Effective priors over model structures applied to DNA binding assay data

Netherlands Cancer Institute, Amsterdam
Nicos Angelopoulos and Lodewyk Wessels

Abstract. This paper discusses Distributional Logic Programming (Dlp), a formalism for combining logic programming and probabilistic reasoning. In particular, we focus on representing prior knowledge for Bayesian reasoning applications. We explore the representational power of the formalism for defining priors of model spaces and delve to some detail on generative priors over graphical models. We present an alternative graphical model learnt from published data. The model presented here is compared to a graphical model learnt previously from the same data.

1 Introduction

The ability to represent complex prior knowledge would greatly benefit the application of Bayesian methods as it can focus computational resources in areas that of particular interest as expressed by the prior. Furthermore, Bayesian methods provide a convenient, clean framework in which such knowledge can be incorporated. Expressing complex prior knowledge and its incorporation within Bayesian statistics is thus an important and promising line of research.

Logic programming (LP) is an attractive formalism for representing crisp knowledge. It is basis on formal mathematical logic lends it strong affinity to the very long tradition in research in knowledge representation and logical reasoning. However, the boolean nature of first-order logic sits uncomfortably with modern apporaches to epistemic inference, which statistical in nature. To remedy this, a number of probabilistic extensions to LP have been proposed. Of particular interest to this contribution are those that have been introduced for the purpose of representing Bayesian priors ([5, 1]). Here, we present an extension to the probabilistic aspects of their formalism based on probabilistic guards which have been used in more abstract probabilistic languages such as PCCP ([7]).

Currently, biology is an area of scientific knowledge that is expanding at an unprecedented rate. Vast volumes of data is being generated and knowledge in the form of scientific papers is begin accumulated. Invariably, however, statistical analysis is performed ab initio and knowledge is considered only implicitly in the form of assumptions. These can not be precise or quantative. By incorporating existing knowledge in a disciplined framework computational and statistical inference can be guided to the areas that are still lacking evidence and drive the construction of more precise models. Knowledge based data analysis is [12]

We demonstrate the usefulness of our formalism by applying MCMC inference over graphical models on a published dataset. For comparison, we use a fairly agnostic prior. The MCMC consensus graph constructed from ten MCMC chains, is in broad agreement with that learnt by the boostrapping methods described in [8] and implemented in the Banjo software [15]. The bootstrapping graph was taken from the literature [16]. The use of stronger priors would further benefit our approach, where as only simple information, such as absense/presence of specific edges can be incorporated in most other systems.

2 Preliminaries

A logic program L is a set of clauses of the form $Head$:- $Body$ defining a number of predicates. $Head$ is a single positive literal or atom, constructed from a predicate symbol and a number of term arguments. Each term is a recursively defined structure that might be an atomic value, a variable or a function constructed by an atomic function symbol and n term arguments. A query or goal G_i is a conjunction of literals $(A_{(i,1)}, \ldots, A_{(i,n)})$ which the logic engine attempts to refute against the clauses in L. This is done by employing SLD with a top to bottom scan of the clauses as the rule. Linear resolution at step i will resolve $A_{(i,1)}$ with the head (H_i) of a matching clause (M_i) and replace it with the body of the clause. The step at $i+1$ is recursively defined by applying resolution to the newly formed goal. Matching is via the unification algorithm, which when successful, provides a substitution θ_i such that $A_{(i,1)}/\theta_i = H_i$. A computation terminates when the current goal is the empty one or a failure to match any clause occurs. The logic engine can be used to explore unreached parts of the space by returning to the latest matching step and attempting to find alternative resolution clauses. The full search ends when all alternatives have been exhausted. In what follows we will use A_i to refer to $A_{(i,1)}$, i.e. the atom used for the ith resolution step. As an illustrating example of a logic program consider the following two clauses defining the $member/2$ relation (also referred to as predicate):

```
(C1) member(H, [H|T]).
(C2) member(El, [H|T]) :-
                member(El, T).
```

The first clause (C_1) states that the head of a list is one of its members, while the second one states that element El is a member of the list, if it is a member of the tail (T) of the list. Lists are convenient recursive structures term structures commonly used in logic programming to hold a collection of terms. Posed with a query of the form $? - member(X, [a, b, c])$ the LP engine will use SLD resolution which scans the query left to right and the program top to bottom as to provide all possible answers in the form of alternative values for X.

2.1 Probability Theory and Logic Programming

Logic programming implements a systematic search of a non-deterministic space. In this paper we will review some of the difficulties of mixing such spaces with probabilistic ones and present one way to achieve this. The thesis we propose is that any formalism which treats probabilities as top-level constructs, must define a single probabilistic space and in the case of logic programming a clear distribution over Θ. Current approaches to probabilistic formalisms include: the replacement of non-determinism by a probabilistic operator, the use of a primitive that appears within limited non-determinism and a clear separation of the two spaces. First, SLPs [11] under the semantics presented in [5] replace SLD resolution with sampling over pure programs that only contain stochastic clauses. An example of the second category is Prism, [14]. It provides a single probabilistic construct that instantiates an unbound variable from the elements of a list according to the probability values attached to each element. It was introduced with parameter learning in the

context of PCFGs (Probabilistic Context Free Grammars) and hidden Markov models in mind. PCLP [13] and clp(pfd(Y) [2] employ constraint programming to, in distinct ways, create two separate spaces. The non-determinism remains within the clausal level while the probabilistic is constructed in the constraint store with the constraint solver used to reason/infer from this information.

3 Syntax

We extend the clausal syntax with probabilistic guards that associate a resolution step to a probability which is computed on-the-fly. The main intuition is that in addition to the logical relation a clause defines over the objects in its head arguments it also defines a probability distribution over aspects of this relation.

Definition 1. *Probabilistic clauses in Dlp are a syntactic extension of definite clauses in LP. Let Expr be an arithmetic expression in which all variables appear in the clause-unique unary functions of the comma separated tuple $GVars$. Let Guard be a goal and $PVars$ be a comma separated tuple of variables that appear in Head. A probabilistic clause is defined by:*

$$Expr : GVars \cdot Guard \sim PVars : Head \text{ :- } Body \tag{1}$$

Arithmetic expressions of clauses defined by (1) will be evaluated at resolution time. In cases where this can be done successfully, the clauses will be used to define a distribution over the probabilistic variables ($PVars$). The distribution may depend on an arbitrary number of input terms via calls to the guard.

We also allow goals that appear in the body of clause definitions to be labelled by a tuple of unary functions each wrapping an arithmetic expression. Each of the unary functions corresponds to the functions in $GVars$. The intuition behind labelled goals in the body of clauses ($Body$) is that often probability labels of recursive calls can be easily computed from their parent call thus the interpreter can avoid recomputing all or some of the guards. For a single probabilistic predicate all clauses must define the same set of probabilistic variables. In what follows we let $C_i^{\sim}$ denote the set of probabilistic variables of clause C_i. Comparing to the already introduced $member/2$ relation, the following is a probabilistic version $pmember/2$.

(C_3) $\frac{1}{L}$: l(L) · length([H|T], L), $0 < L$ ∼H:
pmember(H, [H|T]).
(C_4) $1 - \frac{1}{L}$: l(L) · length([H|T],L), $0 < L$ ∼El:
pmember(El, [H|T]) :- l(L-1): pmember(El,T).

These clauses have attached to them expressions which will be computed at resolution time. (C_3) is labelled by $\frac{1}{L}$ where L is the length of the input list (as defined by standard predicate $list/2$ which is present in all prolog systems). (C_4) claims the residual probability. The recursive call has been augmented to carry forward the value of L as the length of T is one less than that of the input list and thus we avoid recomputing the guard. Intuitively, for the query $? - pmember(X, List)$ where $List$ is a known list, the program defines an equiprobable distribution over all the possible element selections from the list. The three corresponding probabilities when $List = [a, b, c]$ are computed as $\frac{1}{3}, \frac{2}{3} \times \frac{1}{2}, \frac{1}{3} \times \frac{1}{2} \times 1$. Clauses

(C_3) and (C_4) are written in full syntax. It is often unnecessary to be as verbose. We will drop the unary function from variables that are named with the upper case version of the functor name, that is in our example $l(L)$ reduces to L. We will share guards among clauses, thus the guard part of (C_4) can be removed. We also drop the unary functor from guard variables in body calls when either the corresponding predicate has a single guarded variable or a single guarded input variable is involved in an expression. Finally, in the interest of clarity we introduce guard lines to our programs which factor the guard section out. The example program is then:

(G_1) L ·length(List, L), $0 < L \sim$El : pmember(El,List).
(C_3') $\frac{1}{L}$: G_1 : pmember(H, [H|T]).
(C_4') $1 - \frac{1}{L}$: pmember(El, [H|T]) :-
L-1: pmember(El, T).

A distributional logic program R is the union of a set of definite clauses L and a set of distributional clauses D, defining the logical and probabilistic parts of the program respectively. D and L must define disjoint sets of predicates. Dlp grew out of the need to extend SLPs. Having labels that are set numbers has the major advantage that parameter learning can be done efficiently, [6], but can not describe complex probabilistic dependencies. For instance, $pmember/2$, as defined above cannot be capture by a simple stochastic logic program. The fact that SLPs labels are fixed numbers means that they cannot encode the uniform choice of a list element in a linear fashion by traversing the list.

4 Priors over graphical models

Hereafter, we will use the terms graphical model and BN, for Bayes nets, interchangably. A graphical model can be easily represented as a LP term. For instance, the structure of a BN with nodes $1, 2$ and 3, and two parent edges from 1 to 3 and from 2 to 3 corresponds to the term structure $[1-[3], 2-[3]]$. (Stochastic) logic programs can be written that define the space of all models, say all possible BNs with N nodes. The benefit of doing so, is that the high-level and theoretically sound properties of logic programs can provide a suitable platform for representing domain knowledge.

One approach to constructing the dependency graph of a BN is by recursively choosing parents for each of the possible nodes. Care must be taken however as to avoid introducing cycles in the graph. This method is well suited to situations where prior information regarding edges in the graph is available. The top level non-stochastic part of the selection expressed in logic programming is :

(B_2) $bn([\,], \_Nds, [\,]).$
(B_3) $bn([Nd|Nds], AllNds, BN) :-$
$parents\_of(Nd, AllNds, Pa),$
$BN = [Nd - Pa|BN],$
$bn(Nds, AllNds, BN).$

(B_1) $bn(Nds, BN) :-$
$bn(Nds, Nds, BN),$
$no\_cycles(BN).$

Given a list of possible nodes Nds that appear in BN, predicate $bn/2$ constructs a candidate graph and then checks that the graph produced includes no cycles. If that is not

the case, the program fails. Predicate $bn/3$ traverses the nodes selecting parents for each one of them from $AllNds$. When an ordering is known over the variables in the BN, its construction can proceed without checking for cycles. The ordering constraint [8] specifies that the order of nodes (Nds) is significant and that each node can only have parents from the section of the ordering that follows it.

(B_5) $bn([\,], \_AllNds, [\,])$.

(B_6) $bn([Nd|Nds], PossPa, BN) :-$
$parents\_of(Nd, PossPa, Pa)$,
$BN = [Nd - Pa|TBN]$,
$bn(Nds, [Nd|PossPa], TBN)$.

(B_4) $bn(Nds, BN) :-$
$bn(Nds, [\,], BN)$.

Clauses $(B_4 - B_6)$ provide a compact implementation for the ordering constraint. The program is also robust in relation to the probabilistic paths associated to the model instances they generate. Each model has a unique non-probabilistic part with regard to this program segment and it never leads to a failure. On the contrary clauses $(B_1 - B_3)$ lead to failure and loss of probability mass when a cycle is introduced. This can only be detected after some probability is assigned to the failed path. It is worth noting that clause (B_6) selects parents for a node from the set of possible parents rather than the set of all nodes. Also, when the ordering is not known (program $B_1 - B_3$) there is no good reason why child variables should be selected in sequential order. The following program puts the two ideas in use:

(B_8) $bn([\,], \_All, BN, BN)$.

(B_9) $bn(Nds, All, BnSoFar, BN) :-$
$pmember(Nds, Nd, RemNds)$,
$poss\_pa(Nd, BnSoFar, All, PossPa)$,
$parents\_of(Nd, PossPa, Pa)$,
$add(Nd - Pa, BnSoFar, NextBnSF)$,
$bn(RemNds, All, NextBnSF, BN)$.

(B_7) $bn(Nds, BN) :-$
$bn(Nds, Nds, [\,], BN)$.

Here the node is selected probabilistically ($pmember/3$) from the nodes still available. The selection can be either fair or biased. Clause (B_9) uses an auxiliary structure $BnSoFar$ which accumulates the graph of the BN at the current level. This is used by $poss\_pa/4$ to eliminate cycle introducing parents. Clause (B_8) terminates the recursion. Once all nodes have been assigned parents, the auxiliary structure is unified to the variable of the complete BN. A number of distributions from the literature can be fitted over the edge selection that connects children in the BN to their parents. [9] introduced $p(BN) \propto \kappa^{\delta}$ where κ is a user defined parameter and δ is the number of differing edges/arcs between BN and a 'prior network' which encapsulates the user's prior belief about the network structure. [3] suggested a generalisation of the above that allows for arbitrary weights for each missing edge: $p(BN) \propto \sum_{ij} \kappa_{ij}$. A Dlp can encode such information by simply passing a list of length n_i as as extra argument to the BN constructing clauses. Each sublist is a list of length n_j weights that can be used to call the following predicate :

(B_{10}) K_{ij}: $parent\_edge(PP, [PP|TPa], TPa)$.

(B_{11}) $1 - K_{ij}$: $parent\_edge(PP, TPa, TPa)$.

In the context of learning from expression array data, [17] constructed tabular priors over the existence of some edges. This is complementary to penalising missing edges. A similar program to the one presented above can capture such knowledge.

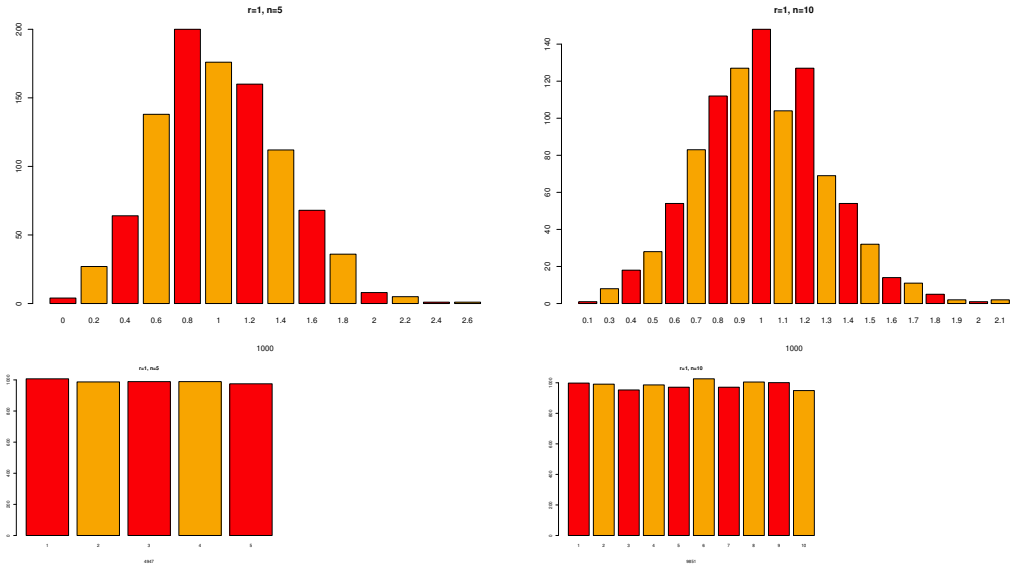

Fig. 1. 1000 samples from clauses B_{12}-B_{17}. Top left: average family size for r=1, n=5. Top right: average family size for $4 = 1, n = 10$. Bottom left: number of times node i (x-axis) was a parent, for $r = 1, n = 5$. Bottom right, as adjacently but for $r = 1, n = 10$.

Another approach proposed in [8] is that of limiting the number of parents a node may have. It seems natural that a prior distribution over the parental population maybe a suitable extension to this and in Dlp it can be written as :

$$
\begin{array}{rll}
(G_2) & L\cdot & length(PossPa, L1), L \text{ is } L1 - 1 \\
 & R\cdot & \\
 & \sim Pa : & parents\_of(PossPa, R, Pa).
\end{array}
$$

$$
\begin{array}{ll}
(B_{13})\ graph([\,], \_Nds, \_R, G). & (B_{12})\ graph(Nds, R, G) :- \\
(B_{14})\ graph([H|T], Nds, R, G) :- & \qquad graph(Nds, Nds, R, G). \\
\qquad select(Nd, Nds, PossPa), & \\
\qquad parents\_of(Nd, PossPa, Pa), & \\
\qquad G = [H - Pa|TG], & \\
\qquad graph(T, Nds, R, TG). &
\end{array}
$$

$$
\begin{array}{rll}
(B_{15}) & 1: & parents\_of([\,], \_R, [\,]) \quad . \\
(B_{16}) & \frac{R}{L}: & parents\_of([PP|PPs], R, Pa) :- \\
 & & \qquad Pa = [PP|TPa], \\
 & & L, R\text{: } parent\_of(PPs, R, TPa). \\
(B_{17}) & 1 - \frac{R}{L}: & parents\_of([\_PP|PPs], R, Pa) :- \\
 & & L, R\text{: } parent\_of(PPs, R, Pa).
\end{array}
$$

Guard (G_2) sets up L to the number of possible parents under consideration (the number of all nodes minus 1) and R to the expected number of parents per family. Clause

(B_{15}) is the base case of the recursion while (B_{16}) adds a possible parent (PP) to the list of parents (Pa) and (B_{14}) eliminates the candidate. By setting the selection probability to R/L we expect R number of parents to be selected. The top part of Figure 1 shows the average number of family size from 1000 ndependent samples from this prior. There are two different values of n, the length of all nodes list, 5 and 10 for a single value of $r = 1$. The experimental results show that indeed the prior defines a normal distribution for the average family size with mean eauql to 1. The bottom part of Figure 1 clearly shows that there is no bias in the selection of parents. The x-axis plots graph nodes and y-axis plots number of times the nodes were used as parents. Note that clauses B_{13}-B_{17} define a graph structure which might include cycles and as such not strictly a BN structure. We did so as to make the Dlp easier to follow and demonstrate the sampling distribution.

4.1 Likelihood based learning

Bayesian learning methods seek in the posterior distribution for either single models that maximise some measure or an approximation of the whole posterior. The posterior over models given some data $P(M|D)$ is proportional to the prior and a likelihood function, $P(M|D) \propto p(M)P(D|M)$. Since the space is in all but trivial examples is too large to enumerate, various approximate methods have been introduced. Variational methods [10] approximate the inference on the evidence by considering a simpler inference task. Markov chain Monte Carlo algorithms sample from the posterior indirectly So far we have concentrated on building priors that provide access to the choices via probabilistic paths. In this section we discuss one algorithm that can take advantage of the defined priors and its application to a real-world machine learning task.

4.2 Metropolis-Hastings

Metropolis-Hastings (MH) algorithms approximate the posterior by stochastic moves through the model space. A chain of visited model is constructed. At each iteration the last model added to the chain is used as a base from where a new model M' is proposed which is accepted or rejected stochastically. The distribution with which M' is reached from M is the proposal $q(M, M')$ and the acceptance probability is given by

$$\propto \frac{p(M'), P(M'|D), q(M, M')}{p(M), P(M|D), q(M', M)}$$

To our knowledge all MH algorithms in the literature, with the exception those based on SLPs, have distinct functions for computing the prior and the proposal. Standard MH requires two separate computations. The first is the prior over models: $p(M)$, and the second is a distribution for proposing a new model M' from current model M. The proposed model is accepted with probability that is proportional to the ratio given above which also includes the marginal likelihood of the model that measures the goodness of fit to the data $P(M|D)$). This often leads to restricting the choices of either the prior [8] or the proposal [4]. Furthermore, writing two programs that manipulate the same model space means that the algorithms are hard to extend to other spaces. The MH algorithm over Dlp requires the construction of a single program, that of the prior.

A generic MH algorithm for SLPs was suggested in [5] and further developed in [1]. The main idea is to use the choices in the probabilistic path as points from which alternative models can be sampled. Proposals are thus tightly coupled to the prior and take the form of a function f such that $\pi_j^M = f(\pi^M)$ where π^M is the path produced for deriving model M. π_j is the point from which M' will be sampled. As Dlp also provides a clear connection between computed instantiations and probabilistic choices the MH algorithm can be fitted over their priors. Open source software for MCMC simulations over the priors described here is web available [1].

4.3 Chromatin interaction graph

We ran the MCMCMS system on the data from [16]. A simple prior was used that fits a gamma distribution over the parents with a mean value of 1.2 for the average family. The analysis presented in [16] was to build an 80% consensus graph based on repeated simulated annealing search using the Banjo software [15, 18]. The dataset consists of 4380 rows each representing a possible biding location for each of the 43 chromatin proteins (columns). We discretised the data as per the original paper by setting the strongest 5% of the bindings for each protein to 1 and the rest to 0.

We ran 10 chains of 5×10^5 iterations each. We then averaged the results in the form of a graph by including edges that appear more time than the threshold (80%). We will refer to the graph learnt by our method as M_{80} and to the graph in [16] as BN_{80}. In Fig.2 we show BN_{80} with edges that do not appear in M_{80} highlighted in blue, whereas Fig.3 shows M_{80} with edges that do not appear in BN_{80} highlighted in orange. Bidirectional edges depict averages that could only achieve the threshold for inclusion when considering both directions of the edge in the Markov chains. For ease of comparison both graphs are given with the same topology: that of [16]. Note, that this introduces artificial visual bias, making some new edges in M_{80} appear as long range effects. The directions of arrows in BN_{80} should be ignonred as they are those of M_{80} except for those edges that are not present in M_{80} in which case the order is random.

There are 9 edges that were not in M_{80}, 6 edges that were not in $BN_{80}$and 40 common edges. M_{80} concurs with BN_{80} on all the major families. For instance the wiring of the classical heterochromatin family around HP1 is remarkably conserved. Graph M_{80} also contains all experimentally validated edges of BN_{80}. That is, the edges of HP1 with HP3 and HP6 and the edges of BRM with JRA, GAF and SU(VAR)3-7.

5 Conclusions

This paper discusses a general programming language for combining logical and probabilistic reasoning in logic programming specially for the purpose of defining prior Bayesian knowledge. The characterisation is of relevance not only to Dlp but also to other generative formalisms that combine logic and probability. We have argued that for certain classes of programs the kind of knowledge that can be represented in a convenient way is substantially improved. Furthermore, we illustrated via examples on how to write correct and efficient programs that capture knowledge from the Bayesian learning literature.

[1] http://scibsfs.bch.ed.ac.uk/ nicos/sware/dlp/mcmcms/

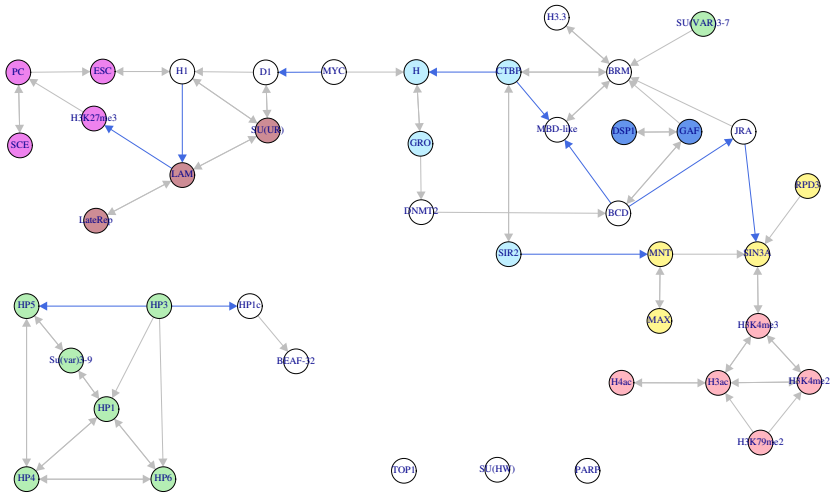

Fig. 2. Model BN_{80}, as presented in van Steensel et.al. (Ignoring edge directions.) Blue edges do not appear in M_{80}.
.

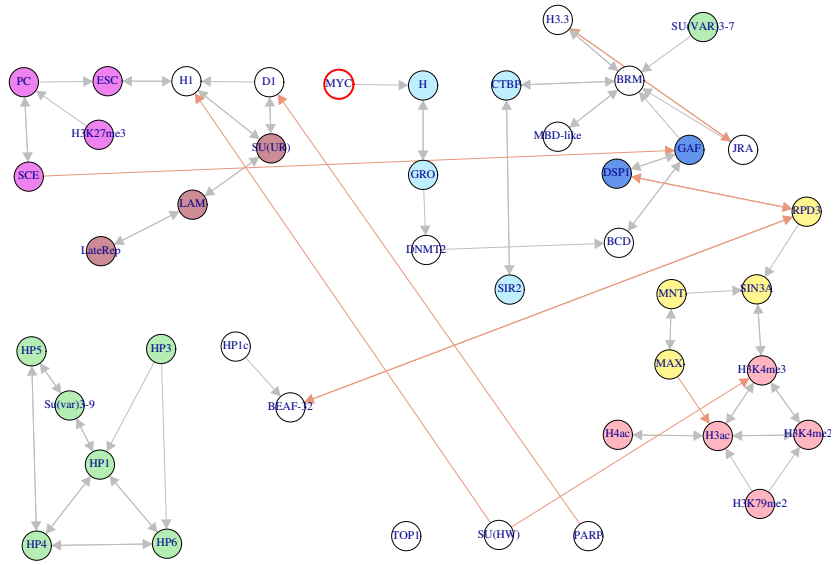

Fig. 3. Model M_{80}, learnt from 10 chains, each of 500,000 BNs. Shown interactions appear in 80% of the models. Single direction are for edges over the cut-off at one direction. Orange edges do not appear in BN_{80}.

We have used an MCMC schema over the probabilistic language to learn a graphical model from a recently published dataset. The consensus graph learnt by our method is in a

broad agreement ($\approx 80\%$) with a previously published graph. Furthermore the agreement coincides with the experimentally validated interactions.

References

1. Angelopoulos, N., Cussens, J.: MCMC using tree-based priors on model structure. In: UAI'01. pp. 16–23 (2001)
2. Angelopoulos, N., Gilbert, D.R.: A statistical view of probabilistic finite domains. In: Workshop on Quantitative Aspects of Programming Laguages (2001)
3. Buntine, W.L.: Theory refinement on BNs. In: UAI'91. pp. 52–60 (1991)
4. Chipman H, George E, M.R.: Bayesian CART model search (with discussion). Journal of the American Statistical Association 93, 935–960 (1998)
5. Cussens, J.: Stochastic logic programs: Sampling, inference and applications. In: UAI'2000. pp. 115–122 (2000)
6. Cussens, J.: Parameter estimation in stochastic logic programs. Machine Learning 44(3), 245–271 (2001)
7. Di Pierro, A., Wiklicky, H.: An operational semantics for probabilistic concurrent constraint programming. In: IEEE Comp.Soc.Conf. on Comp. Languages (1998)
8. Friedman, N., Koller, D.: Being Bayesian about network structure. In: UAI-00. pp. 201–210 (2000)
9. Heckerman, D., Geiger, D., Chickering, D.: Learning BNs: The combination of knowledge and statistical data. Machine Learning 20(3), 197–243 (1995)
10. Jordan, M.I., Ghahramani, Z., Jaakkola, T.S., Saul, L.K.: An introduction to variational methods for graphical models. In: Learning in Graphical Models. MIT Press (1999)
11. Muggleton, S.: Stochastic logic programs. In: de Raedt, L. (ed.) Advances in Inductive Logic Programming, pp. 254–264. IOS Press (1996)
12. Ochs, M.F.: Knowledge-based data analysis comes of age. Briefings in Bioinformatics 11(1), 30–39 (2010), http://bib.oxfordjournals.org/content/11/1/30.abstract
13. Riezler, S.: Probabilistic Constraint Logic Programming. Ph.D. thesis, Neuphilologische Fakultät, Universität Tübingen, Tubingen, Germany (1998)
14. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of AI Research 15, 391–454 (2001)
15. Smith, V., Jarvis, E., Hartemink, A.: evaluating functional network inference using simulations of complex biological systems. Bioinformatics 18, S216S224 (2002), intelligent Systems in Molecular Biology 2002 (ISMB02)
16. van Steensel, B., Braunschweig, U., Filion, G.J., Chen, M., van Bemmel, J.G., Ideker, T.: Bayesian network analysis of targeting interactions in chromatin. Genome Research 20(2), 190–200 (February 2010), http://dx.doi.org/10.1101/gr.098822.109
17. Werhli, A., Husmeier, D.: Reconstructing gene regulatory networks with bns by combining expression data with multiple sources of prior knowledge. Stat. App. in Genetics and Molecular Biology 6(1) (2007)
18. Yu, J., Smith, V., Wang, P., Hartemink, A., Jarvis, E.: Advances to bayesian network inference for generating causal networks from observational biological data. Bioinformatics 20, 35943603 (December 2004)