On Applying Tabling to Inductive Logic Programming

Ricardo Rocha[1], Nuno Fonseca[1], and Vítor Santos Costa[2⋆]

[1] DCC-FC & LIACC
University of Porto, Portugal
{ricroc,nf}@ncc.up.pt
[2] Department of Biostatistics and Medical Informatics
University of Wisconsin-Madison, USA
vitor@biostat.wisc.edu

Abstract. Inductive Logic Programming (ILP) is an established subfield of Machine Learning. Nevertheless, it is recognized that efficiency and scalability is a major obstacle to an increased usage of ILP systems in complex applications with large hypotheses spaces. In this work, we focus on improving the efficiency and scalability of ILP systems by exploring tabling mechanisms available in the underlying Logic Programming systems. Tabling is an implementation technique that improves the declarativeness and performance of Prolog systems by reusing answers to subgoals. To validate our approach, we ran the April ILP system in the YapTab Prolog tabling system using two well-known datasets. The results obtained show quite impressive gains without changing the accuracy and quality of the theories generated.

1 Introduction

Inductive Logic Programming (ILP) has been successfully applied to problems in several application domains [1]. Nevertheless, the flexibility of ILP comes at a price: for complex applications with large hypotheses spaces, ILP systems can take several hours, if not days, to return a theory. Past research on improving the efficiency of ILP systems has mainly focused in reducing their sequential execution time, either by reducing the number of hypotheses generated [2, 3], or by efficiently testing candidate hypotheses [4, 5]. One key observation in this research is that ILP search space is highly redundant: we repeatedly test similar, and sometimes even the same, hypotheses. This argues for using techniques such as *memoing* or *tabling* [6], that have been developed for this very purpose.

On the other hand, ILP systems are often developed on top of logic programming systems, such as Prolog systems. One reason is that ILP systems can

⋆ This work has been partially supported by APRIL (POSI/SRI/40749/2001), Myddas (POSC/EIA/59154/2004), U.S. Air Force (grant F30602-01-2-0571), and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia (FCT) and Programa POSC. Nuno Fonseca is funded by the FCT grant SFRH/BD/7045/2001.

benefit from the extensive work done in improving the performance of Prolog systems. An emerging technology is *tabling*, that showed to be very effective for a variety of applications. Tabling based models can reduce the search space, avoid looping, and have better termination properties than traditional Prolog based models. The question thus arises if the tabling mechanisms being developed for efficient execution of logic programs can be useful in improving ILP performance.

In this work, we show that tabling can indeed significantly reduce the execution time of ILP applications. We present two different approaches to achieve this goal. Our first approach is a direct application of tabling to query execution. The second approach is designed to take advantage of the redundancy in ILP search. We validate our approach by experimenting the April ILP system [7] on two well known ILP datasets. One advantage of using April in our study is that it includes a strong caching mechanism, thus giving us a good baseline for our studies. Tabling is implemented through the YapTab Prolog tabling system [8].

The remainder of the paper is organized as follows. First, we introduce the motivation for our work. Then, we briefly describe tabling for logic programs. Next, we discuss how tabling can be used to speedup ILP applications. We then present initial experimental results and conclude by outlining some conclusions.

2 Background and Motivation

The fundamental goal of an ILP system is to find a consistent and complete *theory*, from a set of examples and prior knowledge, the *background knowledge*, that explains all given positive examples, while being consistent with the given negative examples. In general, the background knowledge and the set of examples can be arbitrary logic programs.

Since it is not usually obvious which set of hypotheses should be selected as the theory, an ILP system must traverse the *hypotheses space* searching for a set with the desired properties. A general ILP system thus spends most of its time evaluating hypotheses, either because the number of examples is large or because testing each example is computationally hard.

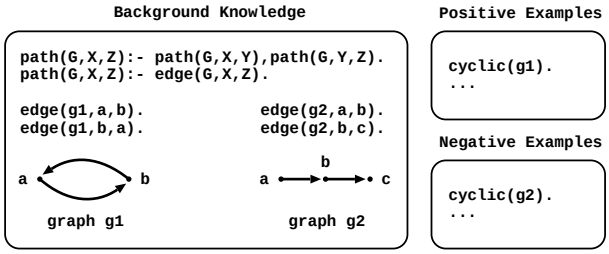

Fig. 1. Representing cyclic graphs in a ILP dataset

An important characteristic of ILP systems is that they generate candidate hypotheses (clauses) which have many similarities among them. Usually, these similarities tend to correspond to common prefixes (subgoals) among the candidate hypotheses. Consider, for example, a background knowledge containing a set of directed graphs, represented by `edge/3` facts, with a relation of reachability, given by a `path/3` predicate (see Fig. 1 for details). Consider also that we are interested in learning the concept of being a cyclic graph.

Now assume that, during the search process, the ILP system generates an hypothesis '`cyclic(G):- path(G,X,Y).`' which obtains *good coverage*, that is, the number of positive examples covered by it is high and the number of negative example is low. Then, it is quite possible that the system will use it to generate more specific hypotheses such as '`cyclic(G):- path(G,X,Y),edge(G,Y,Z).`'.

Computing the coverage of an hypothesis requires, in general, running all positives and negatives examples against the clause. For example, to evaluate if the example `cyclic(g1)` is covered by the hypothesis '`cyclic(G):- path(G,X,Y).`', the system executes the goal `once(path(g1,X,Y))`. The `once/1` predicate is a primitive that prunes over the search space preventing the unnecessary search for further answers. It can be defined in Prolog as '`once(Goal):- call(Goal),!.`'.

If the same example, `cyclic(g1)`, is later evaluated against the other hypothesis, goal `once(path(g1,X,Y),edge(g1,Y,Z))`, part of the computation of `path(g1,X,Y)` will be repeated. This suggests two approaches to avoid recomputation. First, if the computation of `path(g1,X,Y)` is computationally expensive, we can table this query. Second, the subgoal `path(g1,X,Y)` forms a prefix of the new clause. We can *table prefixes*, in the hope that they will be called repeatedly.

Notice that both approaches have problems. The first approach will only work if the computation for a subgoal is expensive. It will bring no benefit if, say, the subgoal reduces to a database access. The second approach is only useful if we repeatedly generate the same prefix. If we have a large number of prefixes which are only called a few times, we may need large amounts of space to store the tables, and gain little time-wise. To best implement these approaches requires some understanding of the basic tabling mechanisms, that we discuss next.

3 Tabling for Inductive Logic Programming

The basic idea behind tabling is straightforward: programs are evaluated by storing newly found answers for current subgoals in an appropriate data space, called the *table space*. The method then uses this table to verify whether calls to subgoals are repeated. Whenever such a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses. One of the major characteristics of this execution model is that it reduces the search space by avoiding the recomputation of tabled subgoals. This is the most significant contribution that tabling can offer to ILP. Moreover, because tabling based models are also able to avoid infinite loops, they can ensure termination for a wider class of programs. The latter can be useful when dealing with datasets with recursive definitions in the background knowledge.

3.1 Tabled Evaluation

Figure 2 uses the example from the background knowledge in Fig. 1 to illustrate how tabling works. At the top, the figure shows the program code (the left box), and the final state of the table space (the right box). Declaration '`:-table path/3.`' indicates that calls to predicate `path/3` should be tabled.

The main sub-figure below shows the evaluation sequence for the query goal '`?- path(g1,b,Z).`'. Note that traditional Prolog would immediately enter an infinite loop because the first clause of `path/3` leads to a repeated call to `path(g1,b,Z)`. In contrast, if tabling is applied then termination is ensured.

Whenever a tabled subgoal is first called, a new entry is added to the table space. We name these calls *generator nodes* (nodes depicted by white oval boxes). In this example, the execution begins with a generator. The first step is to resolve `path(g1,b,Z)` against the first clause for `path/3`, creating node 1. Node 1 is a variant call to `path(g1,b,Z)`. We do not resolve the subgoal against the program at these nodes, instead we consume answers from the table. Such nodes are thus called *consumer nodes* (nodes depicted by black oval boxes). At this point, the table does not have answers for this call, and thus, the current evaluation is *suspended*. We then backtrack to node 0, thus calling `edge(g1,b,Z)`. The `edge/3` predicate is not tabled, hence it must be resolved against the program, as Prolog would. The first clause for `edge/3` fails, but the second succeeds obtaining an answer for `path(g1,b,Z)` (step 4).

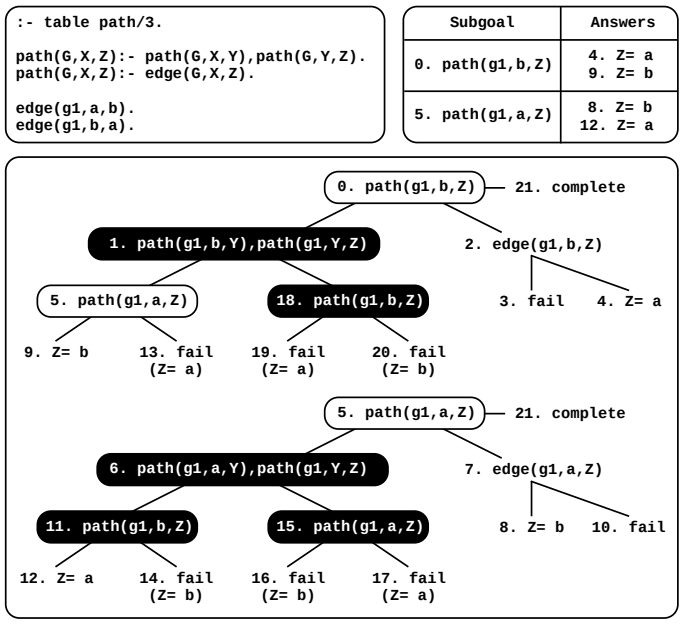

Fig. 2. A tabled evaluation

In the continuation, we backtrack again to node 0, but now it has no more clauses left to try. So, we check whether it has *completed*. It has not, as node 1 has now one unconsumed answer. We thus forward the answer to it, and `path(g1,a,Z)` is then called. As this is the first call to `path(g1,a,Z)`, we add a new entry for it in the table, and proceed as shown in the bottommost tree. Again, `path(g1,a,Z)` calls itself recursively, suspends at node 6, backtracks, and succeeds with `Z=b` (step 8). We then follow a Prolog-like strategy and continue forward execution. The binding `Z=b` is thus returned to `path(g1,b,Z)` and stored in its table entry (step 9). This will be the last answer to `path(g1,b,Z)`, but we can only prove so after fully exploiting the tree.

We then fail in step 10, backtrack to node 5, and resume node 6 with answer `Z=b`. This leads to a new consumer for `path(g1,b,Z)` (node 11). The table has two answers for it, so we can continue immediately. This gives new answers

to `path(g1,a,Z)` (step 12) and to `path(g1,b,Z)` (step 13). However, this last answer repeats what we found in step 4. Tabled resolution do not stores duplicate answers in the table. Instead, repeated answers *fail*. This is how we avoid unnecessary computations, and even looping in some cases.

Backtracking sends us back to consumer node 11. We then consume the second answer for it, which generates a repeated answer, so we fail again (step 14). We then try the second answer for node 6, again leading to a repeated subgoal (node 15) and two repeated answers (steps 16 and 17). We then fail back to node 5, but at this point, all answers to the consumers below (nodes 6, 11, and 15) have been tried. However, unfortunately, node 5 cannot complete, because it depends on subgoal `path(g1,b,Z)` (node 11). Completing `path(g1,a,Z)` earlier is not safe because we can loose answers. Note that, new answers can still be found for subgoal `path(g1,b,Z)`. If new answers are found, node 11 should be resumed with the newly found answers, which in turn can lead to new answers for subgoal `path(g1,a,Z)`. If we complete sooner, we can loose such answers.

Execution thus backtracks and we try the answer left for node 1. Steps 19 to 20 show that again we only get repeated answers. We fail and return to node 0. All nodes in the trees for node 0 and node 5 have been exploited. As these trees do not depend on any other tree, we are sure no more answers are forthcoming, so at last step 21 declares the two trees to be complete.

3.2 Tabling Subgoals and Conjunction of Subgoals

The first application of tabling in ILP is simply to table subgoals. The main advantage of this approach is that we need to perform minimal changes to the ILP system. A drawback is that this technique will not help if the subgoal generates a very small computation, say, if the subgoal is defined extensionally in the database as Prolog facts. A second approach is to take advantage of the tabling paradigm and replace the conjunction of predicates in the hypotheses with proper tabled predicates inferred during execution. Consider, for example, the following set of hypotheses:

```
cyclic(G):- edge(G,X,Y), path(G,Y,Z), edge(G,Z,X).
cyclic(G):- edge(G,X,Y), path(G,Y,Z), edge(G,X,Z).
cyclic(G):- edge(G,X,Y), path(G,Y,Z), path(G,Z,X).
```

Note that the two first subgoals, `edge(G,X,Y)` and `path(G,Y,Z)`, are common to all the hypotheses. Thus, if we are able to table the conjunction of both, we only need to compute it once. This idea can be recursively applied as the system generates more specific hypothesis. This idea is similar to the *query packs* technique proposed by Blockeel *et al.* [4].

To implement this approach, we designed the following solution. First, we use a single predicate, `t_all/2`, to table all the conjunctions. The first argument for `t_all/2` is an atom that defines the name given to the conjunction. The second is the set of variables involved. This predicate then calls a `t_conj/2` predicate (with the same arguments) where the conjunctions are defined. The clauses for the `t_conj/2` predicate are dynamically asserted by the ILP system

as new conjunctions are generated. A conjunction of N subgoals is defined as the conjunction of the $N-1$ previous subgoals followed by the Nth subgoal. For example, we would have the following clauses for the set of hypotheses above:

```
% the tabled predicate for all the conjunctions
:- table t_all/2.
t_all(ConjunctionName,VarsList):- t_conj(ConjunctionName,VarsList).

% level 1 conjunctions
t_conj(edge,[V1,V2,V3]):- edge(V1,V2,V3).
t_conj(path,[V1,V2,V3]):- path(V1,V2,V3).

% level 2 conjunctions
t_conj(edge_path,[V1,V2,V3,V4,V5,V6]):- t_all(edge,[V1,V2,V3]),
                                        t_all(path,[V4,V5,V6]).
```

Finally, we need to transform the clauses for the hypotheses. We thus replace the conjunctions of subgoals in the hypotheses to calls to the `t_all/2` predicate. For example, the previous set of hypotheses will be transformed to:

```
cyclic(G):- t_all(edge_path,[G,X,Y,G,Y,Z]), t_all(edge,[G,Z,X]).
cyclic(G):- t_all(edge_path,[G,X,Y,G,Y,Z]), t_all(edge,[G,X,Z]).
cyclic(G):- t_all(edge_path,[G,X,Y,G,Y,Z]), t_all(path,[G,Z,X]).
```

Note that this may cause the same variables to appear at several positions in the second argument for the `t_all/2` predicate (e.g., both `G` and `Y` appear twice for `edge_path`). In practice, the tabling engine only stores the answers once for each different variable, so this only has a small cost. A major problem with our approach is the amount of memory that is needed to represent the answers for the different conjunctions. A simple solution is to abolish the full set of tables from the table space when we run out of memory. An alternative would be to abolish the tables potentially useless when we backtrack in the hypotheses space. This later approach requires further study to avoid incorrect deletions.

At that point, we should reinforce the differences between tabling and between the approach of tabling conjunction of subgoals. Tabling is an implementation technique that comes for free if using a Prolog engine with such support. The tabling of conjunctions is an alternative evaluation strategy that can be explored by ILP systems. Like in query packs, this is done automatically in the innards of the ILP system, and can be parameter controlled. Thus, the final user of the system only needs to declare the strategy to be used: no tabling, subgoal tabling, or subgoal and conjunction tabling.

4 Initial Experimental Results

To evaluate the impact of using tabling in real application problems, we ran the April ILP system [7] with the YapTab Prolog tabling system [8] using two ILP datasets: *mutagenesis* and *carcinogenesis*. April was configured to find hypotheses using breadth-first search, and to evaluate hypotheses using a heuristic that relies on the number of positive and negative examples. YapTab is based on the current development version of Yap, version 4.5.7. The environment for our

experiments was an AMD Athlon MP 2600+ processor with 2 GBytes of main memory and running the Linux kernel 2.6.11.

To evaluate hypotheses we experimented with three different approaches: **(i)** without tabling; **(ii)** subgoals being evaluated using tabling; and **(iii)** subgoals and conjunction of subgoals being evaluated using tabling.

Table 1 shows the running times, in seconds, and the table memory usage, in Mbytes, for the three approaches. We use **na** to mark the experiments not ran and **mo** to mark the runs where a memory overflow occurred. Note that we are not considering any strategy to avoid memory overflows. The value **nodes** is the upper bound on the number of hypotheses, and **hypotheses** is the number of hypotheses effectively generated during the search.

Datasets	**Running Time (s)**			**Table Usage (Mb)**	
nodes/hypotheses	**without**	**subgs**	**conjs**	**subgs**	**conjs**
mutagenesis					
1,000/981	> 4 hours	94	92	2	6
10,000/6,514	na	162	140	5	205
20,000/14,020	na	169	146	6	281
30,000/20,299	na	197	mo	6	mo
40,000/26,484	na	219	mo	6	mo
50,000/32,852	na	236	mo	6	mo
carcinogenesis					
1,000/998	1	1	1	3	11
10,000/9,998	7	9	13	11	259
20,000/19,998	81	91	mo	11	mo
30,000/29,932	121	124	mo	11	mo
40,000/39,932	161	154	mo	11	mo
50,000/49,869	225	209	mo	12	mo

Table 1. Running times and table usage with one example as seed

The results obtained for *mutagenesis* show that tabled evaluation can significantly reduce the execution time for these kind of problems. In particular, for the subgoal approach the gains are quite impressive. The theorem proving effort involved to evaluate a single example against an hypothesis is quite high for this dataset. The conjunction approach also achieved the goal of reducing the execution time (however, we were not able to use more than 20,000 nodes). Regarding memory usage, the results show an insignificant increase in memory consumption when tabling subgoals and a more considerable increase when tabling conjunctions of subgoals.

In the *carcinogenesis* dataset, the results where not so good. The main reason for this relies on the type of predicates that compose its background knowledge. In this dataset most of the predicates are defined extensionally in the database as Prolog facts, and thus, it is quite difficult for the tabling engine reduce the execution time. Even so, when we increase the size of the search space for the *carcinogenesis* dataset (for more than 40,000 nodes), the tabling subgoal approach slightly reduces the execution time when compared with the execution

without tabling. Regarding the conjunction approach we were not able to see its impact in this dataset.

The results obtained suggest that tabling is particular suited for ILP applications with a background knowledge non-deterministic, as the *mutaganesis* dataset. The results also confirm that tabling is not suitable for datasets with a background knowledge defined extensionally. However, apart the small extra memory consumption in the case of tabling subgoals, the execution with tabling do not introduces significant overheads.

5 Conclusions and Further Work

In this work, we proposed the ability of using tabling mechanisms available in the underlying Logic Programming systems to minimize recomputation in ILP systems. The results obtained showed that tabling based models are indeed able to improve the performance of ILP applications. In particular, for some applications, they show quite impressive gains. As tabled evaluation does not influences the accuracy and quality of the models found, we believe that our proposals would apply to several ILP systems.

A major problem with our current implementation, is that we can increase the table memory usage arbitrarily when tabling conjunction of subgoals. We plan to study how we can abolish potentially useless tables when we backtrack in the hypotheses space. We also plan to further investigate the impact of applying our proposals to a larger set of ILP applications.

References

1. Network of Excellence in Inductive Logic Programming ILPnet2: (ILP Applications) Available from `http://www.cs.bris.ac.uk/~ILPnet2/Applications`.
2. Nédellec, C., Rouveirol, C., Adé, H., Bergadano, F., Tausend, B.: Declarative Bias in ILP. In Raedt, L.D., ed.: Advances in Inductive Logic Programming. IOS Press (1996) 82–103
3. Sebag, M., Rouveirol, C.: Tractable Induction and Classification in First-Order Logic via Stochastic Matching. In: International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1997) 888–893
4. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research **16** (2002) 135–166
5. Santos Costa, V., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., Laer, W.V.: Query Transformations for Improving the Efficiency of ILP Systems. Journal of Machine Learning Research **4** (2002) 465–491
6. Michie, D.: Memo Functions and Machine Learning. Nature **218** (1968) 19–22
7. Fonseca, N., Camacho, R., Silva, F., Santos Costa, V.: Induction with April: A Preliminary Report. Technical Report DCC-2003-02, Department of Computer Science, University of Porto (2003)
8. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87