On the Implementation of the CLP($\mathcal{BN}$) Language

Vítor Santos Costa[1]

DCC/FCUP and CRACS-INESC Porto LA,
Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
vsc@dcc.fc.up.pt

Abstract. The last few years have seen great interest in developing models that can describe real-life large-scale structured systems. A popular approach is to address these problems by using logic to describe the patterns or structure of the problems, and by using a calculus of probabilities to address the uncertainty so often found in real life situations. The CLP($\mathcal{BN}$) language is an extension of Prolog that allows the representation, inference, and learning of bayesian networks. The language was inspired on Koller's Probabilistic Relational Models, and is close to other probabilistic relational languages based in Prolog, such as Sato's PRISM.
We present the implementation of CLP($\mathcal{BN}$), showing how bayesian networks are represented in CLP($\mathcal{BN}$) and presenting the implementation of three different inference algorithms: Gibbs Sampling, Variable Elimination, and Junction Trees. We show that these algorithms can be implemented effectively by using a matrix library and a graph manipulation library, and study how the system performs on real-life applications.

1 Introduction

The last few years have seen great interest in developing models that can describe real-life large-scale structured systems. A popular approach is to address these problems by using logic to describe the patterns or structure of the problems, and by using a calculus of probabilities to address the uncertainty so often found in real life situations. Examples include ICL [1], PRISM [2], Probabilistic Relational Models [3], Stochastic Logic Programs [4], Markov Logic Networks [5], Problog [6], and CLP($\mathcal{BN}$) [7, 8], among many others [9, 10].

These languages differ in a number of ways, including the logic used, the graphical (probabilistic) model followed, and the way the two are combined together. Arguably, a key difference is whether the language consists of a set of true statements about objects whose properties are only partially known, as in PRISM or CLP($\mathcal{BN}$), or if the language allows uncertainty about the truth the statements, as in MLNs. Languages also differ on the formalism being used, Prolog being a popular option, and on the underlying graphical model: whether it is discrete or continuous, and whether it is directed or undirected.

Ultimately, the usefulness of these languages strongly depends on their ability to answer complex queries. Inference in Bayesian Networks is known to be NP-hard, and can be quite expensive even in traditional (propositional) networks. Models that combine logic and probabilities have the ability to easily construct very large graphical networks, and can arguably make the problem even harder.

Next, we present how these problems are addressed in the context of the CLP($\mathcal{BN}$) language. In this language, uncertainty about the value of a variable is represented as a constraint on the variable [8]. Inference on the logic program naturally constructs a network of random variables and provides a natural method for constructing bayesian networks.

CLP($\mathcal{BN}$) was implemented in Prolog, Our motivation was threefold. First, we wanted to address the licensing and practical issues associated with using external tool-kits. At the time we developed the system most stable implementations of bayesian networks either were commercial, relied on commercial systems, or had significant scalability and/or licensing issues. Second, we wanted to have the flexibility that one can only achieve with its own implementation. This has proven most valuable in supporting learning, and we believe it will prove useful as we experiment novel algorithms in the future. Third, we wanted to experiment with using logic programming for this purpose: Gibbs sampling, say, is not a typical Prolog application.

The literature reports a very large number of inference models for graphical models [9]. The CLP($\mathcal{BN}$) implementation supports three widely used inference methods. *Gibbs sampling* [11] is a popular method for approximate inference. It is often used for complex networks, say, in the BUGS system [12]. *Variable elimination* [13] is a relatively simple inference model that answers queries by reducing the number of random variables one by one. Last, *junction tree* construction is a popular method where queries are answered by using believe propagation over a "junction tree" that represents the network.

2 CLP($\mathcal{BN}$)

The CLP($\mathcal{BN}$) language is an extension to Prolog where variables with undefined values are represented as a constraint. As an example, consider the definition for a coin-flip:

```
flip(X) :-
       { X = flip with p([h,t],[0.5,0.5]) }.
```

The constraint includes two components: the left-hand side of the `with` is a key that uniquely references the random variable; the right-hand side of the key is a term describing the probability distribution of the values. Using a key allows one to identify different logical variables with the same random variable. The term includes X's domain and probability values of every element in the domain.

Querying this procedure returns a constrained object. Prolog then outputs statements about the probabilities entailed by the constraints:

```
?- flip(X).
p(X=h)=0.5,p(X=t)=0.5
```

CLP($\mathcal{BN}$) uses the key `flip` to represent identity across different calls to `flip(X)`. That is, *every constraint with the same key refers to the random variable.* Thus, the call:

```
?- flip(X), flip(Y), flip(Z).
```

entails $X = Y = Z$, as shown in the actual answer given by CLP($\mathcal{BN}$):

```
X = Y = Z,
p(X=h)=0.5,p(X=t)=0.5
```

A sequence of independent coin-throws can be represented as a list of random variables:

```
flips(0, []).
flips(I, [F|Fs]) :-
       I > 0,
       flip(I, F),
       I1 is I-1,
       flips(I1, Fs).

flip(I,X) :-
       { X = flip(I) with p([h,t],[0.5,0.5]) }.
```

As expected, querying this procedure returns a list of random variables:

```
?- flips(5, L).
L = [_A,_B,_C,_D,_E],
p((_E=h,_D=h,_C=h,_B=h,_A=h))=0.03125,
... ?
```

CLP($\mathcal{BN}$) returns the joint distribution, which in this case is uniform. Notice that because of space considerations we do not show the whole output.

Last, a bayesian network can be used to represent conditional dependencies between the different random variables.

```
flips(0, _, []).
flips(I, F, [F|Fs]) :-
       I > 0,
       flip(I, F, F1),
       I1 is I-1,
       flips(I1, F1, Fs).

flip(1, X, _) :- !,
       { X = flip(1) with p([h,t],[0.5,0.5]) }.
flip(I, X, X0) :-
       { X = flip(I) with p([h,t],[0.6,0.4,0.4,0.6],[X0]) }.
```

The first clause for `flip/3` implements the base case, or *prior*, which is still uniform. The second clause implements a *conditional probability distribution*. Each value in the list of floating point numbers is a *conditional probability*. In this case, it states that the probability of having the same value as in the previous flip is 60%, and the probability of having a different value is 40%. The joint probability distribution is quite different in this case:

```
?- flips(5, _, L).
X = [_A,_B,_C,_D,_E],
p((_E=h,_D=h,_C=h,_B=h,_A=h))=0.0648,
... ?
```

Quite often, in probabilistic networks we have information on some random variables, and our goal is to find out how probabilities for other variables (the marginals) change. If the value of a random variable is known in advance, we say that we have *evidence* on the variable. CLP($\mathcal{BN}$) uses unification as a natural mechanism for introducing evidence:

```
?- flips(5, _, L), nth(2,L,h).
L = [_A,_B,_C,_D,_E],
_B=h,
p((_E=h,_D=h,_C=h,_A=h))=0.1296,... ?
```

3 The Implementation

The current implementation of CLP($\mathcal{BN}$) works in two steps. In a first step, execution creates a network of constraints. In a second step, this network is sent to a constraint solver that computes the joint probability of the possible values of the query variables.

Figure 1 shows the structure of the CLP($\mathcal{BN}$) implementation. As shown in the examples above, execution starts in Prolog style, by launching a query. During execution, the predicate `{}/2` will be called a number of times, creating a set of random variables. Random variables are represented as attributed variables. The main attributes are

- the reference to the random variable, or *key*;
- the unique identifier for the distribution, or *id*;
- the set of parent variables, or *parents*.

The identifier *id* refers to a table of distributions, that stores the domain and the probability tables for each different distribution.

Query execution terminates by calling the `clpbn:project_attributes/2` predicate. The predicate receives two arguments: one is a list of query variables, Qs, and the other a list of all attributed variables, As. The task of `clpbn:project_attributes/2` is to compute the joint distribution of the random variables in Qs, given the network of constraints that connects together the variables in As. The predicate first performs simplification and calls one

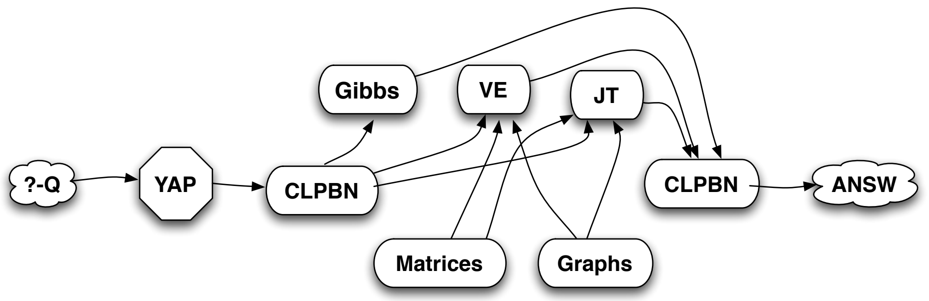

Fig. 1. Structure of the CLP($\mathcal{BN}$) system.

of the three solvers. The solvers ultimately output the distribution over the query variables, or marginals. The distribution is exported as an extra constraint, `posterior`. The top-level calls `clpbn_display:attribute_goal/2` to transform the `posterior` constraint into a set of goals, shown in the examples above.

Figure 1 shows the three current solvers. Note that originally, CLP($\mathcal{BN}$) used as solver an interface that sent the network to the Bayesian Network Toolbox (BNT) [14], a toolkit written in Matlab. Unfortunately, this solution offered a number of limitations, leading us to implement specific solvers for CLP($\mathcal{BN}$).

Experience showed that to implement the solvers require extensive matrix operations as they manipulate tables for discrete distributions. Algorithms such as junction trees further require extensive graph manipulation. Our implementation thus relies on two support libraries.

Matrices The *matrix* library is `C`-code that implements multidimensional matrices of integers or floating point numbers. The library provides straightforward matrix operations and the operations required to support the solvers. The *dist* library is responsible to transform a probability distribution from the initial list format into a `matrix`.

Matrices reside on the Prolog global stack as *blobs*. The `blob` mechanism was originally implemented to support very large numbers, but has since been shown useful for a variety of purposes. Each blob can be seen as variant of compound term with a special functor, a tag identifying the type of blob, the size of the blob, and the actual data itself. In the case of matrices data is matrix type, number of dimensions, a linear array with the size of each dimension, and the actual matrix.

Graphs Bayesian Networks are graphs, therefore it is unsurprising that the algorithms described next require a number of graph operations.

We believe Prolog is an excellent language for graph manipulation, as it allows sophisticated pattern matching, and high-level description of graph operations. We therefore implemented a number of Prolog libraries for graph manipulation.

The libraries implement graphs over association lists, that themselves are implemented as red-black trees. The `dgraphs` library implements the basic operation over directed graphs. This library is then reused by the `undgraphs` library, as this one implements undirected graphs by transporting them into directed graphs with edges in both directions. Last, weighted versions are implemented on top of these two libraries. The libraries take advantage of the `reexport` module directive to avoid code duplication.

4 Variable Elimination

As the name says, the variable elimination procedure [13] executes by eliminating random variables until only the query variables remain. The (simplified) main procedure is shown next:

```
solve_ve(QVs, AllVs, Ps) :-
        random_vars_to_graph(AllVs, Graph),
        process(Graph, QVs, Dist),
        normalise_CPT(Dist,MPs),
        export(MPs, Ps).
```

The first step of the algorithm is to generate a graph from the random variables. The graph is accessible through two data structures: a list of *factors* or tables, and a list of *variables*, or nodes. Factors store a table and corresponding variables. Variables maintain a list of every factor they participate in.

The `random_vars_to_graph/2` generates the initial factors from each variable's probability tables, and associates each variable with the factors it participates in. Variables with evidence must also be eliminated before variable elimination starts. This requires discarding every entry that is not compatible with the evidence, and is implemented as a single matrix operation.

The `process/3` predicate is the core of the algorithm. It operates as follows:

```
process(Vs, QVs, Out) :-
        find_best(Vs, V, WorkTables, LVI, QVs), !,
        multiply_tables(WorkTables, Table, Parents),
        project_from_CPT(V,Table,NewTable),
        include(Vs, Parents, NewTable, NewVs),
        process(NewVs, InputVs, Out).
process(Vs, _, Out) :-
        fetch_tables(LV0, WorkTables),
        multiply_tables(WorkTables, Out).
```

The procedure first finds a variable that can be eliminated. If one such variable is found, it "multiplies" all tables where the variable participates, projects the variable out of the joint table, and makes the parents replace their previous table by a new table. Unification is used to simplify the latter task.

Notice that multiplication operation is not actual matrix multiplication. The operation is implemented through the matrix library. Notice also that the output matrix is not an actual probability table, as the columns are not guaranteed to add to one.

If no such variable can be found, the remaining variables must be the query variables. The algorithm multiplies the remaining factors, and normalises them in order to obtain a true probability table.

The performance of the algorithm strongly depends on finding the best sequence of variables: if we have a variable with N binary neighbors, table size will be 2^{N+1}. CLP($\mathcal{BN}$) follows the standard approach and uses a greedy algorithm where one chooses the variable with the smallest number neighbors at each point.

5 Junction Trees

A popular algorithm for inference in bayesian networks is junction tree construction. The intuition of the algorithm is to transform a general bayesian network, a directed acyclic graph, into a tree. Bayesian inference in trees is much simpler than in the general case. The algorithm therefore proceeds in two steps:

1. Construct a *junction tree*, a representation of the graphical network. Each node in the tree is a factor obtained from nodes that are strongly connected.
2. Apply belief propagation to the junction-tree. This executes as two steps: first, an upward step propagates information from the leaf-nodes (e.g., evidence). Second, a downward step propagates the joint values to the individual variables.

Junction trees are popular because constructing the junction tree, although expensive, is independent of the query variables and of any evidence. Thus, as soon as we construct the junction tree for a network we can use it for different query variables and for different evidence.

The CLP($\mathcal{BN}$) implementation is as follows:

```
build_jt(BayesNets, CPTs, Tree) :-
        init_undgraph(BayesNet, Moral0),
        moralised(BayesNet, Moral0, Markov),
        undgraph_vertices(Markov, Vertices),
        triangulate(Vertices, Markov, Markov, _, Cliques0),
        cliques(Cliques0, EndCliques),
        wundgraph_max_tree(EndCliques, J0Tree, _),
        root(J0Tree, JTree),
        populate(CPTs, JTree, Tree).
```

The junction-tree algorithm [15] essentially tries to construct a tree from a directed acyclic graph. It does so by adding edges until the graph is triangulated, constructing a graph from the cliques in the triangulated graph, and obtaining a tree:

1. Create a *moral graph*, an undirected graph where the parents of every node are always connected together, by introducing edges between parents.

2. Perform *triangulation*, that is it ensure that the graph is chordal by introducing edges so that every cycle that has more than 3 nodes has at a least a chord.
3. Detect all cliques in the undirected graph, and construct a new weighted undirected graph, where each node is a clique of variables. Nodes have edges if they share a variable. The weight of each edge is given by the number of variables shared between the nodes.
4. Select an arbitrary node as a root of the clique tree.
5. Construct a *maximal covering tree* of the clique tree. Call that weighted directed tree the junction tree.
6. Fill each node of the tree with the factor obtained from multiplying all tables where the variables belong to the clique.

Two steps of the algorithm are not deterministic: there are different ways to perform triangulation, and any node can be chosen as a tree root. CLP($\mathcal{BN}$) uses a version of the algorithm where one does not explicitly perform triangulation. Instead, one first searches for *simplicial nodes*, that is, nodes such that the node and all its neighbors form a clique, and removes that node (this is based on the observation that a triangulated graph always has a simplicial node). If no such node exists, it searches for the node that could be made a simplicial node with the smallest clique.

The implementation strongly relies on the graph library. Although the algorithm is rather complex, most operations are little more than transforming directed into undirected or weighted undirected graphs, and then adding or removing edges.

The second step of the algorithm perform belief propagation in four steps:

1. simplify cliques with evidence;
2. do message passing upwards;
3. do message passing downwards;
4. select a clique with the query variables, and marginalise.

Belief propagation can be seen as a sequence of matrix multiplications (in the style of variable elimination), and its implementation thus relies heavily on the matrix library. As an example, a simplified version of upward message propagation is shown next:

```
upward([], _, Tab,  [], Tab).
upward([t(CVars,CTab,CKids)|Sibs],Vars,Tab,
       [t(CVars,UpdCTab,UpdCKids)|UpdSibs],NTab) :-
        upward(CKids, CVars, CTab, NTabKids, UpdCTab),
        ord_intersection(CVars, Vars, Int),
        sum_out_from_CPT(Int, NewTab1, CVars, Tab1),
        multiply_CPTs(Tab, Tab1, ITab),
        upward(Kids, Vars, ITab, NKids, NTab).
```

Each node in the tree is represented as a term `t/3`, where the arguments are the random variables in the clique, the factor table, and a list of children. The algorithm constructs a new tree recursively by calling upward message propagation for each child, summing out the variables not in the intersection, and then multiplying the child's table (the message) by the current table. The process repeats until messages from all children have been received. Downward propagation executes in the opposite fashion.

6 Gibbs Sampling

The last inference method we use is the well-know Gibbs sampler. This is an example of a Markov Chain Monte Carlo method. The method works as follows:

1. Start from a random initial setting of the random variables.
2. For N steps:
   (a) For every variables v:
      i. Compute the current distribution of v based on the current value of its neighbors;
      ii. Sample a new value v_i for v according to this distribution;
3. For the query variable q, compute the probability of each q_i as the number of times $q = q_i$ over N.

It is often the case that we discard the first M steps, or burn-in steps. Also, often one runs several iterations of this algorithm, or *chains*, in parallel. The advantage is that by comparing the current estimates from the different chains, one can estimate how close to convergence the algorithm is. The number of burn-in steps, M, chains, C, and maximum number of steps, N, are the main parameters to this algorithm.

Implementing this algorithm in Prolog is reasonably straightforward. Unfortunately, we did found that computing the current distribution of v is quite expensive because we need to generate the variable's Markov blanket. It requires:

1. Finding all neighbors of the variable (the so-called *Markov Blanket*);
2. Multiplying all probability tables in the Markov Blanket;
3. Eliminating according to evidence.

In order to speed-up this process, CLP($\mathcal{BN}$) performs a pre-processing step where it constructs a table with the distributions for all possible alternatives of the Markov Blanket for each variable. CLP($\mathcal{BN}$) benefits from the multi-way indexing in YAP to guarantee efficient access even for reasonably large Markov blankets. [16]. The key code is:

```
do_var(I,Sample,Sample0,Graph) :-
        arg(I,Graph,var(_,_,_,_,_,CPTs,Parents,_,_)),
        fetch_parents(Parents,I,Sample,Sample0,Bindings),
        ( compiled(I) ->
         recorded(mblanket,m(I,Bindings,Vals),_)
        ;
         multiply_all_in_context(Parents,Bindings,CPTs,Graph,Vals)
```

```
    ),
    X is random,
    pick_new_value(Vals,X,0,Val),
    arg(I,Sample,Val).
```

If the variable I has been preprocessed, it is fetches the current value of the parents and consults the data-base. We use the YAP internal database for compactness, as YAP can index sub-terms within this database. Otherwise, we need to multiply the tables. Next, we take a value from the distribution. We then set this value by simply unifying the argument I of a compound term. In this case, we take advantage of YAP's ability to manipulate large arity compound terms.

7 Evaluation

Our experience in implementing CLP($\mathcal{BN}$) shows that Prolog is a much better tool for these tasks than what we expected. It also seems to indicate it could be an even better tool.

In our experience, the main advantages of using Prolog are flexibility and compactness. Regarding compactness, the final code we needed to write for each algorithm was remarkably small: around 300 lines for variable elimination, and less than 600 for junction trees and the gibbs sampler. The glue libraries that connect to the matrix and the graph libraries are even smaller: less 300 lines for the matrix interface, and a bit over 300 for the `dist` library (that encapsulates distributions).

Prolog also makes it possible to write and change code quite easily. The code is mostly side-effects free, which makes it possible to easily debug by redoing computations. In general, the ability to quickly specify a procedure in a top-down fashion, and then debug and refine it incrementally is a major advantage of Prolog.

On the other hand, we found a number of drawbacks. The first drawback we noticed is the limited number of libraries and toolboxes currently available. For example, graph manipulation is natural to implement in Prolog. Unfortunately, when we started this project there was a single free library for graph manipulation, written 30 years ago. This was helpful, but not sufficient and we had to write our own code. Fortunately, we believe that this lesson has been well learned by the Prolog community. Initiatives such as Prolog Commons are trying to address this problem.

The matrix library is a second case in point. Originally, we represented multi-dimensional matrices as lists. Unfortunately, lists lose structure and force sequential access. Our second implementation used compound terms. This implementation allowed direct access but ultimately we still found it hard to maintain and understand code that performed matrix transposition, summing out, and similar operations. In the end, we felt it was just more natural to write this code in `C`, and more efficient. We believe that the lesson is that it is important to be able to connect easily to languages written in other languages: we simply cannot expect to do everything in Prolog.

One major problem we found in designing the system is that Prolog is a hard language for software engineering. As an example, changing the implementation of probability tables from lists to compound terms, and then to C-code exposed a number of structural problems in the system. There are two problems:

– Prolog makes it easy to state a first solution and refine it, but it does not make it easy to add meta-information and structure the program;
– Without this information, it becomes very hard to obtain encapsulation, which makes the programs unnecessary brittle.

The single tool Prolog has to provide structure is the module system. There has been some interest in trying to address these problems, by introducing type systems [17, 18]. We believe that even simpler solutions, such as a macro expansion mechanism that could encapsulate access to a data structure may be useful.

7.1 Experiments

In order to obtain a feeling about the performance of the system, we apply CLP($\mathcal{BN}$) to a typical learning task. The goal is to learn the parameters of a structured bayesian network representing a simulated school with students, professors, grades, and courses. We assume that we know the structure of the network, given as CLP($\mathcal{BN}$) clauses. For example, `registration_grade` depends on the value of the course difficulty and the student's intelligence:

```
registration_grade(RegKey, Grade) :-
        registration(RegKey, CKey, SKey),
        course_difficulty(CKey, Dif),
        student_intelligence(SKey, Int),
        grade_table(Int, Dif, Table),
        { Grade = grade(Key) with Table }.
```

Our goal is to find out the parameters in $Table$. In this example, we need to learn the parameters for 6 different CLP($\mathcal{BN}$) predicates. We have 32 professors, 64 courses, 256 students, and 856 registrations in the database. The database was actually generated by sampling from the CLP($\mathcal{BN}$) program, so our goal can be seen as simply trying to recover the parameters.

If all the information for the database is available (there is no missing data), then the parameters can be computed by using the maximum likelihood estimator. In other words, the parameters can be computed by simply counting the numbers of the different cases, and normalising. No Bayesian inference is required in this case.

If data is missing, we need to estimate what are the most likely values for the missing values. CLP($\mathcal{BN}$) implements a version of the Expectation Maximisation (EM) algorithm. The algorithm iterates by computing the expected values for the missing data, and then using these values, weighted by their probabilities, to obtain new estimates of missing data. The process is guaranteed to eventually converge to a local maximum. In practice, we impose a maximum number of iterations and/or stop if the improvement is below some threshold.

Missing	**Variable Elimination**			**Junction Trees**			**Gibbs Sampling**		
Data	Run Time	LL	Iters	Run Time	LL	Iters	Run Time	LL	Iters
0%	294	2085	0						
5%	330	2079	2	425	2133	2	2688	2118	2
10%	629	2100	3	4971	2133	2	5436	2138	2
20%	912	2117	3	19607	2104	3	13503	2158	2
30%	3477	2159	3	89450	2134	3	56099	2163	3
50%	43702	2141	5	313770	2098	5	128328	2161	5
90%	2319	2543	1	5165	2533	1	136929	2536	1

Table 1. Performance on School DataSet. For each method, the table shows running times in msec, log-likelihood of the data given the parameters found (LL), and number of iterations of the EM algorithm.

Clearly, the more data is missing, the more inference we should have to perform. In fact, the problem is compounded by the fact that missing data tends to result in larger networks per query: as we have less evidence we need to consult more variables to obtain an accurate answer.

The CLP($\mathcal{BN}$) implementation of EM learning therefore can call the inference routine a large number of times on large networks. To reduce overheads, the actual implementation performs two separate calls: an *initialization* call constructs the network, and preliminary data-structures. For example, with the junction-tree solver, junction trees are constructed only once. In a second call, variables are marginalised against the current parameters.

Table 1 shows performance on a MacBookPro with a 2.5 GHz Intel Core2 Duo running OSX 10.5.8, with 4GB of installed memory. We present the log-likelihood of the data, that is, the logarithm of the probability of the observed data given the learned parameters, and the number of iterations that the EM algorithm took to obtain the parameters. We vary the missing data between 0% (base-line) and 90% of all data. In this dataset, variable elimination performs very well, significantly better than the other methods. Performance tends to vary linearly with the number of marginalised variables and iterations, except for 50% noise, where one creates very large networks.

Gibbs Sampling performs second. Performance is independent on evidence: it depends only on the number of iterations and on the number of variables to marginalise. This shows one of the best advantages of Gibbs sampling: it is robust to complex dependency graphs that can result from different evidence. Unfortunately, even with pre-compilation running Gibbs sampling is slower.

Our hope in implementing junction trees was that the same structure would occur with different queries. This does not seem to be the case. Thus, we have to compile lots of different networks into different junction trees, cancelling out the benefit from using junction trees. The problem is particularly severe when there is much missing evidence, and we may have large networks. On the other hand, belief propagation is very fast, so after compilation junction trees are usually faster than variable elimination. This may make an important difference if converge is very slow.

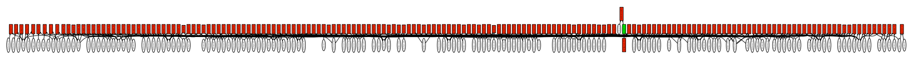

Fig. 2. A Bird's Eye View of a Bayesian Network Induced by evidence in the School Database. The picture clearly depicts the linear nature of the bayesian network graph.

We also observe that the EM algorithm seems to be doing quite well at recovering the original parameters: although likelihood tends to degrade as we remove more data, it is close to the one with full data almost up to 50% of missing data. The method cannot achieve miracles, and cannot recover the parameters if only given 10% of the full data.

Although these results show a significant advantage of variable elimination, they should be taken carefully. Fig 2 shows a bird's eye view of a graph induced by this application with 30% missing evidence (evidence nodes are shown in red, the query node in green). The graph is almost perfectly linear, making it perfect for variable elimination.

For our second experiment, we use Gene Expression Data. In this case, we have a collection of time-series data for yeast expression, and we want to generate probabilistic rules that predict gene activity [19]. We use the ILP system Aleph to generate rules. The ILP Prolog rules are then adapted to CLP($\mathcal{BN}$), which then uses the Expectation Maximisation algorithm to learn the parameters. The problem includes both learning parameters and rules.

We use the same platform as before. We have 2940 examples of time expression, corresponding to 19 different genes/proteins in yeast and to 23 different time series. In practice, learning proceeds by finding a rule for the first gene, and then adding the rules for each new gene until all genes are explained. Thus, the rules initially construct a mostly empty network, but as learning proceeds the networks will grow more and more complex. About 10% of the data is missing, but the distribution of the missing data is often not random (e.g., an experiment may be missing the time-series for a gene).

Missing	**Variable Elimination**		**Junction Trees**		**Gibbs Sampling**	
Gene	Run Time	Likelihood	Run Time	Likelihood	Run Time	Likelihood
1	9.78	2285	10.30	2285	298	2285
4	4.60	2281	5.10	2281	175	2133
8	8.81	2272	8.96	2272	248	2133
12	68.37	2264	56.66	2264	1085	2264
16	4.26	2249	9.40	2249	144	2249
Total	397.51	2237	406.79	2237	7851	2234

Table 2. Performance on Yeast DBN. We show results after gene 1, 4, 8, 12, and 16, and total execution. For each method, he table shows running times in sec, and likelihood of the data given the parameters found. Notice that the final row exhibits the total execution time.

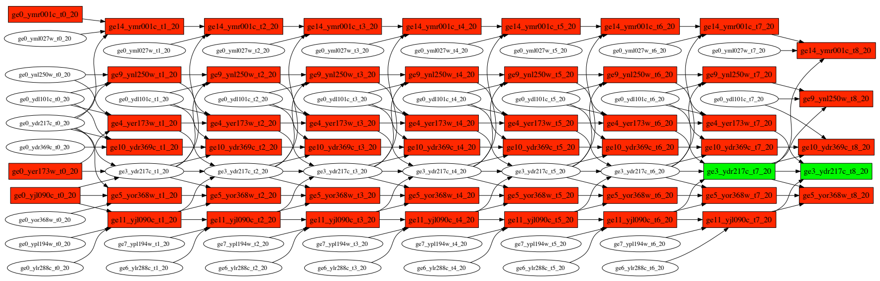

Fig. 3. An Example Dynamic Bayesian Network induced by evidence in the Yeast Expression Dataset. Red nodes are evidence nodes, and the two green nodes are the query nodes.

Table 2 shows performance in this example. Again, variable elimination does quite well, although junction trees do quite close, and in fact perform better in some cases. Gibbs sampling pays a large overhead and ends up being up to an order of magnitude slower than the other two approaches.

Figure 3 shows an example network. The network has the layered structure typical of time series and dynamic bayesian networks. The red boxes indicate evidence: the results show that the network was induced by missing evidence on expression of the genes such as `YML027W` when trying to estimate expression at the end of the time series.

8 Conclusions

We present a logic programming based implementation of a statistical relational learning system, CLP($\mathcal{BN}$), available as part of the YAP Prolog development distribution. Our results show that the system can be applied effectively to large problems, such as estimating gene pathways from gene expression data. This was possible by implementing the most expensive operations in a `C` library. We therefore benefit from the expressiveness of Prolog, while maintaining what we believe is reasonable efficiency.

The area of Statistical Relational Learning is an exciting area, and we hope that CLP($\mathcal{BN}$) will be able to contribute. Recently, there has been much interest in novel inference methods that take advantage of the relational nature of the problem [9]. We believe that CLP($\mathcal{BN}$) may be very well suited for this task. Implementing novel, complex, algorithm will also require re-factoring existing code. We believe how best to do so is an exciting problem, and hope CLP($\mathcal{BN}$) will benefit from the progress in Prolog development technology.

Acknowledgments This work has been partially supported by funds granted to CRACS through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and by the the POSC JEDI (PTDC/EIA/66924/2006), HORUS (PTDC/EIA/100897/2008) and STAMPA (PTDC/EIA/67738/2006) projects.

References

1. Poole, D.: The independent choice logic and beyond. In Raedt, L.D., Frasconi, P., Kersting, K., Muggleton, S., eds.: Probabilistic Inductive Logic Programming - Theory and Applications. Volume 4911 of LNCS., Springer (2008) 222–243
2. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of Artificial Intelligence Research **15** (2001) 391–454
3. Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Relational Data Mining. Springer (2001) 307–335
4. Muggleton, S.: Stochastic logic programs. In Raedt, L.D., ed.: Advances in Inductive Logic Programming. Volume 32 of Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (1996) 254–264
5. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning **62** (2006) 107–136
6. Raedt, L.D., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In Veloso, M.M., ed.: IJCAI 2007,, Hyderabad, India, January 6-12, 2007. (2007) 2462–2467
7. Santos Costa, V., Page, D., Qazi, M., Cussens, J.: CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilistic Knowledge. In: UAI03. (August 2003) 517–524
8. Santos Costa, V., Page, C.D., Cussens, J.: CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilistic Knowledge. In: Prob. Ind. Logic Programming. (2007)
9. Taskar, B., Getoor, L.: Introduction to Statistical Relational Learning. MIT Press (2007)
10. Raedt, L.D., Frasconi, P., Kersting, K., Muggleton, S: Probabilistic Inductive Logic Programming - Theory and Applications. Volume 4911 of LNCS. (2008)
11. Casella, G., George, E.I.: Explaining the gibbs sampler. The American Statistician **46**(3) (1992) 167–174
12. Spiegelhalter, D., Thomas, A., Best, N., Gilks, W.: BUGS 0.5 Bayesian inference using Gibbs Sampling Manual. MRC Biostatistics Unit, Cambridge. (1996)
13. Zhang, N.L., Poole, D.: Exploiting causal independence in bayesian network inference. J. Artif. Intell. Res. (JAIR) **5** (1996) 301–328
14. Murphy, K.P.: The Bayes Net Toolbox for Matlab. Computing Science and Statistics (2001)
15. Jensen, F.V.: Bayesian Networks and Decision Graphs. Springer Verlag (2001)
16. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-driven indexing of prolog clauses. In ICLP 2007. Volume 4670 of LNCS., Springer (2007) 305–409
17. Hermenegildo, M.V. et al: An overview of the ciao multiparadigm language and program development environment and its design philosophy. In: Concurrency, Graphs and Model. (2008) 209–237
18. Schrijvers, T., Costa, V.S., Wielemaker, J., Demoen, B.: Towards typed prolog. In: ICLP 2008. 693–697
19. Ong, I., Page, D., Santos Costa, V.: Inferring regulatory networks from time series expression data and relational data via inductive logic programming. In Muggleton, S., Otero, R., eds.: In-Area Short Papers for 2006 International Conference on Inductive Logic Programming. (2006)