Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 124–133
http://www.floc-conference.org/ICLP-home.html

DEDICATED TABLING FOR A PROBABILISTIC SETTING

THEOFRASTOS MANTADELIS [1] AND GERDA JANSSENS [1]

[1] Departement Computerwetenschappen, Katholieke Universiteit Leuven
Celestijnenlaan 200A - bus 2402, 3001 Heverlee, Belgium
E-mail address: {Theofrastos.Mantadelis,Gerda.Janssens}@cs.kuleuven.be

Abstract. ProbLog is a probabilistic framework that extends Prolog with probabilistic facts. To compute the probability of a query, the complete SLD proof tree of the query is collected as a sum of products. ProbLog applies advanced techniques to make this feasible and to assess the correct probability. Tabling is a well-known technique to avoid repeated subcomputations and to terminate loops. We investigate how tabling can be used in ProbLog. The challenge is that we have to reconcile tabling with the advanced ProbLog techniques. While standard tabling collects only the answers for the calls, we do need the SLD proof tree. Finally we discuss how to deal with loops in our probabilistic framework. By avoiding repeated subcomputations, our tabling approach not only improves the execution time of ProbLog programs, but also decreases accordingly the memory consumption. We obtain promising results for ProbLog programs using exact probability inference.

1. Introduction

ProbLog [4] is a probabilistic framework that extends Prolog with probabilistic facts and answers several kinds of probabilistic queries. While the framework includes different inference methods, we focus for this paper on the exact probability inference method.

The implementation of ProbLog [8] is based on the use of tries [5] and reduced ordered binary decision diagrams (ROBDDs) [1, 2]. The execution of ProbLog programs uses SLD-resolution to collect all the proofs for a query. ProbLog gathers for each successful proof of the query the list of probabilistic facts the proof uses and compactly represents all proofs in a trie. Such a trie is then considered to be a sum of products (a disjunction of conjunctions of probabilistic facts). ROBDDs are used to solve the disjoint sum problem and to obtain the correct probability of the query.

The challenge is to find out how tabling can be combined with the ProbLog execution mechanism. Tabling mechanisms are available in XSB [11], YAP [12] and other Prolog systems. The basic idea is to collect the answers of a tabled subgoal in a table and, when the subgoal is re-encountered, to reuse the tabled answers instead of computing them. As a consequence of this memoization, tabling ensures termination of programs with the bounded term-size property, i.e. programs where the size of subgoals and answers produced during an evaluation is less than some fixed number. In the case of ProbLog, tabling answers is not sufficient, as the proofs are needed too. Also loops have to be dealt with correctly.

1998 ACM Subject Classification: I.2.2, I.2.8, D.1.6, G.3.
Key words and phrases: Tabling, Loop Detection, Probabilistic Logical Programming, ProbLog.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.124

The PRISM [14] assumptions such as exclusiveness and no loops imply that PRISM computations are simpler, in the sense that they do not have to deal with the disjoint sum problem. PRISM contains a linear tabling system [15]. Only when PRISM is executing its learning algorithms, is its tabling extended to do something special, namely to build support graphs which represent the shared structure of explanations for an observed goal. The support graphs play a central role in efficient EM learning of PRISM programs. The proofs ProbLog needs to compute are somewhat similar to these explanations.

This paper extends our early work [9]. The contributions of this paper are the identification of the necessary tabling mechanism for ProbLog programs and their implementation while respecting the current ProbLog optimizations, such as the exploitation of tries and the optimized translation of tries into ROBDDs.

ProbLog's motivating link discovery applications and other typical ProbLog programs have only ground goals. In order to table them, we represent the SLD proof tree as **nested tries**. We implemented a light-weight dedicated tabling for ground goals that supports nested tries and obtained impressive time improvements for some classes of programs. By the virtue of the nested tries, we also realize suffix sharing and thus a substantial memory compaction. By adding **loop detection**, path-finding programs, typical for link discovery, also benefit from the memoization.

The paper is structured as follows. First, we briefly introduce ProbLog and its relevant implementation details in Section 2. We present the nested tries and identify the necessary tabling support in Section 3. Transforming the nested tries to a sum of products efficiently is in Section 4. Section 5 contains the experimental evaluation. Related work and conclusions are in Sections 6 and 7 respectively.

2. ProbLog

ProbLog [4] is essentially an extension of Prolog where facts are labeled with probabilities[1] that they belong to a randomly sampled program. As such, a ProbLog program specifies a probability distribution over all its possible non-probabilistic subprograms. The success probability of a query is defined as the probability that it succeeds in such a random subprogram. ProbLog follows the distribution semantics [13]. We use the ProbLog program from Example 2.1 to describe how ProbLog calculates the exact probability of a query. The graph in Figure 1 is represented by the probabilistic facts `edge/2`. The rest of the program is normal Prolog code for finding a path in a graph. Consider the query `path(1,4)`.

Example 2.1 (Program path/2).

```
path(X, Y):- path(X, Y, [X]).
path(X, Y, P):- edge(X, Z), Y \== Z, \+ member(Z, P), path(Z, Y, [Z|P]).
path(X, Y, _):- edge(X, Y).
```

ProbLog first uses SLD-resolution to collect all the proofs of a query. Actually, ProbLog gathers for each successful proof of the query the list of probabilistic facts the proof uses. For the `path(1,4)` query, ProbLog collects eight successful proofs and for each proof a list of probabilistic `edge/2` facts as shown in Figure 2. ProbLog uses a trie to represent such a set of lists. The trie of a query is built during SLD-resolution: as soon as a successful proof is found, it is added to the trie. We use **proofs** for SLD-refutations as well as for lists of probabilistic facts. The trie for our example is given in Figure 2.

[1] Probabilistic facts are mutually independent random variables.

```
0.9::edge(1, 2).  0.9::edge(2, 1).  0.2::edge(5, 4).
0.4::edge(6, 5).  0.4::edge(5, 6).  0.2::edge(4, 5).
0.8::edge(2, 3).  0.8::edge(3, 2).  0.7::edge(1, 6).
0.5::edge(2, 6).  0.5::edge(6, 2).  0.7::edge(6, 1).
0.7::edge(5, 3).  0.7::edge(3, 5).
0.6::edge(3, 4).  0.6::edge(4, 3).
```

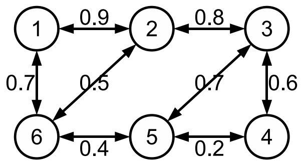


Figure 1: Graph of Example 2.1.

```
edge(1,2),edge(2,3),edge(3,5),edge(5,4)
edge(1,2),edge(2,3),edge(3,4)
edge(1,2),edge(2,6),edge(6,5),edge(5,3),edge(3,4)
edge(1,2),edge(2,6),edge(6,5),edge(5,4)
edge(1,6),edge(6,2),edge(2,3),edge(3,5),edge(5,4)
edge(1,6),edge(6,2),edge(2,3),edge(3,4)
edge(1,6),edge(6,5),edge(5,3),edge(3,4)
edge(1,6),edge(6,5),edge(5,4)
```

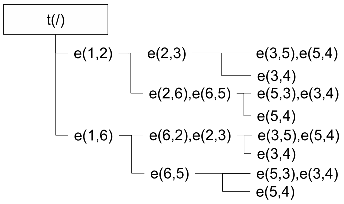


Figure 2: The successful proofs and the respective trie for *path(1,4)* of Example 2.1.

Now, think of the edge/2 facts as Boolean random variables indicating whether the facts are in the logic program. As a consequence, each proof corresponds to a conjunction of random variables and as a whole the trie represents a disjunction of conjunctions of probabilistic facts, also known as a sum of products. ProbLog needs to assess the probability of the sum of products, which has been shown to be #P-Hard [17]. ProbLog transforms the trie into a ROBDD in two steps. First, starting from the trie, a so-called ROBDD script is generated. Secondly, the ROBDD script is executed by a ROBDD package. From the ROBDD, ProbLog calculates the success probability (0.53864 for our example) of the query by a bottom-up dynamic programming algorithm over the ROBDD structure [8].

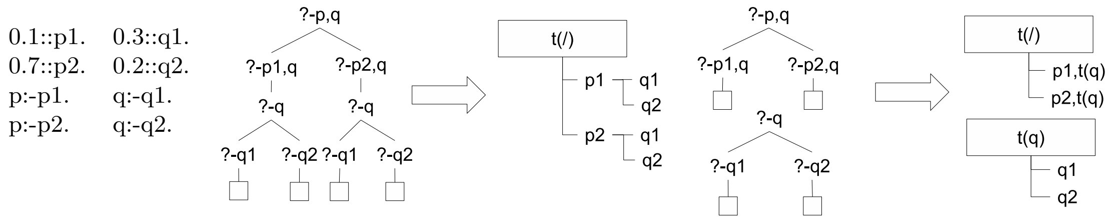
As argued in [8], representing the proofs compactly in a trie establishes prefix sharing and this turns out to be indispensable for the typical ProbLog mining applications. This prefix sharing can be clearly seen in Figure 2. In this paper we show how our tabling also establishes suffix sharing and thus further reduces the memory consumption.

3. Memoization by Tabling

Typical ProbLog goals are ground; indeed, in a probabilistic framework, one is interested in the probability that a goal can be proven, rather than in what the answers are. While our work can be generalized for non-ground goals, we focus on ground goals for this paper.

ProbLog collects all the SLD-refutations of a query as lists of probabilistic facts in a trie. Our tabling builds a forest of SLG-trees [3], one for the original query and one for each tabled subgoal. For each tabled goal, we need to memoize its contribution to the trie: we break up a single trie into a set of nested tries. The **nested** trie of a goal represents the successful proofs of the goal just as a normal trie does, but the parts of the trie that are contributed by other tabled subgoals are replaced by a reference to the trie of that subgoal.

Consider the query ?-p,q. for the program of Figure 3a. The SLD-tree and the corresponding trie are in Figure 3b. Tabling the predicate q/0 avoids recomputation during resolution and results in the forest of SLG-trees and the nested tries shown in Figure 3c. Note that the nested trie for the subgoal q is denoted by t(q) and t(/) denotes the trie of the topquery.



(a) Tabling q/0. (b) SLD-tree and trie. (c) SLG-trees and the nested tries.

Figure 3: Example of SLD-resolution and SLG-resolution trees and tries.

When ProbLog uses tabling while proving the topquery, it constructs a set of nested tries. This set of nested tries is equivalent with the trie that a non-tabled program would generate: it contains all the information about the complete successful proofs, the SLD-refutations, of the topquery. In the non-tabled evaluation, repeated subcomputations give rise to tries with repeated suffixes. The use of nested tries such as `t(q)` establishes suffix-sharing which reduces substantially the memory consumption of ProbLog.

Tabling uses a suspension/resumption mechanism to built a forest of SLG-trees. Tabling keeps in its tables an entry for each tabled subgoal that contains the nested trie of the subgoal. When ProbLog encounters the first instance of a tabled subgoal, the **generator** subgoal, tabling suspends the resolution of the parent goal, creates an entry for the subgoal with an empty nested trie, and starts proving it. As soon as a successful proof for the subgoal is found, it is added to its nested trie. Note that if the subgoal fails, no proof will be added and the nested trie will remain empty.

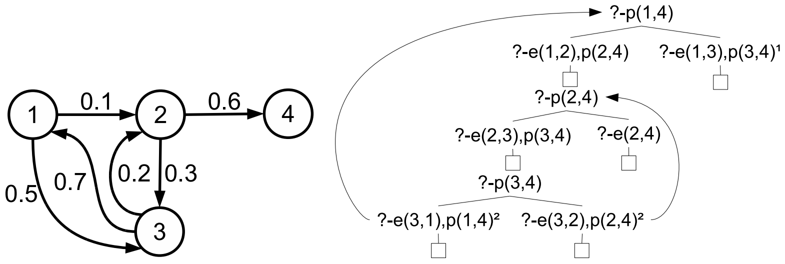
We eagerly collect the proofs for the generator goals. For programs without loops, tabling deals completely with a tabled subgoal before the parent goal is resumed and it is known whether the subgoal failed or succeeded. If a tabled subgoal fails, resumption of the parent goal fails its current proof. If the tabled subgoal succeeds then, on resumption of the parent goal, a reference to the nested trie of the tabled subgoal is added to the current proof of the parent goal. For subsequent occurrences of tabled subgoals, the **consumer** subgoals, tabling avoids recomputation by adding a reference to the appropriate nested trie in the current proof.

An attentive reader might indeed have noticed that the path program of Example 2.1 is a version that encodes loop detection explicitly by using `absent/2`. That version does not benefit from tabling as all the calls to `path` are different. The `path/2` program in Example 3.1 has no code to detect loops. We use this program and the graph of Figure 4a to explain how tabling should support loop handling in a probabilistic framework.

Example 3.1 (Program path/2 without loop detection)**.**

```
0.1::edge(1, 2).    0.5::edge(1, 3).    0.7::edge(3, 1).
0.3::edge(2, 3).    0.2::edge(3, 2).    0.6::edge(2, 4).
path(X, Y):- edge(X, Z), Y \== Z, path(Z, Y).
path(X, Y):- edge(X, Y).
```

For the query `path(1,4)` on this graph with the program of Example 2.1, we collect the following two proofs: `edge(1,2), edge(2,4)` and `edge(1,3), edge(3,2), edge(2,4)`. While proving the query `path(1,4)` with the program of Example 3.1, one will enter in infinite loops, such as the one between nodes 2 and 3 due to `edge(2,3)` and `edge(3,2)`. The



(a) Graph of Example 3.1. (b) SLG-trees with goal suspension.

Figure 4: Graph and SLG-trees of Example 3.1.

SLD-tree for `path(1,4)` is indeed infinite. The SLG-trees are in Figure 4b with backward arcs indicating the detected loops.

In the presence of loops, tabling normally uses a completion mechanism to ensure that all answers are returned to all consumers. For our ground ProbLog goals, this boils down to returning information about failure or success and in case of success the nested trie. A simple way to achieve this for consumer goals that give rise to a loop, is to assume that the generator goal will succeed and add the reference to the partially completed nested trie to the parent goal proof. Although a nested trie represents all the proofs for the subgoal, we do not need its final value to re-use it, as we put a reference to it in the other tries.

Now, our nested tries no longer only contain successful proofs. We optimistically assumed success for consumer goals giving rise to a loop. If none of the goals involved in the loop has a finite successful proof, they all fail. In this case, the nested tries contain references to failed subgoals. We deal with this during the ROBDD script generation step.

We have built a light-weight tabling prototype which allow us to experiment without having to change the ProbLog implementation. A program transformation is used to add tabling specific predicate calls that manipulate the extra tabling data structures. In this transformation we use findall to implement the eager proof collection for generator goals.

We implemented SLG tabling with suspension and resumption. As we have ground goals, we do not collect answers for goals, but their success or failure. Due to the probabilistic context, we need a special mechanism to construct a nested trie for each tabled subgoal, as we do not want to lose the prefix sharing required by ProbLog. Also note that ProbLog needs all the successful proofs, whereas clever normal tabling would stop after one successful proof for a ground goal.

For consumer goals that give rise to a loop, our optimistic approach assumes success of the goal, but in the end the goal might fail. Our nested tries then contain references to failed subgoals. This failure needs to be detected during the BDD script generation step.

4. Extracting the Boolean Formula from the Set of Nested Tries

Tabling computes for the original query a set of nested tries, which contain all the proofs of the query. As mentioned in Section 2, the ROBDD script generation step of ProbLog extracts all collected proofs from the Trie and generates a Boolean formula that expresses the proofs as a sum of products. In this section we are going to describe how to generate the Boolean formula from the set of nested tries. A naive implementation would have a performance cost similar to that of the non-tabled SLD resolution. We use dynamic programming to implement a top down traversal of the nested tries with loop detection. While generating the formula, we want to preserve the prefix sharing present in the tries and to exploit the suffix sharing. Dynamic programming enables the re-usage of completely unfolded tries, while re-usage of partially unfolded tries can be realized by introducing an ancestor subset check.

Starting from the nested trie of the original query, we reconstruct its proofs as a normal trie by unfolding the nested tries, but we also establish suffix sharing. To unfold a reference to another nested trie $t(g)$, we replace in the partial trie the reference by the nested trie itself. For each branch of the partial trie, we keep an **ancestor list**: initially it is empty and each time we unfold a reference to a $t(g)$, the g is added. In programs without loops, a nested trie $t(g)$ is unfolded only once, dynamic programming makes the other occurrences of references to $t(g)$ reuse the first unfolding and as such we now have tries with suffix sharing.

For programs with loops, we start from a finite representation of the infinite SLD-tree. The ancestor list enables us to detect loops during unfolding. Loops typically give rise to proofs that do not contribute to the final probability.

When unfolding detects a loop, we can prune the current proof: either because is a failing proof or because it gives rise to non-minimal proofs. If pruning results in an empty trie, which encodes failure, we can prune the parent branch. Figure 5 shows the nested tries, that contain infinite loops, for the query of Example 3.1.

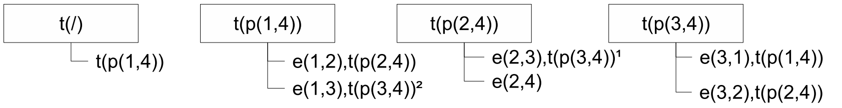


Figure 5: Nested tries of Example 3.1.

We have to be careful not to miss proofs because of the dynamic programming memoization. During unfolding, a nested trie typically occurs in different proofs. Its position in the SLD tree determines whether it gives rise to a loop or not. Consider $t(p(3,4))$ of Figure 5, suppose we first encounter its occurrence annotated with ^{1}. The two proofs this occurrence gives rise to, have loops and thus they are pruned. But, the second occurrence annotated with ^{2} belongs to a proof without a loop.

The $t(p(3,4))$ example shows that re-using results for nested tries that introduced loops, is not safe as different occurrences might give rise to different proofs. Actually, it depends on the context of the occurrence of the nested trie, in particular on the ancestor list of the occurrence. In order to improve suffix sharing, we start from the following observation. When two occurrences of a reference to a nested trie have exactly the same ancestor list, then obviously the unfolding of the references will introduce exactly the same loops. Generally, the occurrence of a goal will introduce at least the same loops as a previous occurrence, when the ancestor list of the previous occurrence is a subset of the current ancestor list.

5. Experiments

Our tabling directly interacts with the first two execution steps of ProbLog, SLD-resolution and ROBDD script generation. The third step is affected indirectly. It is well-known that ROBDD packages use heuristics while constructing a ROBDD and that their behaviour depends on the input, i.e. different inputs describing the same Boolean formula

	Memory		SLD/SLG		ROBDD Script		ROBDD Script	
	(Bytes)		resolution		generation		execution	
Day	non-tab	tab	non-tab	tab	non-tab	tab	non-tab	tab
1	352	912	0	0	0	0	5	5
2	1048	2148	0	0	0	0	5	5
13	184941472	15744	115400	2	2584	4	802	37
14	554824408	16980	380011	3	7938	4	2380	36
167	-	206088	-	25	-	89	-	9728
1600	-	1977276	-	262	-	3587	-	-

Table 1: Results for the weather program.

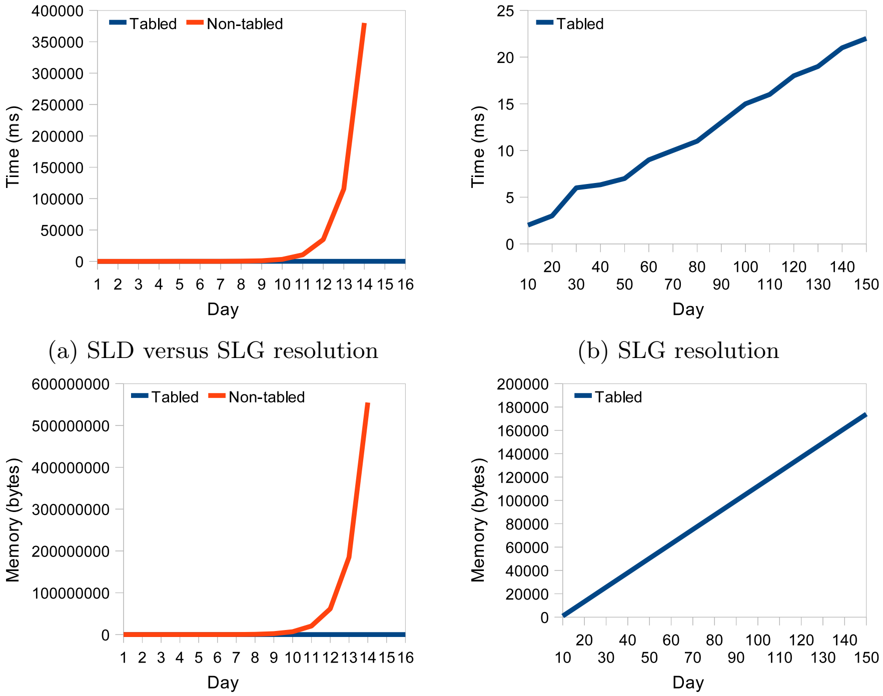
can give rise to different results and/or execution times. We address the following questions: (1) How does our tabling implementation perform both in time and in space for the SLD-resolution? (2) How do the nested tries compare with their flatten equivalents during the ROBDD script generation and the ROBDD script execution? (3) How do the nested tries with loops perform and what are the effects of the ancestor subset check?

Our benchmarks represent two typical categories of problems. Our **weather** benchmark is an example of a {Hidden} Markov Model (HMM). The value of the current time state depends on the value of the previous time state. It is well known that time series problems, when naively implemented, are of exponential complexity. Using tabling for this type of problems we expect significant improvement as memoization reduces the complexity of the problem's SLD-resolution to linear. The size of the weather problem is determined by the "Day" argument.

From the link discovery [4] applications, we took a **graph** benchmark, namely a number of graphs from the biomine database [16]. This benchmark expresses connections between various types of objects such as genes, proteins, tissues, etc and predicts relationships among them. We use the program of Example 2.1 for the non-tabled version and the program of Example 3.1 for the tabled version. For our experiments we used the first sample of graphs and the queries of [4]. The size is determined by the number of edges of the graph and the interconnectivity between the nodes. As these graphs are cyclic, loop handling is necessary.

All the experiments are done on an Intel^{R} Core^{TM}2 Duo CPU at 3.00GHz with 2GB of RAM memory running Ubuntu 8.04.2 Linux under a usual load. The reported times are the averages of five runs from which we dropped the best and worst, and all times are in milliseconds. For the SLD resolution and the ROBDD script generation, we used a time-out of 1 hour, while for the ROBDD script execution we used a 1 minute time-out (for our benchmarks most ROBDDs either will be built within one minute or run out of memory).

For **weather**, figure 6a shows that the SLD-resolution execution times of the non-tabled version are exponential with respect to the "Day" argument, while the tabled version is linear. Figure 6b shows the scaling of the SLD-resolution for queries that the non-tabled version fails to compute, as it exceeds the available memory. In Table 1, we see that the tabled version manages to compute "Day 167", while the non-tabled version stops at "Day 14". The tabled version is limited by the ROBDD script execution step that generates the ROBDD using a state-of-the-art ROBDD tool. For weather, the tabled version outperforms the non-tabled version in all stages including the ROBDD script execution. The memory usage to represent the proofs goes from exponential to linear for the tabled version as shown in Figure 6c. In Figure 6d we see the gain in memory is similar to the gain in time. This can be explained by the suffix sharing in the nested tries.



(a) SLD versus SLG resolution

(b) SLG resolution

(c) SLD versus SLG resolution

(d) SLG resolution

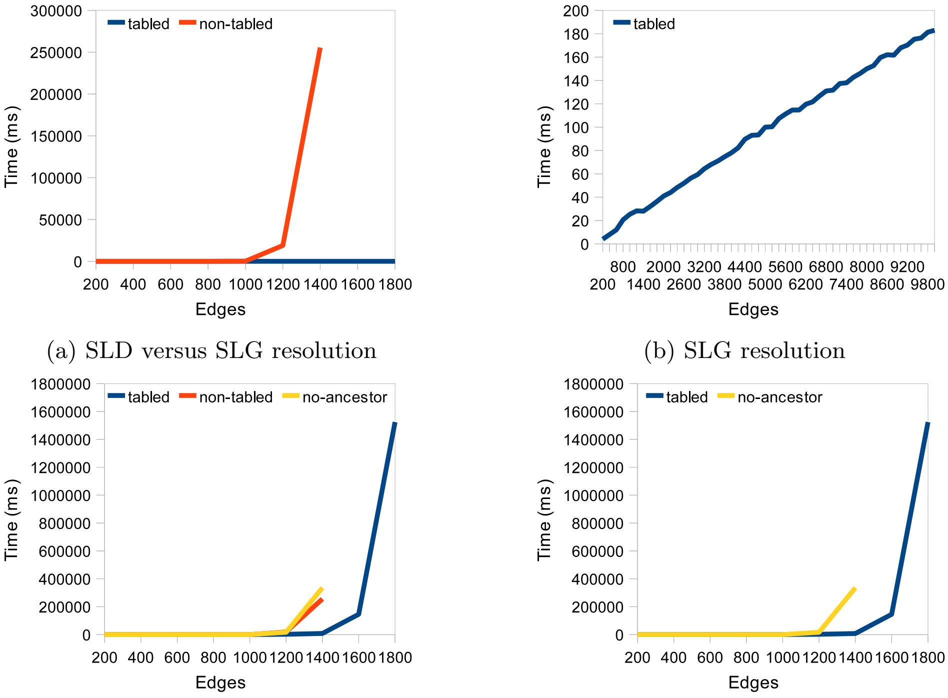
Figure 6: SLD/SLG-resolution times and memory consumption for the weather program.

In the graph benchmark, we study the benefits of tabling in combination with loop handling. As shown in Figure 7a, the tabled version has a significant performance improvement for the SLD-resolution. Figure 7b shows that increasing the number of edges in the graph affects the SLG-resolution linearly. Table 2 displays the results of the graph benchmark. The effect of tabling on the memory usage is a bit different now. Using nested tries for tabling favours suffix sharing rather than prefix sharing. It seems that in some graphs prefix sharing is more important for memory compaction than suffix sharing. However, in bigger graphs, the nested tries are again improving significantly memory consumption. That the tabled version requires to construct the nested tries even for goals that fail or succeed without probabilistic facts, introduces a significant minimum memory cost.

Unfortunately, we notice in Figure 7c that all the versions behave exponentially when summing up the SLD/SLG-resolution times and the ROBDD script generation times. Figure 7d presents the times for generating the ROBDD scripts from the nested tries with loops and the performance gain of the ancestor subset check. While the tabled version without the ancestor subset check underperformed the non-tabled in total time, the version with the ancestor subset check has significant performance gains in all cases.

6. Related Work

This paper investigates a dedicated form of tabling: memoization of all the proofs for ground goals in nested tries. Our work is similar to the PRISM [15, 14] tabling mechanism, as both mechanisms are restricted to grounded goals and both are memorising all the proofs.



(a) SLD versus SLG resolution

(b) SLG resolution

(c) SLD/SLG-resolution & ROBBD script generation

(d) ROBBD script generation

Figure 7: Graph program results.

Edges	Memory (Kilobytes)		SLD/SLG resolution		ROBBD Script generation			ROBDD Script execution		
	non-tab	*tab*	*non-tab*	*tab*	*non-tab*	*tab*	*no-anc*	*non-tab*	*tab*	*no-anc*
800	< 1	192	19	17	0	22	37	3	3	4
1000	32	239	245	20	1	107	258	36	33	37
1200	3303	286	18928	25	28	1982	16844	119	214	196
1400	39020	333	255546	28	473	7832	335853	455	454	278
1600	-	380	-	33	-	145669	-	-	26789	-
1800	-	426	-	37	-	1523948	-	-	-	-

Table 2: Results for the graph program.

One could compare ProbLog's nested tries with PRISM support graphs. The differences are that PRISM assumes the exclusiveness condition for the proofs, while ProbLog does not, and that ProbLog requires the handling of loops as it is intended for link discovery in graphs. Another example of tabling that needs the memoization of proofs, is in the scope of justification [6]. Note that tabling in justification keeps only one proof [10] instead of all the proofs, and that it requires tabling of non-grounded goals. Finally, [7] proposes tabling for another ProbLog inference method, namely Monte Carlo sampling.

7. Conclusions and Future Work

We successfully identified the requirements for a tabling mechanism for ProbLog and we realized a light-weight implementation of such a mechanism. Our experiments have shown that tabling is definitely beneficial for the SLD-resolution step, where the memory and time consumption can go from exponential to linear. Nested tries establish prefix and suffix sharing. We presented how loop handling can be performed using the nested tries. Tabling also affects the next ROBDD related steps of ProbLog. For benchmarks without loops, tabling further reduces the execution times, as these steps also benefit from the compaction by the sharing. In the graph benchmark, we see that tabling improves the overall performance of the system. While the improvement for SLD resolution is remarkable, the work is partly transferred to the ROBDD script generation step. In the presence of loops, the ancestor subset check is indispensable. We want to extend tabling for non-ground goals and use tabling for approximate inference methods.

Acknowledgements: This research is supported by GOA/08/008 “Probabilistic Logic Learning”.

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[3] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[4] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *Proceedings of IJCAI*, pages 2462–2467, 2007.

[5] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.

[6] Hai-Feng Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *Proceedings of ICLP*, pages 150–165, 2001.

[7] Angelika Kimmig, Bernd Gutmann, and Vítor Santos Costa. Trading memory for answers: Towards tabling ProbLog. In *Proceedings of SRL*, 2009.

[8] Angelika Kimmig, Vítor Santos Costa, Ricardo Rocha, Bart Demoen, and Luc De Raedt. On the efficient execution of ProbLog programs. In *Proceedings of ICLP*, pages 175–189, 2008.

[9] Theofrastos Mantadelis and Gerda Janssens. Tabling relevant parts of SLD proofs for ground goals in a probabilistic setting. In *Proceedings of CICLOPS*, 2009.

[10] Giridhar Pemmasani, Hai-Feng Guo, Yifei Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *LNCS: Logic Programming*, pages 500–501, 2003.

[11] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for effciently computing wfs. In *Proceedings of LPNMR*, pages 431–441, 1997.

[12] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. A Tabling Engine for the Yap Prolog System. In *Proceedings of AGP*, 2000.

[13] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of ICLP*, pages 715–729. MIT Press, 1995.

[14] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *JAIR*, 15:391–454, 2001.

[15] Taisuke Sato and Yoshitaka Kameya. Statistical abduction with tabulation. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 567–587, 2002.

[16] Petteri Sevon, Lauri Eronen, Petteri Hintsanen, Kimmo Kulovesi, and Hannu Toivonen. Link discovery in graphs derived from biological databases. In *Proceedings of DILS*, pages 35–49, 2006.

[17] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.