Variable Compression in ProbLog

Theofrastos Mantadelis, Gerda Janssens

Report CW 586, May 2010

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Variable Compression in ProbLog

Theofrastos Mantadelis, Gerda Janssens

Report CW 586, May 2010

Department of Computer Science, K.U.Leuven

Abstract

The paper identifies patterns of Boolean variables that occur in Boolean formulae, namely AND-clusters and OR-clusters. We give a polynomial algorithm that detects AND-clusters in disjunctive normal form (DNF) Boolean formulae, or OR-clusters in conjunctive normal form (CNF) Boolean formulae. Furthermore, we explain how to exploit the clusters in the context of ProbLog.

In ProbLog, Boolean formulae are used to express how the probability of a query depends on the probabilistic part of a ProbLog program. Boolean formulae in ProbLog are represented by Reduced Ordered Binary Decision Diagrams (ROBDD). Depending on the Boolean formula, the generation of a ROBDD can be very costly. In this paper we present how to compress clusters in Boolean formulae and make the generation of the ROBDD more efficient without affecting the probability of the query.

We did an experimental evaluation of the effects of AND-cluster compression for a real application of ProbLog. With our prototype implementation we have a significant improvement in performance (up to 87%) for the generation of ROBDDs.

Keywords : ProbLog, Statistical Relation Learning, Probabilistic Logic Programming, Variable Compression, Binary Decision Diagrams
CR Subject Classification : I.2.2, G.3, D.1.6, F.1.2

Variable Compression in ProbLog

Theofrastos Mantadelis, Gerda Janssens
Departement Computerwetenschappen, K.U. Leuven
Celestijnenlaan 200A - bus 2402, B-3001 Heverlee, Belgium
{firstname.lastname}@cs.kuleuven.be

ABSTRACT

The paper identifies patterns of Boolean variables that occur in Boolean formulae, namely AND-clusters and OR-clusters. We give a polynomial algorithm that detects AND-clusters in disjunctive normal form (DNF) Boolean formulae, or OR-clusters in conjunctive normal form (CNF) Boolean formulae. Furthermore, we explain how to exploit the clusters in the context of ProbLog.

In ProbLog, Boolean formulae are used to express how the probability of a query depends on the probabilistic part of a ProbLog program. Boolean formulae in ProbLog are represented by Reduced Ordered Binary Decision Diagrams (ROBDD). Depending on the Boolean formula, the generation of a ROBDD can be very costly. In this paper we present how to compress clusters in Boolean formulae and make the generation of the ROBDD more efficient without affecting the probability of the query.

We did an experimental evaluation of the effects of AND-cluster compression for a real application of ProbLog. With our prototype implementation we have a significant improvement in performance (up to 87%) for the generation of ROBDDs.

Keywords

ProbLog, Statistical Relation Learning, Probabilistic Logic Programming, Variable Compression, Binary Decision Diagrams

1. INTRODUCTION

ProbLog [4, 5] is a probabilistic framework that extends Prolog with probabilistic facts. ProbLog computes the probability of a query in two main steps. First, ProbLog collects the probabilistic facts for each SLD proof of the query. Each probabilistic fact is represented by a Boolean variable, each proof by the conjunction of probabilistic facts used in the proof, and the set of all proofs by a disjunction of conjunctions (a DNF). In the second step, ProbLog uses ROBDDs [1, 3] to calculate the success probability of the query. Note that assessing the probability of a DNF Boolean formula is a #P-complete problem and using ROBBDs is a state-of-art approach [8].

For typical ProbLog applications, generating a ROBDD can become one of the limiting factors. State-of-art ROBDD packages use heuristics to find good variable orderings that construct smaller ROBDDs for a Boolean formula. As the size of the ROBDDs is important for the performance, we present in this paper a complementary optimisation that allows us to construct smaller ROBDDs.

We observed patterns (AND-clusters, OR-clusters) in the ROBDDs that make it possible to replace a set of Boolean variables with a single new one. In order for the ROBDD generation to benefit from the compression based on clusters, the clusters should be discovered before the actual generation. The paper has two main contributions. The first contribution is the definition of the AND-clusters and OR-clusters and their usage in assessing the probability of a DNF Boolean formula. The second contribution is the Book Marking algorithm that detects AND-clusters in ProbLog set of proofs.

We also evaluate experimentally the effects of the AND-clusters in a typical ProbLog application [5, 10]. Our experiments proof the impact of the AND-cluster compression: the number of variables in the ROBDD is on average reduced with 28% and the performance of the generation of the ROBDDs improves on average with 41%. Computing the AND-clusters can be done in parallel with the SLD resolution itself. Our prototype implementation is still sequential but allows us to verify the potential of the AND-clusters.

We introduce ProbLog in Section 2 and also explain how the ROBDDs are used. In Section 3 we define AND-clusters, OR-clusters and present the algorithm to detect them. The experiments follow in Section 4 and the complexity analysis is in Section 5. A setting for a parallel exploitation is sketched in Section 6. Section 7 concludes.

2. PROBLOG AND ITS USE OF ROBDDS

2.1 The ProbLog Language

A ProbLog program T [7] consists of a set of labelled ground facts $p_i :: pf_i$ together with a set of definite clauses. Each such fact pf_i is true with probability p_i, that is, these facts

```
0.5::edge(1, 2). % x0
0.4::edge(1, 4). % x1
0.7::edge(2, 3). % x2
0.8::edge(2, 6). % x3
0.9::edge(4, 5). % x4
0.7::edge(5, 2). % x5
0.6::edge(5, 7). % x6
0.4::edge(6, 3). % x7
0.3::edge(6, 7). % x8
path(X, Y):- path(X, Y, [X]).
path(X, Y, _A):- edge(X, Y).
path(X, Y, A):-
  edge(X, Z), absent(Z, A),
  path(Z, Y, [Z|A]).
absent(_, []).
absent(X, [Y|Z]):- X \= Y, absent(X, Z).

% Query: problog_exact(path(1, 3), Probability)
```

Figure 1: Example ProbLog program path/2.

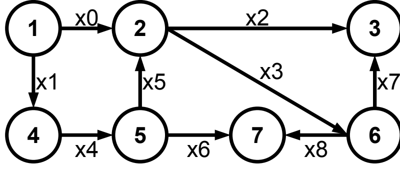


Figure 2: Graph of example program path/2.

correspond to random variables, which are assumed to be mutually independent. Together, they thus define a distribution over subsets of $L_T = \{pf_1, \ldots, pf_n\}$. 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.

ProbLog inference calculates the *success probability* $P_s(q|T)$ of a query q in a ProbLog program T, that is, the probability that the query q is *provable* in a logic program that combines BK with a randomly sampled subset of L_T.

Figure 1 shows a small ProbLog program encoding a probabilistic graph. The success probability of `path(1,3)` corresponds to the probability that a randomly sampled subgraph contains at least one of the four possible paths from node `1` to node `3`. The graph in Figure 2 is represented by the probabilistic facts `edge/2`.

2.2 Program Execution in ProbLog

ProbLog programs are executed in two 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 probabilistic facts in a *trie data structure*. This trie represents the proofs of the query q in a compact way as it exploits prefix sharing between proofs. The usage of tries is not important for this paper.

The query `path(1,3)` triggers the SLD resolution of the `path/2` predicate collecting all the proofs of the query. In this example only the `edge/2` facts are collected. SLD resolution finds four proofs which are represented by the following lists of probabilistic facts:

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

In general these lists of probabilistic facts express the Boolean formula:

$$\bigvee_{pr_j \in proofs} \left(\bigwedge_{pf_i \in pr_j} pf_i \right) \quad (1)$$

where the pf_i represent the probabilistic facts used in proof pr_j. Using the x_i to represent the `edge/2` facts as indicated in the ProbLog program, the DNF for the `path(1,3)` query is the formula $(x_0 \wedge x_2) \vee (x_0 \wedge x_3 \wedge x_7) \vee (x_1 \wedge x_4 \wedge x_5 \wedge x_2) \vee (x_1 \wedge x_4 \wedge x_5 \wedge x_3 \wedge x_7)$. In order to compute the correct probability for (1), ProbLog faces the disjoint sum problem. ProbLog solves this in its second step, namely by the transformation of the disjunction of conjunctions into mutually disjoint conjunctions by constructing a ROBDD for (1). ProbLog uses the frontend SimpleCUDD[1] for the ROBDD package CUDD[2].

Consider the simple formula $(pf_1 \wedge pf_2) \vee (pf_2 \wedge pf_3))$. The probability of this formula is equal to $P(pf_1) \cdot P(pf_2) + (1 - P(pf_1)) \cdot P(pf_2) \cdot P(pf_3)$. The ROBDD for the formula $(pf_1 \wedge pf_2) \vee (pf_2 \wedge pf_3)$ is in Figure 3. The topmost circular node in the ROBDD corresponds to the probabilistic fact pf_2 and is called a variable node. A variable node has two successors pointed to by the **"high"** edge and the **"low"** edge. Edges are implicitly directed: they point downwards. The ROBDD that is rooted at the low successor represents the Boolean expression that is yielded by substituting "false" for the variable. The high successor represents the Boolean expression that is yielded by substituting "true". The "true" node 1 and the "false" node 0 represent whether the binary formula is satisfied or not.

The probability of a ROBDD node is calculated bottom-up and is equal to the probability of the node variable multiplied with the probability of the "high" successor plus one minus the probability of the node variable multiplied with the probability of the "low" successor: $P_{node} = P_{V_{node}} \cdot P_{highsuccessor} + (1 - P_{V_{node}}) \cdot P_{lowsuccessor}$. The "true" node has probability 1 and the "false" node has probability 0. A more detailed explanation can be found in [7]. For the above simple example, the probability is calculated as follows:

$$\begin{aligned}
&P(node_{pf_2}) \\
&= P(pf_2) \cdot P(node_{pf_1}) + (1 - P(pf_2)) \cdot 0 \\
&= P(pf_2) \cdot (P(pf_1) \cdot 1 + (1 - P(pf_1)) \cdot P(node(pf_3))) \\
&= P(pf_2) \cdot (P(pf_1) + (1 - P(pf_1)) \cdot (P(pf_3) \cdot 1 + \\
&\qquad (1 - P(pf_3)) \cdot 0)) \\
&= P(pf_2) \cdot (P(pf_1) + (1 - P(pf_1)) \cdot P(pf_3)) \\
&= P(pf_1) \cdot P(pf_2) + (1 - P(pf_1)) \cdot P(pf_2) \cdot P(pf_3) \ .
\end{aligned}$$

[1] `http://www.cs.kuleuven.be/~theo/tools/simplecudd.html`

[2] `http://vlsi.colorado.edu/~fabio/CUDD/`

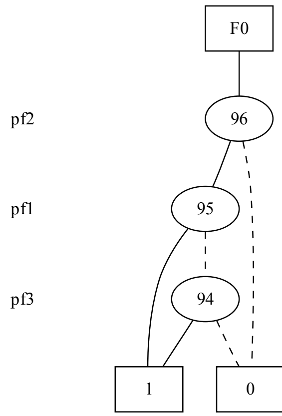
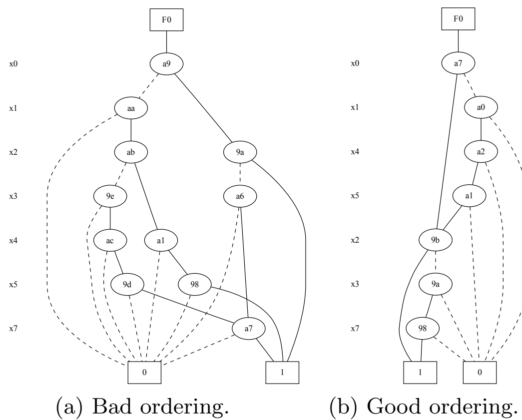


Figure 3: ROBDD for $(pf_1 \wedge pf_2) \vee (pf_2 \wedge pf_3)$. Notation: solid lines are “high” and dashed lines are “low” edges.



(a) Bad ordering. (b) Good ordering.

Figure 4: ROBDD for the query path(1,3).

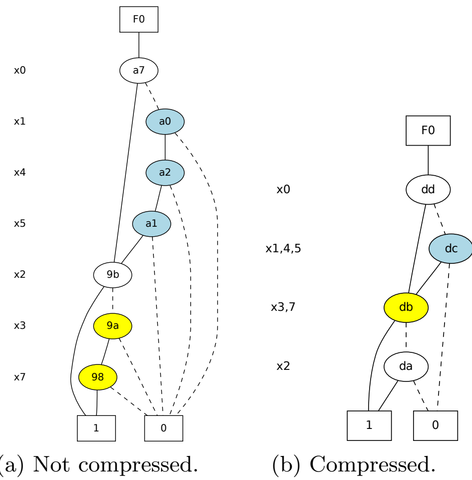
A ROBDD for the `path(1,3)` example of Figure 1 is given in Figure 4a. ProbLog computes that 0.498296 is the exact probability that a path from node 1 to node 3 exists.

Note that in general a variable can have multiple nodes in a ROBDD. For example, in Figure 4a the variables x2, x3, x4, x5 have two nodes each. A ROBDD imposes an order on the Boolean variables and the different variables appear in that order in all the paths of the ROBDD. ROBDDs are reduced which means that two nodes never have the same successors if they are nodes of the same variable and that no node has the same “high” and “low” successor. These reductions do not perform the variable compression we are aiming at.

3. VARIABLE COMPRESSION

3.1 Motivation

ProbLog uses ROBDDs to solve the disjoint sum problem and to compute the success probability of a query, which is a #P-complete problem [3]. While the complexity of the calculation of the probability is linear in terms of the size of the ROBDD, the generation of the ROBDD is NP-hard.



(a) Not compressed. (b) Compressed.

Figure 5: ROBDD of path(1,3) with a good ordering. Notation: coloured nodes represent clusters, representative variables.

It is well-known that the variable ordering used while constructing the ROBDD for a Boolean formula has an impact on the size of the ROBDD. For our path(1,3) example, Figure 4a and 4b use two different variable orderings. The orderings that give rise to smaller ROBDDs are called good orderings: constructing smaller ROBDDs takes less time and space and also the computation of probability is faster. State-of-the-art ROBDD tools use heuristics to decide about the variable ordering, whose search space is exponential.

We reduce the search space for the variable ordering by decreasing the number of variables in the Boolean formula, namely by replacing subsets of variables by new representative variables. We call this **variable compression**. We can only do variable compression if we do not affect the probability.

We discovered sets of variables in the ROBDDs for which we can compute the contribution of such a set of variables to the probability of the ROBDD in advance, independently from the rest of the ROBBD. For example, the set of variables x1, x4 and x5 in Figure 5a. This implies that we can replace those three variables by a new representative variable, whose probability is computed from the probabilities of x1, x4 and x5. Later in this section we define this set as an AND-cluster.

In order to do variable compression before ROBDD generation, we need to detect these patterns in the Boolean formulae, or in the case of ProbLog at the level of the DNF (1). It turns out that in the proofs of path(1,3) either the probabilistic facts corresponding to x1, x4 and x5 appear all three together in a proof, or none of them occurs. The AND-clusters are determined for particular DNFs and as such they are query-dependent. For path(1,7), x1, x4 and x5 no longer form an AND-cluster as we also have a proof `edge(1,4),edge(4,5),edge(5,7)` that does not contain x5. Now only x1 and x4 form an AND-cluster.

As the AND-clusters are query-dependent, they do not appear in the ProbLog source program itself. Although one could be tempted to replace the facts `edge(1,4),edge(4,5)` by a single one, this is not a good idea because path/2 queries could have 4 as a starting node.

3.2 Cluster Definitions

We define two kinds of clusters and proof that their compression does not effect the final probability. We define the clusters at the level of the ROBDDs because the patterns can also be valuable for other application areas that use ROBDDs.

Definition 1 (AND-cluster). *Let F be a Boolean formula with variables $v_1, \ldots, v_l$. The variables $\{x_1, \ldots, x_k\} \subseteq \{v_1, \ldots, v_l\}, k > 1$, form an* ***AND-cluster*** *if there exists a variable ordering such that the ROBDD R of F*

1. *has only one node n_i for variable $x_i, \forall 1 \leq i \leq k$,*
2. *node n_j has as only incoming edge the "high" edge of node $n_{j-1}, \forall 2 \leq j \leq k$,*
3. *and the "low" edges of the nodes $\{n_1, \ldots, n_k\}$ connect to the same node in R.*

Definition 2 (OR-cluster). *Let F be a Boolean formula with variables $v_1, \ldots, v_l$. The variables $\{x_1, \ldots, x_k\} \subseteq \{v_1, \ldots, v_l\}, k > 1$, form an* ***OR-cluster*** *if there exists a variable ordering such that the ROBDD R of F*

1. *has only one node n_i for variable $x_i, \forall 1 \leq i \leq k$,*
2. *node n_j has as only incoming edge the "low" edge of node $n_{j-1}, \forall 2 \leq j \leq k$,*
3. *and the "high" edges of the nodes $\{n_1, \ldots, n_k\}$ connect to the same node in R.*

In a probabilistic framework like ProbLog that uses ROBDDs to calculate probabilities, each ROBDD variable has an assigned probability. To be able to compress the clusters of variables to new representative variables, we need to compute the probabilities of the representative variables.

Theorem 1 (Probability of an AND-cluster). *An AND-cluster $\{x_1, \ldots, x_n\}$ can be compressed to the representative variable V whose probability P_V is equal to:*

$$P_V = P_{AND}(\{x_1, \ldots, x_n\}) = \prod_{i=1}^{n} P(x_i)$$

Proof. First consider the simple case of a ROBDD that consist of exactly one AND-cluster, $\{x_1, \ldots, x_n\}$. The probability of this ROBDD is $P(x_1) \cdot ... \cdot P(x_i) \cdot ... \cdot P(x_n) \cdot 1 + (1 - P(x_1) \cdot ... \cdot P(x_i) \cdot ... \cdot P(x_n)) \cdot 0 = \prod_{i=1}^{n} P(x_i)$. In general, an AND-cluster has an outgoing "high" edge to a part T with P_T and its "low" edges connect to a part F with P_F. The probability can be generalised as $P = P(x_1) \cdot ... \cdot P(x_i) \cdot ... \cdot P(x_n) \cdot P_T + (1 - P(x_1) \cdot ... \cdot P(x_i) \cdot ... \cdot P(x_n)) \cdot P_F = P_T \cdot \prod_{i=1}^{n} P(x_i) + P_F - P_F \cdot \prod_{i=1}^{n} P(x_i) = (P_T - P_F) \cdot \prod_{i=1}^{n} P(x_i) + P_F$. If we replace the AND-cluster with a new representative variable V with P_V and calculate the probability, we get $P = P_V \cdot P_T + (1 - P_V) \cdot P_F = P_V \cdot P_T + P_F - P_V \cdot P_F = (P_T - P_F) \cdot P_V + P_F$. Therefore $P_V = P_{AND}(\{x_1, \ldots, x_n\}) = \prod_{i=1}^{n} P(x_i)$. □

Theorem 2 (Probability of an OR-cluster). *An OR-cluster $\{x_1, \ldots, x_n\}$ can be compressed to the representative variable V whose probability P_V is equal to:*

$$P_V = P_{OR}(\{x_1, \ldots, x_n\}) = P(x_1) + (1 - P(x_1)) \cdot P_{OR}(\{x_2, \ldots, x_n\})$$

Proof. First consider the simple case of a ROBDD that consist of exactly one OR-cluster, $\{x_1, \ldots, x_n\}$. The probability of this ROBDD is $P(x_1) \cdot 1 + (1 - P(x_1)) \cdot (P(x_2) \cdot 1 + (1 - P(x_2)) \cdot \ldots \cdot (P(x_i) \cdot 1 + (1 - P(x_i)) \cdot \ldots \cdot (P(x_n) \cdot 1 + (1 - P(x_n)) \cdot 0)) \ldots) = P(x_1) + (1 - P(x_1)) \cdot P_{OR}(\{x_2, \ldots, x_n\})$. In general an OR-cluster has its "high" edges to a part T with P_T and an outgoing "low" edge to a part F with P_F. The probability can be generalised as $P = P(x_1) \cdot P_T + (1 - P(x_1)) \cdot (P(x_2) \cdot P_T + (1 - P(x_2)) \cdot \ldots \cdot (P(x_i) \cdot P_T + (1 - P(x_i)) \cdot \ldots \cdot (P(x_n) \cdot P_T + (1 - P(x_n)) \cdot P_F)) \ldots) = (P(x_1) + (1 - P(x_1)) \cdot P(x_2) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{i-1})) \cdot P(x_i) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{n-1})) \cdot P(x_n)) \cdot P_T + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_n)) \cdot (P_F / P_T)$. If we replace the OR-cluster with a new representative variable V with P_V and calculate the probability, we get $P = P_V \cdot P_T + (1 - P_V) \cdot P_F$ if we replace $P_V = P(x_1) + (1 - P(x_1)) \cdot P(x_2) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{i-1})) \cdot P(x_i) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{n-1})) \cdot P(x_n)$ then we need to prove that $1 - P_V = (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_n)) \Rightarrow 1 - (P(x_1) + (1 - P(x_1)) \cdot P(x_2) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{i-1})) \cdot P(x_i) + \ldots + (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_{n-1})) \cdot P(x_n)) = (1 - P(x_1)) \cdot \ldots \cdot (1 - P(x_n))$. Finally by using the distribution rule we see that the previous formula is a tautology. Therefore $P_V = P_{OR}(\{x_1, \ldots, x_n\}) = P(x_1) + (1 - P(x_1)) \cdot P_{OR}(\{x_2, \ldots, x_n\})$. □

3.3 Discovering AND-clusters

How can we discover AND-clusters before we construct the ROBDD? To generate the ROBDD we collect all the proofs of a query and represent them as a DNF. Fortunately, the information needed to detect AND-clusters also appears in the proofs: either all the probabilistic facts of an AND-cluster appear in a proof, or none of them. A naive approach to detect AND-clusters in the proofs could be to compare all the collected proofs in order to find longest common subsequences (LCS) which is an NP-hard problem. As our problem is a special case of the LCS we can do better.

Definition 3 (AND-cluster in proofs). *An AND-cluster in a set of proofs $\{pr_1, \ldots, pr_m\}$ is a set of probabilistic facts $\{pf1, \ldots, pf_n\}$ such that $\forall pf_i \in \{pf1, \ldots, pf_n\}$*

it holds that $occur(pf_i) = (\bigcap_{pf_i \in pr_j} pr_j) \cap (\bigcap_{pf_i \notin pr_j} \overline{pr_j}) = \{pf_1, \ldots, pf_n\}$

The first part of $occur(pf_i)$ is the set of probabilistic facts that occur in each proof in which pf_i occurs. The second part is the set of probabilistic facts that do not occur in proofs that do not contain pf. We use $\overline{pr_j}$ to denote the complement of the set pr_j with respect to the set of the probabilistic facts in all proofs. The first set is a possible AND-cluster for pf_i but it might also contain probabilistic facts that occur in proofs that do not contain pf_i. In order to exclude the latter ones, the possible AND-cluster has to be restricted to probabilistic facts that only occur in proofs containing pf_i.

The Book Marking algorithm of Section 3.4 deals with all the proofs one-by-one and computes $occur(pf_i)$ for each probabilistic fact. The first version of the algorithm ensures that after adding a new proof, for all probabilistic facts pf_i seen by the algorithm so far $occur(pf_i)$ is computed. The second version of the algorithm computes the possible AND-clusters and postpones their restriction.

3.4 The Book Marking Algorithm for AND-cluster

The algorithm has two main parts: the encoding of each proof by a bit string and the identification of potential AND-clusters. Our algorithm constructs an order for the probabilistic facts of the proofs: it uses the order in which they appear in the proofs. Each probabilistic fact is then identified by its position in this order. The ith probabilistic fact is denoted by $probf(i)$.

The encoding of a proof is a bitstring: the value of the ith bit encodes whether the probabilistic fact $probf(i)$ is used in the proof. We refer to the bitstring as the **occurrence number**. Consider the proofs of the path(1,3) example in Figure 1 encoded as Boolean variables:

{x0, x2}

{x0, x3, x7}

{x1, x4, x5, x2}

{x1, x4, x5, x3, x7}

The first proof uses the probabilistic facts $x0$ and $x2$: this gives rise to the order $x0 < x2$ and the occurrence number is 11. For the second proof we add x3 and x7 to this order: $x0 < x2 < x3 < x7$ and the occurrence number is 1101 (we are reading the bitstrings from right to left).

For the second step we use a two dimensional matrix of bits to encode the AND-clusters. The rows (and actually also the columns) represent AND-clusters: row j corresponds to $probf(j)$ and represents $occur(probf(j))$.

The size of each dimension is equal to the number of different already seen probabilistic facts: this matrix will grow incrementally as we deal with the proofs one by one. The ith row/column corresponds to $probf(i)$. Returning to our example, for the first proof we construct a matrix with 2 rows and columns which grows to 4 rows and columns after the second proof.

As row j represents $occur(probf(j))$, it should be the case that all seen proofs that contain $probf(j)$ also contain all the other probabilistic facts whose corresponding bit is set to '1' in row j. For the first proof this will obviously fill all the matrix bits with '1'. Table 1 presents the current proof processed as P, the list giving the current order of the probabilistic facts as OL, ON the occurrence number of proof P as a bitstring and more compactly as an integer, and M a list of the compact representation of the rows of the matrix as well as the matrix. The matrix at the top of Table 1 shows the matrix after dealing with the first proof $x0, x2$.

Next we explain how we deal with a new proof given a matrix that encodes the AND-clusters for all the previous proofs. The new proof affects the first intersection of *occur* value of all the probabilistic facts pf_i that occur in the new proof. The *occur* values of the other probabilistic facts are affected in their second intersection. Our algorithm uses the occurrence number of the new proof to modify the matrix in 3 different ways. The first modification concerns the old rows of the probabilistic facts that occur in the occurrence number. We compute the conjunctions of the bits of these rows with the occurrence number deactivating possible probabilistic facts that do not appear in the current proof together with probabilistic fact of the row. For example when the second proof appears we compute the conjunction of the row of x0 with the occurrence number $11 \wedge 1101 = 0011 \wedge 1101 = 0001$ and by doing this we deactivate the bit of the probabilistic facts x2.

The second modification concerns the other old rows of the matrix: we compute the conjunction of such a row with the negation of the occurrence number. This deactivates the bits of the probabilistic facts that occur in the current proof and previously occurred together with the probabilistic fact of the row. For our example the second proof modifies the row of x2 as follows $11 \wedge neg(1101) = 0011 \wedge 0010 = 0010$ and deactivates the bit for the probabilistic fact x0.

The last modification concerns the addition of new rows to the matrix that represent the newly added probabilistic facts. We add new rows by computing the conjunction of the occurrence number with the negation of a row that had the bits corresponding to the previously existing facts set to '1'. This will deactivate the bits of all the old probabilistic facts as they have appeared already at least once without the newest probabilistic facts. In our example the two last rows for the probabilistic facts x3 and x7 will be created as $neg(11) \wedge 1101 = 1100 \wedge 1101 = 1100$. The matrix after the second proof is in the bottom part of Table 1

Note that in every example we automatically expanded the columns of the matrix to the new size and filled the bits with '0'. This behaviour can be easily obtained by considering the occurrence number and each row of the matrix as integer numbers and each operator bitwise. This not only simplifies our three steps but also improves the performance of the

Proof(P) = x0, x2
Order List(OL) = [x0, x2]
Occurrence Number (ON) =
= 11 = 3
Matrix(M) = [3, 3]

1	1	x2
1	1	x0
x2	x0	

P = x0, x3, x7
OL = [x0, x2, x3, x7]
ON = 1101 = 13
M = $[3 \wedge 13, 3 \wedge neg(13), neg(3) \wedge 13, neg(3) \wedge 13]$
M = [1, 2, 12, 12]

1	1	0	0	x7
1	1	0	0	x3
0	0	1	0	x2
0	0	0	1	x0
x7	x3	x2	x0	

Table 1: Book Marking algorithm example.

implementation.

Table 6 in the Appendix presents the Book Marking Algorithm step by step for the complete example and Table 2 contains the Book Marking algorithm in pseudocode.

The second modification of the Book Marking algorithm restricts the possible AND-clusters and can be skipped for efficiency reasons, as it can be reconstructed in a later step. Our prototype implementation skips it during the book marking and does the reconstruction in a later step. This implies that the matrix now contains possible AND-clusters as some deactivations might not have been done.

3.5 Processing the Book Marking Matrix

When all three modifications are executed, the resulting matrix can be processed in a straight forward way: iterate over all matrix rows starting the count from 0, and every row i that has a value greater than $2^{(i+1)}$ is representing an AND-cluster. Because the row of each probabilistic fact that is a member of an AND-cluster fully represents that AND-cluster, we have multiple appearances of the same AND-cluster. As the matrix is symmetric with respect to its diagonal we can check if any bit before the ith is active and that informs us about duplicates.

If the second modification is skipped, we have possible AND-clusters and we need to verify them or in other words we need to regain lost deactivations. To verify that a row i is an AND-cluster, we need to compare for each occurring probabilistic fact j its matrix row for possible deactivation of the $probf(j)$. The algorithm that verifies this in the book marking matrix is given in Table 3.

For the example of Figure 1 the Book Marking Algorithm finds two different AND-clusters, namely $\{x1, x4, x5\}$ and $\{x3, x7\}$.

3.6 Concluding our Example

By executing the query `path(1,3)` for the program of Figure 1 ProbLog collects the proofs:

{x0, x2}

```
GLOBAL OrderList, Matrix,
       ListLength = 0, MatrixSize = 0

BookMark(Proof) {
 OccurrenceNumber = encode(Proof)
 updatematrix(OccurrenceNumber)
}
encode(Proof) {
 OccurrenceNumber = 0
 for each PF in Proof {
  Order = findorder(PF)
  OccurrenceNumber = OccurrenceNumber + 2 ^ Order
 }
 return OccurrenceNumber
}
findorder(PF) {
 Order = 0
 for each OPF in OrderList {
  if PF = OPF then return Order
  Order = Order + 1
 }
 add PF in OrderList
 ListLength = Order
 return Order
}
updatematrix(OccurrenceNumber) {
 for (i = 0; i < MatrixSize; i++) {
  if (2 ^ i AND OccurrenceNumber) > 0 then
   \\ Old row of pf in ON - 1st Modification
   Matrix[i] = Matrix[i] AND OccurrenceNumber
  else
   \\ Old row of pf not in ON - 2nd Modification
   Matrix[i] = Matrix[i] AND neg(OccurrenceNumber)
 }
 expand Matrix to ListLength
 for (i = MatrixSize; i < ListLength; i++) {
  \\ Add a new row - 3rd Modification
  Matrix[i] = neg(2 ^ MatrixSize - 1) AND
                               OccurrenceNumber
 }
 MatrixSize = ListLength
}
```

Table 2: The Book Marking algorithm.

```
examinmatrix {
 for (i = 0; i < MatrixSize; i++) {
  empty AND-cluster
  if Matrix[i] AND (2 ^ (i + 1) - 1) = 0 then {
   \\ Row has not been checked previously
   if Matrix[i] > 2 ^ (i + 1) then {
    \\ Possible AND-cluster verify it
    for (j = i + 1; 2 ^ j < Matrix[i]; j++) {
     if (Matrix[j] AND 2 ^ i AND 2 ^ j) > 0 then
      \\ verified that PFj belongs to cluster
      add PFj to AND-cluster
    }
    if AND-cluster not empty then {
      add PFi to AND-cluster
      make AND-cluster
    }
   }
  }
 }
}
```

Table 3: The Verifying algorithm.

{x0, x3, x7}

{x1, x4, x5, x2}

{x1, x4, x5, x3, x7}

ProbLog generates the ROBDD of Figure 6 for them, which has a size in between the sizes of the ROBDDs in Figure 4. The Book Marking Algorithm finds two different AND-clusters, namely $\{x1, x4, x5\}$ and $\{x3, x7\}$. We use them to compress the variables of the AND-clusters to a representative variable x1,4,5 with $P(x1, 4, 5) = 0.252$ and x3,7 with $P(x3, 7) = 0.32$ and got the compressed proofs:

{x0, x2}

{x0, x3,7}

{x1,4,5, x2}

{x1,4,5, x3,7}

For the compressed proof ProbLog generates the compressed ROBDD of Figure 5b.

3.7 Using OR-clusters

At this point we can make the interesting exercise to see what OR-clusters could add, although we currently do not have a detection algorithm for OR-clusters. The ROBDD of Figure 7a has two OR-clusters which we could further compress to get the ROBDD of Figure 7b which then would contain an AND-cluster which we can compress and finally get the ROBDD of Figure 7c. Not all ROBDDs can be compressed to a single variable. The combination of AND-clusters and OR-clusters will be part of our future work.

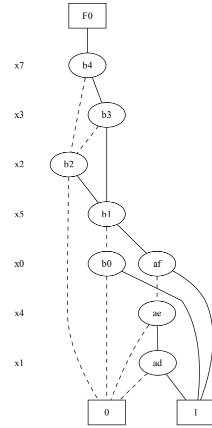
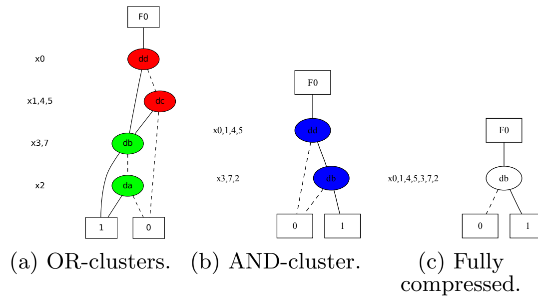


Figure 6: ProbLog generated ROBDD for the query path(1,3).



(a) OR-clusters. (b) AND-cluster. (c) Fully compressed.

Figure 7: Recursively compressing AND-cluster, OR-clusters.

Below we present all the calculations for compressing the ROBDD step by step and then resulting to the same probability as calculated from ProbLog exact inference method.

$$
\begin{aligned}
P(x1,4,5) &= P_{AND}(\{x1,x4,x5\}) = \\
&= 0.4 \cdot 0.9 \cdot 0.7 = 0.252 \\
P(x3,7) &= P_{AND}(\{x3,x7\}) = 0.8 \cdot 0.4 = 0.32 \\
P(x0,1,4,5) &= P_{OR}(\{x0,x1,4,5\}) = \\
&= 0.5 + (1-0.5) \cdot 0.252 = 0.626 \\
P(x3,7,2) &= P_{OR}(\{x3,7,x2\}) = \\
&= 0.32 + (1-0.32) \cdot 0.7 = 0.796 \\
P(x0,1,4,5,3,7,2) &= P_{AND}(\{x0,1,4,5,x3,7,2\}) = \\
&= 0.626 \cdot 0.796 = 0.498296
\end{aligned}
$$

4. EXPERIMENTS FOR AND-CLUSTERS

We implemented the variable compression method using only AND-clusters within ProbLog. To judge the practicality and the impact we used the same data set and the same queries as [5], which is a real world application. The experiments should give answers to the following questions:

1. What is the compression rate in a real life data set?
2. How does compression improve the performance of generating a ROBDD?
3. If and where would the variable compression be beneficial?

We executed the 360 queries of [5] in random graphs but 100 queries do not use any probabilistic facts to be proved. We divided the other 260 queries in 3 groups: the first group contains 92 queries that generate tiny ROBDDs with less than 20 variables; the second group contains 152 queries that generate small ROBDDs with 20 or more variables but less than 100; and finally the third group contains the queries that generated relatively big ROBDDs with more than 100 variables.

We computed averages for the three groups and we also present the average results of all the queries with more than 20 variables. The results are in Table 4. We computed the compression rate for the variables [3], the time gain realised for the ROBDD generation [4] and the time loss [5].

Query Group	ROBDD Comp. Rate	ROBDD Gen. Time Gain	Book Marking Time loss
Tiny	42%	-29%	7%
Small	28%	40%	26%
Big	27%	47%	69%
All	28%	41%	32%

Table 4: Averaged results.

[3](number of variables before compression - number of variables after compression) / number of variables before compression

[4](ROBDD time without compression - ROBDD time with compression) / ROBDD time without compression

[5](SLD time with book marking algorithm - SLD time without) / SLD time without

While the results might be specific for the application we are using, they give an indication of the actual presence of AND-clusters in real world applications. We expect that artificial problems because they contain less redundant data will have less appearances of AND-clusters. In this real application we encounter a surprising high compression rate that ranges from 7% to 61% with an average of 28%.

The state-of-art tool we are using for the ROBDD generation uses the following memory-time trade-off. It starts by consuming memory without reordering the variables, once memory usage passes a threshold it starts reordering the variables and as a consequence consuming time. This affects our results making the 'Tiny query' group actually non informative, as even if it has a huge compression rate of 42% there is no actual generation gain. For that reason we chose not to use this group when we calculated the average over the whole set of queries. This observation suggests that we should trigger the variable compression after the count of appeared probabilistic facts exceeds a threshold.

In the 'Small query' group 44% of the benchmarks have a small number of variables that do not need variable reordering neither before nor after compression: their time gain is near 0. The 'Small query' group also has many benchmarks that need reordering before and no reordering after compression, so they have a huge time gain up to 87%. On average we end up with a gain of 40%.

For the 'Big query' group the average gain is larger namely 47%, but the variation is less as all the benchmarks need reordering before and after compression. Here the gain comes from having less variables that have to be dealt with during the reordering by the state-of-the-art tool.

In addition to the averaged numbers we also present some interesting results separately in Table 5. We give the size of the generated ROBDD and the time in milliseconds for the generation of the ROBDD without compression and the same information after compression. The first line is about the query with the best compression rate, namely 60% and a time gain of 21%. Next the results for the query with the best gain during the ROBDD generation (87%) and a compression rate of 31%. Finally we present an example of a query that even if it has a compression rate of 27% in the generation of the ROBDD underperforms. The reason for this is that the generation of the ROBDD did not trigger any reordering on this example even if it has a fair number of variables.

Currently the Book Marking algorithm is implemented in Yap Prolog [9] while for the ROBDD generation we are using an efficient state-of-art tool implemented in C, this makes the comparison of time loss with time gain irrelevant. It is important to mention that even under this unfair circumstances there are examples for which the time loss in Prolog is significantly lower than the time gained in C. Moreover, the time needed for the Book Marking algorithm is significantly lower than the time needed by the SLD resolution to compute the proofs in all of our benchmarks. Note that the time loss is always below 100%. From this we conclude that parallelism is a valid option that would almost diminish the cost of the optimisation.

Query	Original		Compressed	
	#var	Time	#var	Time
Best Comp.	23	81	9	63
Best Time	103	865	71	106
Worst Case	62	15	45	20

Table 5: Individual experiments. The reported times are in milliseconds.

These experiments[6] gives us promising results, answering our initial questions by showing that there is in real life ProbLog applications space for variable compression by our method; and that the compression improves significantly the performance of the ROBDD generation. We also discovered that the use of Book Marking Algorithm for ProbLog will significantly benefit by parallelism as is argued in Section 6.

5. COMPLEXITY ANALYSIS

The Book Marking algorithm presented at Table 2 has a worst case complexity of $O(M * N^2)$ with usually $M >> N$ where M is the number of proofs seen and N is the number of different probabilistic facts. This worst case complexity is actually because of the naive way that we use to handle the proof encoding. The function encode will iterate $K \leq N$ times the number of probabilistic facts in the current proof and call the function findorder, which will iterate up to $L \leq N$ times the number of seen probabilistic facts. Giving a total worst case complexity for the encode step of $O(N^2)$. The function updatematrix only iterates L times having a worst case complexity of $O(N)$. The Book Marking algorithm is executed M times giving a total complexity of $O(M * (N^2 + N)) = O(M * N^2)$. For this complexity we assume that the arbitrary precision integers have a constant computational cost.

Regarding the space complexity the Book Marking algorithm keeps the order list and the matrix. The matrix consumes N^2 bits but we encode it with arbitrary precision integers and the order list has a linear memory cost of N. Giving a space complexity of $O(N^2)$. While the memory consumption is not that much we found that it is important for Yap to dispose the matrix immediately when we don't require it anymore.

6. CONCURRENCY

In this section we indicate how the Book Marking algorithm could be executed in parallel with the SLD resolution. The necessary input of the Book Marking algorithm is the current proof, the probabilistic fact order list and the matrix, and as output we have a new probabilistic fact order list and a new matrix. The SLD resolution is generating the proofs which normally would only be stored in the data structure that collects the proofs namely a trie. By creating a first in first out (FIFO) queue that holds the proofs we can execute the Book Marking Algorithm completely independent from the SLD resolution. Then when the SLD resolution is finished we only need to wait for the queue to be completely empty before processing the matrix and marking the AND-clusters. The diagram of Figure 8 displays this parallelism.

[6]For our experiments we used 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.

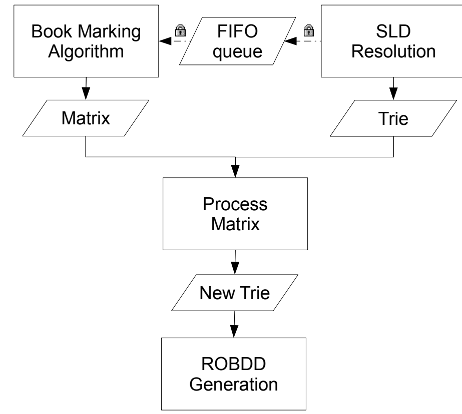


Figure 8: Parallel process diagram.

A simple implementation for the FIFO queue would be to use a pipe and for multi-threading one could just fork the two processes and wait for the Book Marking process to end.

7. CONCLUSIONS, FUTURE AND RELATED WORK

In this paper we have shown that it is possible to take advantage of the AND-clusters contained in a ROBDD to improve the performance of generating the ROBDD. Therefore we detect the AND-clusters at the level of the proofs such that they can be transformed accordingly. Currently the cost of detecting the AND-clusters is still significant. But as parallelism is available even in our personal computers nowadays, the performance cost can be easily paid by doing the detection in parallel. We have presented a polynomial algorithm for detecting the AND-clusters and we have obtained promising results for an application using a real database.

As this is preliminary work the implementation of our algorithm can be improved a lot. While the Book Marking algorithm is still polynomial there are some naive linear costs in the implementation that can become constant. For a future implementation of the Book Marking algorithm C would be a better choice than Prolog both for time efficiency as for space. This would save many hidden constant costs of Prolog and would also save Prolog garbage collector executions. Finally regarding the implementation we need to take advantage of parallelism.

In addition to the technical improvements, a challenging task is to further investigate how we can take advantage of possible OR-clusters and compress even more the ROBDDs. Finally the goal would be to generalise the method and be able to compress repeated structures in the ROBDD. The size of the ROBDDs is one of the limits that is currently reached when executing ProbLog programs and we aim at pushing the current limits through variable compression.

To the best of our knowledge few related work exists [2, 6]. This is probably due to the particular probabilistic setting in which we are studying the ROBDDs. Paper [2] determines a static ordering of the variables before the generation of the

ROBDD and uses a technique to detect a good ordering in a Boolean function in CNF form. Paper [6] discusses about finding the most reliable subgraph. They calculate the probability of subgraphs connecting two nodes and search for the subgraph with the maximum probability. To compute this problem they exploit a special case: the series-parallel subgraphs for which they can compute the probability polynomially. This series-parallel subgraphs have similarities with our AND/OR-clusters.

APPENDIX

A. APPENDIX

Here we present the complete execution of the Book Marking algorithm using the proofs:

x0, x2

x0, x3, x7

x1, x4, x5, x2

x1, x4, x5, x3, x7

These proofs generate the ROBDD in Figure 6 and after the variable compression generate the ROBDD in Figure 5b. In Table 6, you find P the last proof processed, the list OL giving the current order of the variables, ON the occurrence number of proof P as a bitstring and more compactly as an integer, and M a list of the compact representation of the rows of the matrix and finally the matrix step by step.

B. ACKNOWLEDGEMENTS

We want to thank Bart Demoen for the valuable discussions and comments. This research is supported by: GOA/08/008 "Probabilistic Logic Learning".

C. REFERENCES

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

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster SAT and smaller BDDs via common function structure. In *ICCAD*, pages 443–448, 2001.

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

[4] L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. *NIPS*2008 Workshop Probabilistic Programming*, December 2008.

[5] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In M. M. Veloso, editor, *IJCAI*, pages 2462–2467, 2007.

[6] P. Hintsanen. The most reliable subgraph problem. In *PKDD 2007: Proceedings of the 11th European conference on Principles and Practice of Knowledge Discovery in Databases*, pages 471–478, Berlin, Heidelberg, 2007. Springer-Verlag.

Proof(P) = x0, x2
Order List(OL) = [x0, x2]
Occurrence Number (ON) = = 11 = 3
Matrix(M) = [3, 3]

1	1	x2
1	1	x0
x2	x0	

P = x0, x3, x7
OL = [x0, x2, x3, x7]
ON = 1101 = 13
M = $[3 \wedge 13,\ 3 \wedge neg(13),\ neg(3) \wedge 13,\ neg(3) \wedge 13]$
M = [1, 2, 12, 12]

1	1	0	0	x7
1	1	0	0	x3
0	0	1	0	x2
0	0	0	1	x0
x7	x3	x2	x0	

P = x1, x4, x5, x2
OL = [x0, x2, x3, x7, x1, x4, x5]
ON = 1110010 = 114
M = $[1 \wedge neg(114),\ 2 \wedge 114,\ 12 \wedge neg(114),$
$12 \wedge neg(114),\ neg(15) \wedge 114,$
$neg(15) \wedge 114,\ neg(15) \wedge 114]$
M = [1, 2, 12, 12, 112, 112, 112]

1	1	1	0	0	0	0	x5
1	1	1	0	0	0	0	x4
1	1	1	0	0	0	0	x1
0	0	0	1	1	0	0	x7
0	0	0	1	1	0	0	x3
0	0	0	0	0	1	0	x2
0	0	0	0	0	0	1	x0
x5	x4	x1	x7	x3	x2	x0	

P = x1, x4, x5, x3, x7
OL = [x0, x2, x3, x7, x1, x4, x5]
ON = 1111100 = 124
M = $[1 \wedge neg(124),\ 2 \wedge neg(124),\ 12 \wedge 124,$
$12 \wedge 124,\ 112 \wedge 124,\ 112 \wedge 124,\ 112 \wedge 124$
M = [1, 2, 12, 12, 112, 112, 112]

1	1	1	0	0	0	0	x5
1	1	1	0	0	0	0	x4
1	1	1	0	0	0	0	x1
0	0	0	1	1	0	0	x7
0	0	0	1	1	0	0	x3
0	0	0	0	0	1	0	x2
0	0	0	0	0	0	1	x0
x5	x4	x1	x7	x3	x2	x0	

Table 6: Book Marking algorithm example.

[7] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. On the efficient execution of ProbLog programs. In M. G. de la Banda and E. Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2008.

[8] A. Rauzy, E. Châtelet, Y. Dutuit, and C. Bérenguer. A practical comparison of methods to assess sum-of-products. *Reliability Engineering & System Safety*, 79(1):33 – 42, 2003.

[9] R. Rocha, F. M. A. Silva, and V. Santos Costa. YapOr: an Or-Parallel prolog system based on environment copying. In P. Barahona and J. J. Alferes, editors, *EPIA*, volume 1695 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 1999.

[10] P. Sevon, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. Link discovery in graphs derived from biological databases. In U. Leser, F. Naumann, and B. A. Eckman, editors, *DILS*, volume 4075 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2006.