Preprocessing Boolean Formulae for BDDs in a Probabilistic Context

Theofrastos Mantadelis[1], Ricardo Rocha[2], Angelika Kimmig[1] and Gerda Janssens[1]

[1] Departement Computerwetenschappen, K.U. Leuven
Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium
{Theofrastos.Mantadelis,Angelika.Kimmig,Gerda.Janssens}@cs.kuleuven.be
[2] CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
ricroc@dcc.fc.up.pt

Abstract. Inference in many probabilistic logic systems is based on representing the proofs of a query as a DNF Boolean formula. Assessing the probability of such a formula is known as a #P-hard task. In practice, a large DNF is given to a BDD software package to construct the corresponding BDD. The DNF has to be transformed into the input format of the package. This is the preprocessing step. In this paper we investigate and compare different preprocessing methods, including our new trie based approach. Our experiments within the ProbLog system show that the behaviour of the methods changes according to the amount of sharing in the original DNF. The decomposition method is preferred when there is not much sharing in the DNF, whereas DNFs with sharing benefit from our trie based method. While our methods are motivated and applied in the ProbLog context, our results are interesting for other applications that manipulate DNF Boolean formulae.
Keywords: Boolean Formula Manipulation, Binary Decision Diagrams, ProbLog, Probabilistic Logic Learning.

1 Introduction

The past few years have seen a surge of interest in the field of Probabilistic Logic Learning (PLL) [1], also known as Statistical Relational Learning [2]. A multitude of formalisms combining logical or relational languages with probabilistic reasoning has been developed. One line of work, based on the distribution semantics [3], extends Logic Programming (LP) with probabilistic facts, i.e., facts whose truth values are determined probabilistically. Main representatives of this approach are PRISM [4], ICL [5] and ProbLog [6]. Even in such simple probabilistic logics, inference is computationally hard. As learning requires evaluating large amounts of queries, efficient inference engines are crucial for PLL.

The core of inference in these LP-based languages is a reduction to propositional formulae in Disjunctive Normal Form (DNF). Such a DNF describes all proofs of a query in terms of the probabilistic facts used, thus reducing probabilistic inference to calculating the probability of a DNF formula. The PRISM

system requires programs to ensure that the probability of the DNF corresponds to a sum of products. ProbLog has been motivated by a biological network mining task where this is impossible, and therefore obtains the probability from an external Binary Decision Diagram (BDD) tool. To this aim, the DNF is first constructed using logical inference, and then preprocessed into a sequence of BDD definitions which builds up the final BDD by applying Boolean operations on subformulae. While previous work on the efficient implementation of ProbLog has been focused on obtaining the DNF, little attention has been devoted to its further processing. However, as the performance of BDD construction depends on the size and structure of the intermediate BDDs and on the operations among them, the performance of this second phase is crucial for the overall performance of inference in ProbLog.

In this paper, we therefore study different approaches to preprocessing with special attention to the exploitation of repeated formulae to avoid redundant work in BDD construction. To this aim, we introduce a new data structure, named *depth breadth trie*, which facilitates detecting repeated subformulae in the DNF. This results in an improvement of the performance of ProbLog's preprocessing step. At the same time, this new data structure allows one to easily identify more shared subformulae, which can be used to further simplify BDD construction. A second contribution of this work is the implementation in ProbLog of an alternative preprocessing method, called *decomposition* [7], which we used to perform a comparative study of preprocessing methods in ProbLog. Our experimental results show that in structured problems, our trie based approaches are clearly outperforming the decomposition method, but in less structured problems decomposition seems to be better.

The remainder of the paper is organized as follows. First, Section 2 briefly introduces some background concepts about ProbLog, tries and BDDs. Next, Section 3 reviews different preprocessing methods. Then, we present our new approach in detail, including three new optimizations that can be performed with the depth breadth trie in Section 4. We present experimental results in Section 5 and end by outlining some conclusions in Section 6.

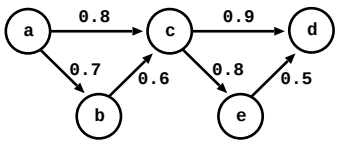
2 ProbLog

A ProbLog program T [6] consists of a set of labeled ground facts $p_i :: c_i$ together with a set of definite clauses. Each such fact c_i is true with probability p_i, i.e., these facts correspond to random variables, which are assumed to be mutually independent. Together, they define a distribution over subsets of $L_T = \{c_1, \ldots, c_n\}$. The definite clauses allow one to add arbitrary *background knowledge* (BK) to those sets of *logical* facts. Given the one-to-one mapping between ground definite clause programs and Herbrand interpretations, a ProbLog program also defines a distribution over its Herbrand interpretations.

Inference in ProbLog calculates the *success probability* $P_s(q|T)$ of a query q in a ProbLog program T, i.e., the probability that the query q is *provable* in a program that combines BK with a randomly sampled subset of L_T. Figure 1 shows

a ProbLog program encoding a probabilistic graph. The success probability of `path(a,d)` corresponds to the probability that a randomly sampled subgraph contains at least one of the four possible paths from node `a` to node `d`.

As checking whether a query is provable in each subprogram is clearly infeasible in most cases, ProbLog inference uses a reduction to a Boolean formula in DNF. This formula describes the set of programs where the query is provable. Variables in the formula correspond to probabilistic facts in the program, conjunctions correspond to specific proofs, and the entire disjunction to the set of all proofs. ProbLog then calculates the probability of that formula being true.




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

0.8 :: edge(a,c).   0.9 :: edge(c,d).
0.7 :: edge(a,b).   0.8 :: edge(c,e).
0.6 :: edge(b,c).   0.5 :: edge(e,d).
```

Fig. 1. A probabilistic graph and its encoding in ProbLog.

While the probability of a single conjunction, which represents the programs containing at least the facts used by the corresponding proof, is the product of the probabilities of these facts, it is impossible to simply sum the probabilities of conjunctions, as enumerating proofs does not partition the set of programs. Instead, we face the so called *disjoint-sum-problem*, which is known to be #P-hard [8]. By tackling this problem with (reduced ordered) BDDs [9], a graphical representation of Boolean formulae that enables probability calculation by means of dynamic programming, the ProbLog implementation scales to DNFs with tens of thousands of conjunctions.

ProbLog programs are executed in three steps. Given a ProbLog program T and a query q, the first step, *SLD-resolution*, collects all proofs for query q in $BK \cup L_T$. Proofs are stored as lists of identifiers corresponding to probabilistic facts in a *trie data structure*. This trie represents the DNF for query q. An essential property of the trie data structure is that common prefixes are stored only once, which in the context of ProbLog allows us to exploit natural prefix sharing of proofs, as two proofs with common prefix will branch off from each other at the first distinguishing probabilistic fact.

The second step, *preprocessing*, converts the DNF represented by the trie into a so-called *script*. A script is a sequence of BDD definitions, which define BDDs corresponding to Boolean random variables or Boolean formulae obtained by applying Boolean operators to previously defined BDDs. The last BDD defined in the script corresponds to the entire DNF.

Finally, the third step, *BDD construction*, follows the script to construct a sequence of intermediate BDDs leading to the final BDD for probability calculation. To this aim, ProbLog uses the front-end SimpleCUDD[3] for the BDD package CUDD[4].

The complexity of combining BDDs by Boolean operators is proportional to the product of their sizes, which depend on the variable order used by the BDD

[3] http://people.cs.kuleuven.be/~theofrastos.mantadelis/tools/simplecudd.html
[4] http://vlsi.colorado.edu/~fabio/CUDD/

package and can be exponential in the number of Boolean variables. As computing the order that minimizes the size of a BDD is a coNP-complete problem [9], BDD packages include heuristics to reduce the size by reordering variables. While reordering is often necessary to handle large BDDs, it can be quite expensive. To control the complexity of BDD construction, it is therefore crucial to restrain the size of the intermediate BDDs and the amount of operations performed. At the very least, preprocessing should aim to avoid repeated construction of BDDs for identical subformulae. In this work, we therefore use tries to exploit prefix sharing on the level of proofs as well as – by means of the new data structure depth breadth trie – on the level of BDD definitions.

3 From Tries to BDDs

In this section, we discuss the different approaches for preprocessing. Remember that preprocessing converts a DNF (represented as trie) to a script. We will use the example in Figure 1 as our running example. Figure 2 shows the set of proofs and the trie for the query `path(a,d)`. On top, the four proofs of the query are represented as conjunctions, where we use xy to denote the Boolean variable corresponding to probabilistic fact `edge(x,y)`. The disjunction of those conjunctions is depicted as a trie, where each branch of the trie corresponds to one conjunction. For simplicity of illustration, in the figures that follow, we will use the same xy notation. In scripts, we use n_i to refer to the i^{th} defined BDD.

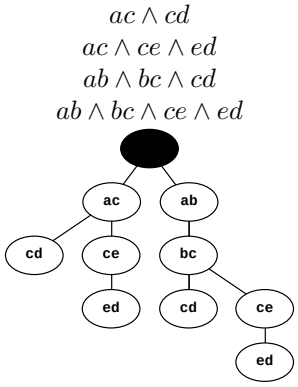


Fig. 2. Collected proofs and respective trie for `path(a,d)`.

The *naive method* directly mirrors the structure of the DNF by first constructing all conjunctions of the DNF formula and then combining those in one big disjunction. Figure 3 shows, on the left, the resulting script for the proofs of our example. The worst case complexity of this preprocessing step is $O(N \cdot M)$[5].

The *decomposition method* [7] recursively divides a Boolean formula in DNF into smaller ones until only one variable remains. To do so, it first chooses a Boolean variable from the formula, the so-called decomposition variable dv, and then breaks the formula into three subformulae. The first subformula f_1' contains the conjunctions that include dv, the second subformula f_2' those that include the negation of dv and the third subformula f_3 those that include neither of the two. Then, by applying the distribution axiom, the original Boolean formula can be re-written as $f = f_1' \vee f_2' \vee f_3 = (dv \wedge f_1) \vee (\neg dv \wedge f_2) \vee f_3$.

As all three new subformulae f_1, f_2 and f_3 do not contain dv, they can be decomposed independently. The most basic choice for the decomposition variable

[5] Complexity results for N proofs and M probabilistic facts. For more details see: https://lirias.kuleuven.be/bitstream/123456789/270070/2/complexity.pdf.

Naive Method	Decomposition Method	Recursive Node Merging
$n_1 = ac \wedge cd$	$n_1 = ce \wedge ed$	$n_1 = ce \wedge ed$
$n_2 = ac \wedge ce \wedge ed$	$n_2 = cd \vee n_1$	$n_2 = cd \vee n_1$
$n_3 = ab \wedge bc \wedge cd$	$n_3 = ce \wedge ed$	$n_3 = ac \wedge n_2$
$n_4 = ab \wedge bc \wedge ce \wedge ed$	$n_4 = cd \vee n_3$	$n_4 = bc \wedge n_2$
$n_5 = n_1 \vee n_2 \vee n_3 \vee n_4$	$n_5 = bc \wedge n_4$	$n_5 = ab \wedge n_4$
	$n_6 = ab \wedge n_5$	$n_6 = n_3 \vee n_5$
	$n_7 = ac \wedge n_2$	
	$n_8 = n_7 \vee n_6$	

Fig. 3. Scripts obtained by different preprocessing methods for the example DNF.

Algorithm 1 Recursive node merging. Takes a trie T representing a DNF and an index i and writes a script. REPLACE(T, C, n_i) replaces each occurrence of C in T by n_i.

function RECURSIVE_NODE_MERGING(T, i)
  if $\neg leaf(T)$ **then**
    $S_\wedge := \{(C, P) |$ leaf C is the only child of P in $T\}$
    for all $(C, P) \in S_\wedge$ **do**
      write $n_i = P \wedge C$
      $T :=$ REPLACE$(T, (C, P), n_i)$
      $i := i + 1$
    $S_\vee := \{[C_1, \ldots, C_n] |$ leaves C_j are all the children of some node P in $T, n > 1\}$
    for all $[C_1, \ldots, C_n] \in S_\vee$ **do**
      write $n_i = C_1 \vee \ldots \vee C_n$
      $T :=$ REPLACE$(T, [C_1, \ldots, C_n], n_i)$
      $i := i + 1$
    RECURSIVE_NODE_MERGING(T, i)

is the first variable of the current formula, however, various heuristic functions can be used as well, cf. [7]. All definitions resulting from the same decomposition step are written as a block at the end of that step, omitting those equivalent to $false$ to avoid unnecessary BDD operations. Figure 3 (middle column) again shows the result for our example query. The worst case complexity is $O(N \cdot M^2)$.

The approach followed in ProbLog, as described in [6], exploits the sharing of both prefixes – as directly given by the tries – and suffixes, which have to be extracted algorithmically. We will call this approach *recursive node merging*.

Recursive node merging traverses the trie representing the DNF bottom-up. In each iteration it applies two different operations that reduce the trie by merging nodes. The first operation (*depth reduction*) creates the conjunction of a leaf node with its parent, provided that the leaf is the only child of the parent. The second operation (*breadth reduction*) creates the disjunction of all child nodes of a node, provided that these child nodes are all leaves. Algorithm 1 shows the details for recursive node merging and Figure 4 illustrates its step-by-step application to the example trie in Figure 2. The resulting script can be found on the right in Figure 3.

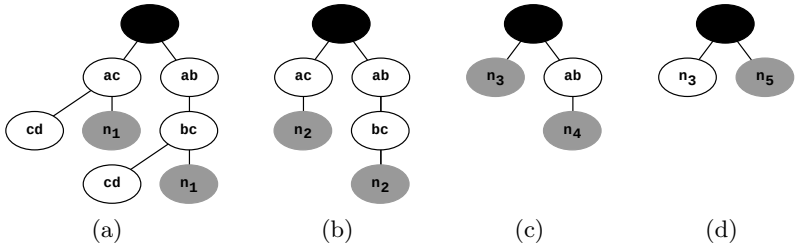


Fig. 4. Tries obtained during recursive node merging applied to the trie for `path(a,d)`.

For both reduction types, a subtree that occurs multiple times in the trie is reduced only once, and the resulting conjunction/disjunction is used for all occurrences of that subtree, thus performing some suffix sharing. Note however that the REPLACE() procedure can be quite costly when fully traversing the trie to search for repeated occurrences of subtrees.

4 Depth Breadth Trie

In this section, we introduce our new approach to implementing recursive node merging. The initial implementation explicitly performed the costly REPLACE() procedure of Algorithm 1. The new approach avoids this by storing all BDD definitions during recursive node merging. Once merging is completed, the script is obtained from this store. For each definition encountered during merging, we first check if it is already present in the store, and if so, reuse the corresponding reference n_i. As such a check/insert operation can be done in a single pass for tries, we introduce an additional and specific trie configuration for this purpose, that we named *depth breadth trie*. Apart from this improvement, the depth breadth trie has the additional advantage of allowing one to easily identify common prefixes on the level of BDD definitions, which was not possible before. As we will see, this leads to the definition of three new (optional) optimizations that can be performed during recursive node merging to further reduce the number of Boolean operations to be performed in BDD construction.

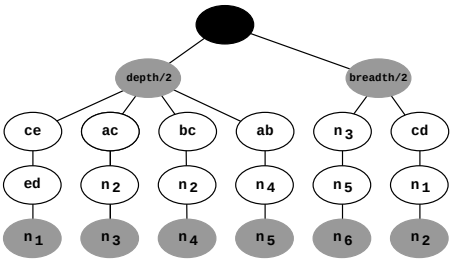
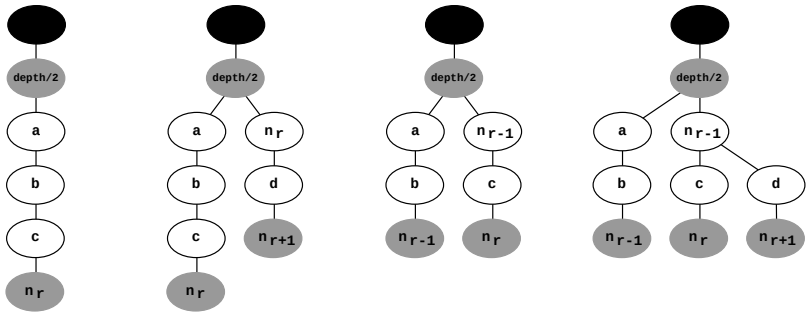


Fig. 5. Depth breadth trie containing a complete set of definitions (right column of Figure 3) for the DNF Boolean formula representing `path(a,d)`. Each leaf node contains a unique reference identifying the corresponding path's definition. Each branch in the depth/2 (breadth/2) part defines the conjunction (disjunction) of the entries in its white nodes (Boolean variables or definition references).

A depth breadth trie is divided in two parts corresponding to the two reduction types of recursive node merging: the *depth part* collects the conjunctions, the *breadth part* the disjunctions. This separation is achieved by two specific functors of arity two, `depth/2` and `breadth/2`. Their first argument is a Prolog list containing the literals that participate in the formula, the second argument is the unique reference n_i assigned to the corresponding BDD definition.

For example, the definitions $n1 = ce \wedge ed$ and $n2 = cd \vee n1$ are represented respectively by the terms `depth([ce,ed],n1)` and `breadth([cd,n1],n2)`. Note that reference `n1` introduced by the first term is used in the second term to refer to the corresponding subformula. At the same time, those references provide the order in which BDDs are defined in the script. Figure 5 shows the complete depth breadth trie built by recursive node merging for our example.

In the following, we introduce the three new optimizations that can be exploited with the depth breadth trie. The motivation is again to decrease the amount of operations performed in BDD construction. The optimizations are illustrated in Figure 6. Figure 6(a) presents the initial trie used in all cases, Figures 6(b), 6(c) and 6(d) show Optimizations I, II and III, respectively. The worst case complexity is $O(N \cdot M)$ in all cases.



(a) Initial trie (b) Adding $[a, b, c, d]$ (c) Adding $[a, b]$ (d) Adding $[a, b, d]$

Fig. 6. Examples of definitions that trigger (b) Optimization I, (c) Optimization II and (d) Optimization III when added to the depth breadth trie in (a).

Optimization I (Contains Prefix): The first optimization occurs when a new formula $[p_1, \ldots, p_n]$ to be added to the depth breadth trie contains as prefix an existing formula $[p_1, \ldots, p_i]$, $i \geq 2$, with reference n_r. In this case, the existing formula will be reused and, instead of inserting $[p_1, \ldots, p_n]$, we will insert $[n_r, p_{i+1}, \ldots, p_n]$ and assign a new reference to it.

Optimization II (Is Prefix): The second optimization considers the inverse case of the first optimization. It occurs when a new formula $[p_1, \ldots, p_i]$, $i \geq 2$, to be added to the depth breadth trie is a prefix of an existing formula $[p_1, \ldots, p_n]$ with reference n_r. In this case, we split the existing subformula representing $[p_1, \ldots, p_n]$ in two: the first one representing the new formula

Algorithm 2 Depth breadth trie optimizations. Takes a T representing either the depth or breadth part of the depth breadth trie and a list L with the formula to be added and returns the reference n_i assigned to L in T. $COUNTER$ is a global counter and REPLACE(L, C, n_i) replaces C in L by n_i.

```
function UPDATE_DEPTH_BREADTH_TRIE(T, L)
  if (L, n_i) ∈ T then
    return n_i
  for all (list, n_i) ∈ T do
    if list is prefix of L then
      /* Optimization I */
      L := REPLACE(L, list, n_i)
      return UPDATE_DEPTH_BREADTH_TRIE(T, L)
    if L is prefix of list then
      /* Optimization II */
      T := REMOVE((list, n_i), T)
      T := ADD((L, n_{i-(length(list)-length(L))}), T)
      list := REPLACE(list, L, n_{i-(length(list)-length(L))})
      T := ADD((list, n_i), T)
      return n_{i-(length(list)-length(L))}
    if L and list have a common prefix prefix with length(prefix) > 1 then
      /* Optimization III */
      n_j := UPDATE_DEPTH_BREADTH_TRIE(T, prefix)
      L := REPLACE(L, prefix, n_j)
      return UPDATE_DEPTH_BREADTH_TRIE(T, L)
  COUNTER := COUNTER + length(L)
  T := ADD ((L, n_COUNTER), T)
  return n_COUNTER
```

$[p_1, \ldots, p_i]$ with a new reference n_{r-1}, the other representing the existing formula, but modified to re-use the new reference n_{r-1}, i.e., $[p_1, \ldots, p_n]$ is replaced by $[n_{r-1}, p_{i+1}, \ldots, p_n]$.

Optimization III (Common Prefix): The last optimization exploits definitions branching off from each other. It occurs when a new formula $[p_1, \ldots, p_n]$ shares a common prefix $[p_1, \ldots, p_i]$, $n > i \geq 2$, with an existing formula $[p_1, \ldots, p_i, p'_{i+1}, \ldots, p'_m]$, $m > i$, with reference n_r. In this case, first, the common prefix is inserted as a new formula with reference n_{r-1}, triggering the second optimization, and second, the original new formula is added as $[n_{r-1}, p_{i+1}, \ldots, p_n]$ using n_{r-1} as in the first optimization.

Each repeated occurrence of a prefix of length P identified by one of the optimizations decreases the total number of operations required by $P - 1$. For example, if Optimization III identifies a common prefix f_P of length P of two formulae f_M and f_N of length M and N respectively, the number of operations decreases from $(N-1)+(M-1)$ to $(P-1)+(N-P)+(M-P) = N+M-P-1$, and if a third formula f_K shares the same prefix, the number of operations it requires again reduces to $(K-1)-(P-1) = K-P$.

Algorithm 2 formalizes the implementation of these three optimizations, which roughly speaking replaces the write and replace operations in Algorithm 1. One should notice that with the depth breadth trie, the references n_i are no longer incremented by one but by the length of the formula being added. This is necessary as Optimizations II and III insert additional subformulae that have to be created before the current formula being added, and thus need to be assigned a smaller reference. As our formulae always contain at least two elements, using the length of the formula to increment n_i is sufficient to leave enough free places for later use with Optimizations II and III. The order given by those references will therefore ensure that subformulae will be generated before being referred to. Moreover, as we are using tries, these optimizations can be performed while adding the new formulae. Note that the optimizations require a modification of the trie insertion procedure[6]: if the new definition first differs from an existing one after two or more steps, the insertion of the new formula is frozen while the appropriate optimization is performed and resumed afterwards.

To assess the effect of optimizations, our implementation in fact offers four choices of optimization level: no optimizations, Optimization I only, Optimizations I and II, or all three optimizations. Furthermore, the minimal length of common prefixes (2 by default) can be adapted. Note that depending on the order in which the formulae are inserted, different optimizations might trigger and the resulting trie might be slightly different.

5 Experimental Results

We next report on experiments comparing the four preprocessing methods: naive, decomposition (**dec**), recursive node merging as described in [6] (**rnm**) and recursive node merging with the depth breadth trie (**dbt**). The environment for our experiments was a C2Q 2.83 GHz 8 GB machine running Linux using a single core. The entire ProbLog engine, including preprocessing, is implemented in Yap Prolog 6.0, except for **dbt**, which is implemented in C as its optimizations require modifications to Yap's trie insertion, which is itself implemented in C. BDD construction uses the CUDD BDD package with automatic triggering of variable reordering by group sifting [10]. As **rnm** has been developed to exploit structure sharing in the trie, whereas **dec** is a general purpose method, we consider two benchmarks that contrast in this aspect.

The first benchmark is a three-state Markov model, where we query for the probability of an arbitrary sequence of N steps (starting in a random state at time point 0) ending in a given state. Each of the N time steps in such a sequence involves two new random variables (jointly encoding the three different start states of the step), and the number of proofs is thus 3^N.

[6] A definition is inserted term by term incrementally and each term is compared for identical prefix.

The second benchmark comes from the domain of connectivity queries in biological graphs that originally motivated ProbLog. In this case, we consider two different ProbLog inference methods which lead to different types of formulae. Exact inference as discussed in Section 2 produces formulae with high sharing in both prefixes and suffixes of proofs (which correspond to paths starting and ending at specific nodes). On the other hand, upper bound formulae as encountered in bounded approximation [6] typically contain small numbers of proofs and large numbers of so-called *stopped derivations*, i.e., partial proofs cut off at a probability threshold. While the latter still share prefixes, their suffixes are a lot more diverse. This type of formulae has been observed to be particularly hard for ProbLog. In our experiments, we use a graph with 144 edges (and thus 144 random variables) extracted from the Biomine network also used in [6], and query for acyclic paths between given pairs of nodes. We use a set of 45 different queries, some chosen randomly, and some maximizing the degrees of both nodes. In exact inference, the number of conjunctions in the DNF ranges from 13136 to 351600, with an average of 90127, for upper bound formulae, it ranges from 53 to 26085, with an average of 9516. The number of trie nodes in the exact case varies from 44479 to 1710621 (average 387073), representing between 161100 and 5776734 virtual nodes (average 1376030)[7]. In the upper bound case, tries have 126 to 60246 nodes (average 22251), corresponding to 334 to 232618 virtual nodes (average 80525).

N	**naive**	**dec**	**rnm**	**dbt**
7	251	45	28	25
8	786	87	28	25
9	3,698	188	31	25
10	20,854	539	35	29
11	256,330	1,638	43	31
12	/	5,024	96	31
13	/	-	205	48
14	/	-	-	75

Table 1. Average runtimes for BDD construction on three-state Markov model, for sequence length N. Cases with (/) exceeded the time limit for BDD construction, cases with (−) fail for memory reasons.

We set up experiments to study the following questions:

Q1: How do the different preprocessing methods compare on more, or less structured problems?

Q2: What is the impact of each optimization and which parameters affect them?

As the main goal of this work is to optimize the performance of BDD construction, we will focus on the runtime of this last step of ProbLog inference as central evaluation criterion. The time to calculate probabilities on the final BDD is included in the construction time; it typically is a very small fraction thereof.

Table 1 presents BDD construction times for the Markov model. In this benchmark, no extra optimizations are triggered for **dbt**. Times are given in milliseconds and are averages over three runs. BDD construction uses a timeout of 600 seconds, when a method reaches this timeout, it is not applied to larger problems. These cases are marked with (/). Cases marked with (-) fail due to

[7] The number of virtual nodes roughly corresponds to the number of occurrences of Boolean variables in the DNF written in uncompressed form. Each trie branch is represented by a node for each variable and by two special start and end nodes.

memory during script preprocessing; this also occurs when using **dbt** at length 15. In this experiment, both trie based methods clearly outperform the naive and **dec** methods, and BDD construction also seems to benefit slightly from the modifications used in the **dbt**-based version of **rnm**. As a first answer to **Q1**, we thus conclude that for structured problems, trie-based methods are indeed the first choice to optimize BDD construction.

For the graph domain, we use a timeout of 300 seconds on BDD construction, and a cutting threshold $\delta = 0.05$ for obtaining upper bound formulae. As the naive method always performs worst, it is excluded from the following discussion. Typically, all optimizations for the **dbt** method are triggered, with type I being most frequent (note that type III increases type I usage).

Method	Preprocessing		BDD Constr	
	avg	sdev	avg	sdev
dec	40,076	38,417	2,235	1,313
rnm	3,694	3,632	1,844	1,150
dbt	124	117	1,998	1,318
dbt1	125	118	1,891	1,697
dbt2	125	118	1,481	630
dbt3	128	120	1,446	769

Table 2. BDDs for exact inference: average and standard deviation of times over 45 queries (**dbt**i uses all optimization levels $l \leq i$).

In exact inference, cf. Table 2, BDD construction times for all methods are very close and rarely exceed 4 seconds. However, preprocessing time for **dec** is one order of magnitude higher than for **rnm** (remember that **dbt** should not directly be compared, as it is implemented in a different language). Again, this is due to the high amount of suffix sharing in those tries, which is exploited by our method, but causes repeated work during construction for the decomposition method. These results clearly enforce our first conclusions about **Q1**. Regarding **Q2**, these results show that the **dbt** optimizations are incrementally effective in reducing construction time without introducing costs in preprocessing time.

Group	Method	BDD Constr		Time
		avg	sdev	outs
Easy 19/44	**dec**	9,351	2,323	0
	rnm	24,710	6,415	0
	dbt	10,148	2,192	0
	dbt1	10,714	2,389	0
	dbt2	14,417	3,263	0
	dbt3	15,311	4,055	0
Medium 14/44	**dec**	21,785	5,197	0
	rnm	46,428	9,084	0
	dbt	29,719	4,029	0
	dbt1	39,914	9,084	0
	dbt2	28,522	3,165	0
	dbt3	46,263	19,231	0
Hard 11/44	**dec**	28,979	9,172	0
	rnm	114,870	18,225	0
	dbt	62,612	16,350	3
	dbt1	121,442	29,052	2
	dbt2	94,454	28,753	3
	dbt3	122,150	37,751	3

Table 3. BDDs for upper bounds at threshold 0.05: average and standard deviation of times over 44 queries grouped into categories according to runtimes.

For upper bound BDDs, the results are more diverse. Here, we focus on the comparison between **dec** and **dbt** with different optimiza-

tion levels. For presentation of results in Table 3, we partition queries in categories by using two thresholds on BDD construction time ($t < 15,000ms$ for **Easy**, $15,000ms \leq t < 50,000ms$ for **Medium** and $t \geq 50,000ms$ for **Hard**), and majority vote among the methods (as one single test query finishes in few milliseconds, it is omitted from the results). The last column gives the number of queries reaching the timeout in BDD construction.

Upper bound DNFs contain less conjunctions than those obtained in exact inference, and preprocessing times are one order of magnitude lower for all methods. BDD construction times, however, are generally higher when considering upper bounds. On average, BDDs obtained by **dec** have smaller construction times than those obtained from **rnm** and **dbt**, even though variation is high.

Table 4 compares methods by counting the number of queries for which they achieve fastest upper bound BDD construction compared to competing methods. As **dec** performs best in the overall comparison for this type of problem, we further compare the two implementations of recursive node merging. While the BDDs obtained from the implementation using depth breadth tries often outperform those from the previous implementation, there is no clear winner between the various optimization levels for this type of problem. Together, those results provide the second part of the answer to **Q1**: for problems with less suffix sharing, scripts obtained from **dec** often outperform those obtained from **dbt**.

Method	**All**	**rnm/dbt**	**rnm/dbt***	**dbt***
dec	26	-	-	-
rnm	2	14	6	-
dbt	6	30	13	16
dbt1	2	-	9	10
dbt2	5	-	9	9
dbt3	3	-	7	9

Table 4. Upper bound BDDs: number of queries where a given method leads to fastest BDD construction, comparing all methods (**All**), the two implementations of recursive node merging without optimizations (**rnm/dbt**) or including different optimization levels (**rnm/dbt***), and different depth breadth trie optimization levels only (**dbt***).

Concerning the optimization levels, Tables 3 and 4 indicate that for the graph case, their effect varies greatly. For all levels, we observe cases of improvement as well as deterioration. We suspect that optimizations are often performed too greedily. Initial experimentation on artificially created formulae indicates that several factors influence performance of optimizations, among which are: (i) the length of the shared prefix; (ii) the number of times it occurs; (iii) the structure of the subformulae occurring in the prefix; and (iv) the structure of the suffixes sharing the prefix. While the latter three are harder to control during preprocessing, we performed a first experiment where we only trigger the optimizations for shared prefixes of minimal length $n > 2$. Results on upper bound formulae confirm that this parameter indeed influences BDD construction, again to the better or the worse. We conclude that, while we identified certain parameters influencing the success of optimizations in synthetic data, in the case of less regular data, the answer to **Q2** remains an open issue for further investigation.

6 Conclusions and Future Work

We introduced depth breadth tries as a new data structure to improve preprocessing in ProbLog, and compared the resulting method and its variations to the method used so far as well as to the decomposition method presented by [7]. Our experiments with the three-state Markov model and with exact inference confirm that our trie based method outperforms the other methods on problems with high amount of suffix sharing between proofs. At the same time they reveal that the decomposition method is more suited if this is not the case, and thus is a valuable new contribution to ProbLog.

While the three new optimizations, aimed at reducing the number of BDD operations, can greatly improve performance in some cases, in others, they have opposite effects. Initial experiments suggest that those optimizations should be applied less greedily. Future work therefore includes a more in depth study of the factors influencing the effects of optimizations. We also plan to further investigate the respective strengths of our trie based approach and the decomposition method, and to exploit those in a hybrid preprocessing method. Finally, we need to further explore existing work on BDD construction in other fields, which might provide valuable insights for our specific application context.

Acknowledgments T. Mantadelis is supported by the GOA/08/008 Probabilistic Logic Learning, A. Kimmig is supported by the Research Foundation Flanders (FWO Vlaanderen) and R. Rocha has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and HORUS (PTDC/EIA-EIA/100897/2008).

References

1. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S., eds.: Probabilistic Inductive Logic Programming. Volume 4911 of LNCS. (2008)
2. Getoor, L., Taskar, B., eds.: Statistical Relational Learning. The MIT press (2007)
3. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Proceedings of ICLP. (1995) 715–729
4. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. JAIR **15** (2001) 391–454
5. Poole, D.: The independent choice logic and beyond. [1] 222–243
6. Kimmig, A., Santos Costa, V., Rocha, R., Demoen, B., De Raedt, L.: On the efficient execution of ProbLog programs. In: Proceedings of ICLP. (2008) 175–189
7. Rauzy, A., Châtelet, E., Dutuit, Y., Bérenguer, C.: A practical comparison of methods to assess sum-of-products. Reliab Eng Syst Safe **79**(1) (2003) 33 – 42
8. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing **8**(3) (1979) 410–421
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8) (1986) 677–691
10. Panda, S., Somenzi, F.: Who are the variables in your neighborhood. In: Proceedings of ICCAD. (1995) 74–77