like to clarify is whether coverage caching reduced the number of goal invocations executed. Table **??** shows the total number of calls and retries performed by YAP with the cache activated and deactivated. The result values represent the aggregate number of calls and retries for all datasets. Note that the number of retries shown, with the cache deactivated, is lower than the real value because in some datasets the YAP counters overflowed. In these cases the maximum value possible was used instead. The use of cache reduced the number of calls by 90% and reduced the number of retries by at least 15%. This shows that the use of caching clearly achieves the goal of reducing computation but surprisingly the execution time increased by 56%. Note that the number of calls were reduced by 30 billions approximately.

Module	**cache=yes**	**cache=no**	**yes/no**
Calls	3,141,742,379	33,508,263,954	0.09
Retries	26,112,058,881	>30,730,206,551	0.84
Time (sec.)	38731.23	24718.04	1.56

Table 2. Total number of calls and execution time

We analyzed the profiling logs trying to identify the predicates that were causing the inefficiency problems. Table **??** presents a summary of the number of calls for the predicates considered more relevant. Since the number of calls for most of the predicates decreased with the use of cache, we selected those predicates whose number of calls were still very high, or increased, or operate the Prolog database.

Predicate	**cache=yes**	**cache=no**	**Variation**
prolog:abolish/1	13,304	17,204	-3,900
prolog:assert/1	98,362	5,663	+92,699
prolog:assertz/1	1,592,288	2,049,054	-456,766
prolog:numbervars/3	5,265,269	4,349	+5,260,920
prolog:eraseall/1	5,902,918	7,734,758	-1,831,840
prolog:recordz/3	5,665,526	7,562,647	-1,897,121
prolog:copy_term/2	5,677,883	515,905	+5,161,978
prolog:call/1	6,396,015	8,314,571	-1,918,556
prolog:erase/1	20,674,230	24,155,488	-3,481,258
prolog:recorda/3	25,866,551	23,760,276	+2,106,275
prolog:ground/1	110,305,158	90,361,520	+19,943,638
idb_cache:idb_keys	5,166,049 (789,534)	0	+5,166,049

Table 3. Number of calls for some predicates. The `idb_cache::idb_keys` predicate is a dynamic predicate used to store cache keys, and value in parenthesis is the number of recalls.

Table **??** shows that in the `prolog` module the number of calls increased only for the `assert`, `recorda`, `numbervars`, `copy_term`, and `ground` predicates. The increase of calls in the `idb_cache` module was most felt in the `idb_keys` predicate. All the other predicates in the `idb_cache` make calls to the predicates in the `prolog` module, in particular to the `recorded` predicate that YAP could not show in the profile statistics. From the profile results we estimated that the number of calls to the `recorded` predicate increased by around 22 millions when using coverage caching.

Since YAP does not provide the time spent computing each predicate, we did further experiments to measure the impact of each of those predicates in the execution time. We observed that the predicates that deal with the internal database and clausal database are the main source of execution time overhead. In particular, the dynamic predicate `idb_keys` and the database predicate `recorded` are those with biggest impact. These two heavily used predicates are the main cause for coverage caching inefficiency. Since the reduction or elimination of Prolog database operations is not possible, a solution to cope with this problem is the improvement of the indexing mechanism of YAP Prolog internal database. Moreover, we find that it would be very much useful the support of an efficient indexing mechanism using multiple keys.

4 Conclusions

ILP systems are non-classical Prolog applications because of the use of large sets of ground facts and high resource consumption (memory and CPU). We provided results showing the impact on memory usage and execution time of an ILP technique called coverage caching. This technique uses intensively the internal database to store results in order to avoid recomputation. An empirical analysis of the coverage caching technique using the April ILP system with Yap Prolog showed a degradation of the execution time although it significantly reduced the number of Prolog calls and retries. To pinpoint this unexpected behavior we profiled April using YAP Prolog. The analysis of the profile data lead us to conclude that the use of YAP's database is the cause for performance degradation. Improving the indexing mechanism of YAP Prolog internal database, moreover including efficient support for indexing with multiple keys, will certainly improve April's performance as well as other applications that use the database intensively. It is our hope that these findings will motivate Prolog implementors to further excel their implementations.

References

1. Nuno Fonseca, Fernando Silva, Rui Camacho, and Vitor S. Costa. Induction with April - A preliminary report. Technical report, DCC-FC & LIACC, UP, 2003.
2. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
3. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1997.
4. Nuno Fonseca, Vitor S. Costa, Fernando Silva, and Rui Camacho. On the implementation of an ilp system with prolog. Technical report, DCC-FC & LIACC, UP, 2003.
5. James Cussens. Part-of-speech disambiguation using ilp. Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, 1996.