Journal of Machine Learning Research 12 (2011) 663-689 Submitted 5/10; Revised 2/11; Published 3/11

Efficient Structure Learning of Bayesian Networks using Constraints

Cassio P. de Campos CASSIOPC@ACM.ORG
Dalle Molle Institute for Artificial Intelligence
Galleria 2
Manno 6928, Switzerland

Qiang Ji JIQ@RPI.EDU
Dept. of Electrical, Computer & Systems Engineering
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, USA

Editor: David Maxwell Chickering

Abstract

This paper addresses the problem of learning Bayesian network structures from data based on score functions that are decomposable. It describes properties that strongly reduce the time and memory costs of many known methods without losing global optimality guarantees. These properties are derived for different score criteria such as Minimum Description Length (or Bayesian Information Criterion), Akaike Information Criterion and Bayesian Dirichlet Criterion. Then a branch-and-bound algorithm is presented that integrates structural constraints with data in a way to guarantee global optimality. As an example, structural constraints are used to map the problem of structure learning in Dynamic Bayesian networks into a corresponding augmented Bayesian network. Finally, we show empirically the benefits of using the properties with state-of-the-art methods and with the new algorithm, which is able to handle larger data sets than before.

Keywords: Bayesian networks, structure learning, properties of decomposable scores, structural constraints, branch-and-bound technique

1. Introduction

A Bayesian network is a probabilistic graphical model that relies on a structured dependency among random variables to represent a joint probability distribution in a compact and efficient manner. It is composed by a directed acyclic graph (DAG) where nodes are associated to random variables and conditional probability distributions are defined for variables given their parents in the graph. Learning the graph (or structure) of these networks from data is one of the most challenging problems, even if data are complete. The problem is known to be NP-hard (Chickering et al., 2003), and best exact known methods take exponential time on the number of variables and are applicable to small settings (around 30 variables). Approximate procedures can handle larger networks, but usually they get stuck in local maxima. Nevertheless, the quality of the structure plays a crucial role in the accuracy of the model. If the dependency among variables is not properly learned, the estimated distribution may be far from the *correct* one.

In general terms, the problem is to find the best structure (DAG) according to some score function that depends on the data (Heckerman et al., 1995). There are methods based on other (local) statistical analysis (Spirtes et al., 1993), but they follow a completely different approach. The re-

search on this topic is active (Chickering, 2002; Teyssier and Koller, 2005; Tsamardinos et al., 2006; Silander and Myllymaki, 2006; Parviainen and Koivisto, 2009; de Campos et al., 2009; Jaakkola et al., 2010), mostly focused on complete data. In this case, best exact ideas (where it is guaranteed to find the global best scoring structure) are based on dynamic programming (Koivisto and Sood, 2004; Singh and Moore, 2005; Koivisto, 2006; Silander and Myllymaki, 2006; Parviainen and Koivisto, 2009), and they spend time and memory proportional to $n \cdot 2^n$, where n is the number of variables. Such complexity forbids the use of those methods to a couple of tens of variables, mainly because of the memory consumption (even though time complexity is also a clear issue). Ott and Miyano (2003) devise a faster algorithm when the complexity of the structure is limited (for instance the maximum number of parents per node and the degree of connectivity of a subjacent graph). Perrier et al. (2008) use structural constraints (creating an undirected super-structure from which the undirected subjacent graph of the optimal structure must be a subgraph) to reduce the search space, showing that such direction is promising when one wants to learn structures of large data sets. Kojima et al. (2010) extend the same ideas by using new search strategies that exploit clusters of variables and ancestral constraints. Most methods are based on improving the dynamic programming method to work over reduced search spaces. On a different front, Jaakkola et al. (2010) apply a linear programming relaxation to solve the problem, together with a branch-and-bound search. Branch-and-bound methods can be effective when good bounds and cuts are available. For example, this has happened with certain success in the Traveling Salesman Problem (Applegate et al., 2006). We have proposed an algorithm that also uses branch and bound, but employs a different technique to find bounds (de Campos et al., 2009). It has been showed that branch and bound methods can handle somewhat larger networks than the dynamic programming ideas. The method is described in detail in Section 5.

In the first part of this paper, we present structural constraints as a way to reduce the search space. We explore the use of constraints to devise methods to learn specialized versions of Bayesian networks (such as naive Bayes and Tree-augmented naive Bayes) and generalized versions, such as Dynamic Bayesian networks (DBNs). DBNs are used to model temporal processes. We describe a procedure to map the structural learning problem of a DBN into a corresponding augmented Bayesian network through the use of further constraints, so that the same exact algorithm we discuss for Bayesian networks can be employed for DBNs.

In the second part, we present some properties of the problem that bring a considerable improvement on many known methods. We build on our recent work (de Campos et al., 2009) on *Akaike Information Criterion* (AIC) and *Bayesian Information Criterion* (BIC), and present new results for the Bayesian Dirichlet (BD) criterion (Cooper and Herskovits, 1992) and some derivations under a few assumptions. We show that the search space of possible structures can be reduced drastically without losing the global optimality guarantee and that the memory requirements are very small in many practical cases.

As data sets with many variables cannot be efficiently handled (unless P=NP), a desired property of a learning method is to produce an *anytime* solution, that is, the procedure, if stopped at any moment, provides an approximate solution, while if kept running, its solution improves until a global optimum is found. We point out that the term *anytime* is used to mean that the difference between best current solution and upper bound for the global optimum constantly decreases throughout the algorithm's execution (even though we cannot guarantee whether the improvement happens because a better solution is found or because the upper bound is shrunk). We describe an anytime and exact algorithm using a branch-and-bound (B&B) approach with caches. Scores are

pre-computed during an initialization step to save computational time. Then we perform the search over the possible graphs iterating over arcs. Because of the B&B properties, the algorithm can be stopped with a best current solution and an upper bound for the global optimum, which gives a certificate to the answer and allows the user to stop the computation when she/he believes that the current solution is good enough. For example, such an algorithm can be integrated with a structural Expectation-Maximization (EM) method without the huge computational expenses of other exact methods by using the generalized EM (where finding an improving solution is enough), but still guaranteeing that a global optimum is found if run until the end. Due to this property, the only source of approximation would regard the EM method itself. It worth noting that using a B&B method is not new for structure learning (Suzuki, 1996). Still, that previous idea does not constitute a global exact algorithm, instead the search is conducted after a node ordering is fixed. Our method does not rely on a predefined ordering and finds a global optimum structure considering all possible orderings.

The paper is divided as follows. Section 2 describes the notation and introduces Bayesian networks and the structure learning problem based on score functions. Section 3 presents the structural constraints that are treated in this work, and shows examples on how they can be used to learn different types of networks. Section 4 presents important properties of the score functions that considerably reduce the memory and time costs of many methods. Section 5 details our branch-and-bound algorithm, while Section 6 shows experimental evaluations of the properties, the constraints and the exact method. Finally, Section 7 concludes the paper.

2. Bayesian Networks

A Bayesian network represents a joint probability distribution over a collection of random variables, which we assume to be categorical. It can be defined as a triple $(\mathcal{G},\mathcal{X},\mathcal{P})$, where $\mathcal{G} \doteq (V_\mathcal{G}, E_\mathcal{G})$ is a directed acyclic graph (DAG) with $V_\mathcal{G}$ a collection of n nodes associated to random variables $\mathcal{X}$ (a node per variable), and $E_\mathcal{G}$ a collection of arcs; $\mathcal{P}$ is a collection of conditional mass functions $p(X_i|\Pi_i)$ (one for each instantiation of Π_i), where Π_i denotes the parents of X_i in the graph (Π_i may be empty), respecting the relations of $E_\mathcal{G}$. In a Bayesian network every variable is conditionally independent of its non-descendants given its parents (Markov condition).

We use uppercase letters such as X_i, X_j to represent variables (or nodes of the graph, which are used interchanged), and x_i to represent a generic state of X_i, which has state space $\Omega_{X_i} \doteq \{x_{i1}, x_{i2}, \ldots, x_{ir_i}\}$, where $r_i \doteq |\Omega_{X_i}| \geq 2$ is the number of (finite) categories of X_i ($|\cdot|$ is the cardinality of a set or vector, and the notation $\doteq$ is used to indicate a definition instead of a mathematical equality). Bold letters are used to emphasize sets or vectors. For example, $\mathbf{x} \in \Omega_\mathbf{X} \doteq \times_{X\in\mathbf{X}}\Omega_X$, for $\mathbf{X} \subseteq \mathcal{X}$, is an instantiation for all the variables in $\mathbf{X}$. Furthermore, $r_{\Pi_i} \doteq |\Omega_{\Pi_i}| = \prod_{X_t\in\Pi_i} r_t$ is the number of possible instantiations of the parent set Π_i of X_i, and $\theta = (\theta_{ijk})_{\forall ijk}$ is the entire vector of parameters such that the elements are $\theta_{ijk} = p(x_{ik}|\pi_{ij})$, with $i \in \{1,\ldots,n\}$, $j \in \{1,...,r_{\Pi_i}\}$, $k \in \{1,...,r_i\}$, and $\pi_{ij} \in \Omega_{\Pi_i}$.

Because of the Markov condition, the Bayesian network represents a joint probability distribution by the expression $p(\mathbf{x}) = p(x_1,\ldots,x_n) = \prod_i p(x_i|\pi_i)$, for every $\mathbf{x} \in \Omega_\mathcal{X}$, where every x_i and π_i are consistent with $\mathbf{x}$.

Given a complete data set $D = \{D_1,\ldots,D_N\}$ with N instances, where $D_u \doteq \mathbf{x}_u \in \Omega_\mathcal{X}$ is an instantiation of all the variables, the goal of structure learning is to find a DAG $\mathcal{G}$ that maximizes a given score function, that is, we look for $\mathcal{G}^* = \text{argmax}_{\mathcal{G}\in\mathcal{G}} s_D(\mathcal{G})$, with $\mathcal{G}$ the set of all DAGs with

nodes $\mathcal{X}$, for a given score function s_D (the dependency on data is indicated by the subscript D).[1] In this paper, we consider some well-known score functions: the Bayesian Information Criterion (BIC) (Schwarz, 1978) (which is equivalent to the *Minimum Description Length*), the Akaike Information Criterion (AIC) (Akaike, 1974), and the Bayesian Dirichlet (BD) (Cooper and Herskovits, 1992), which has as subcases BDe and BDeu (Buntine, 1991; Cooper and Herskovits, 1992; Heckerman et al., 1995). As done before in the literature, we assume parameter independence and modularity (Heckerman et al., 1995). The score functions based on BIC and AIC differ only in the weight that is given to the penalty term:

$$BIC/AIC: \quad s_D(\mathcal{G}) = \max_{\theta} L_{\mathcal{G},D}(\theta) - t(\mathcal{G}) \cdot w,$$

where $t(\mathcal{G}) = \sum_{i=1}^{n}(r_{\Pi_i} \cdot (r_i - 1))$ is the number of free parameters, $w = \frac{\log N}{2}$ for BIC and $w = 1$ for AIC, $L_{\mathcal{G},D}$ is the log-likelihood function with respect to data D and graph $\mathcal{G}$:

$$L_{\mathcal{G},D}(\theta) = \log \prod_{i=1}^{n} \prod_{j=1}^{r_{\Pi_i}} \prod_{k=1}^{r_i} \theta_{ijk}^{n_{ijk}},$$

where n_{ijk} indicates how many elements of D contain both x_{ik} and π_{ij}. Note that the values $(n_{ijk})_{\forall ijk}$ depend on the graph $\mathcal{G}$ (more specifically, they depend on the parent set Π_i of each X_i), so a more precise notation would be to use $n_{ijk}^{\Pi_i}$ instead of n_{ijk}. We avoid this heavy notation for simplicity unless necessary in the context. Moreover, we know that $\theta^* = (\theta^*_{ijk})_{\forall ijk} = (\frac{n_{ijk}}{n_{ij}})_{\forall ijk} = \text{argmax}_{\theta} L_{\mathcal{G},D}(\theta)$, with $n_{ij} = \sum_k n_{ijk}$.[2]

In the case of the BD criterion, the idea is to compute a score based on the posterior probability of the structure $p(\mathcal{G}|D)$. For that purpose, the following score function is used:

$$BD: \quad s_D(\mathcal{G}) = \log\left(p(\mathcal{G}) \cdot \int p(D|\mathcal{G},\theta) \cdot p(\theta|\mathcal{G}) d\theta\right),$$

where the logarithmic is often used to simplify computations, $p(\theta|\mathcal{G})$ is the prior of θ for a given graph $\mathcal{G}$, assumed to be a Dirichlet with hyper-parameters $\alpha = (\alpha_{ijk})_{\forall ijk}$ (which are assumed to be strictly positive):

$$p(\theta|\mathcal{G}) = \prod_{i=1}^{n} \prod_{j=1}^{r_{\Pi_i}} \Gamma(\alpha_{ij}) \prod_{k=1}^{r_i} \frac{\theta_{ijk}^{\alpha_{ijk}-1}}{\Gamma(\alpha_{ijk})},$$

where $\alpha_{ij} = \sum_k \alpha_{ijk}$. Hyper-parameters $(\alpha_{ijk})_{\forall ijk}$ also depend on the graph $\mathcal{G}$, and we indicate it by $\alpha_{ijk}^{\Pi_i}$ if necessary in the context. From now on, we also omit the subscript D. We assume that there is no preference for any graph, so $p(\mathcal{G})$ is uniform and vanishes in the computations. Under the assumptions, it has been shown (Cooper and Herskovits, 1992) that for multinomial distributions,

$$s(\mathcal{G}) = \log \prod_{i=1}^{n} \prod_{j=1}^{r_{\Pi_i}} \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})}.$$

The BDe score (Heckerman et al., 1995) assumes that $\alpha_{ijk} = \alpha^* \cdot p(\theta_{ijk}|\mathcal{G})$, where α^* is the hyper-parameter known as the Equivalent Sample Size (ESS), and $p(\theta_{ijk}|\mathcal{G})$ is the prior probability for

1. In case of many optimal DAGs, then we assume to have no preference and argmax returns one of them.
2. If $n_{ij} = 0$, then $n_{ijk} = 0$ and we assume the fraction $\frac{n_{ijk}}{n_{ij}}$ to be equal to one.

$(x_{ik} \wedge \pi_{ij})$ given $\mathcal{G}$ (or simply given Π_i). The BDeu score (Buntine, 1991; Cooper and Herskovits, 1992) assumes further that local priors are such that α_{ijk} becomes $\frac{\alpha^*}{r_{\Pi_i} r_i}$ and α^* is the only free hyper-parameter.

An important property of all such criteria is that their functions are decomposable and can be written in terms of the local nodes of the graph, that is, $s(\mathcal{G}) = \sum_{i=1}^{n} s_i(\Pi_i)$, such that

$$BIC/AIC: \quad s_i(\Pi_i) = \max_{\theta_i} L_{\Pi_i}(\theta_i) - t_i(\Pi_i) \cdot w, \tag{1}$$

where $L_{\Pi_i}(\theta_i) = \sum_{j=1}^{r_{\Pi_i}} \sum_{k=1}^{r_i} n_{ijk} \log \theta_{ijk}$, and $t_i(\Pi_i) = r_{\Pi_i} \cdot (r_i - 1)$. And similarly,

$$BD: \quad s_i(\Pi_i) = \sum_{j=1}^{r_{\Pi_i}} \left(\log \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} + \sum_{k=1}^{r_i} \log \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})} \right). \tag{2}$$

In the case of BIC and AIC, Equation (1) is used to compute the global score of a graph using the local scores at each node, while Equation (2) is employed for BD, BDe and BDeu, using the respective hyper-parameters α.

3. Structural Constraints

A way to reduce the space of possible DAGs is to consider some constraints provided by experts. We work with structural constraints that specify where arcs may or may not be included. These constraints help to reduce the search space and are available in many situations. Moreover, we show examples in Sections 3.1 and 3.2 of how these constraints can be used to learn structures of different types of networks, such as naive Bayes, tree-augmented naive Bayes, and Dynamic Bayesian networks. We work with the following rules, used to build up the structural constraints:

- $indegree(X_j, k, op)$, where $op \in \{\text{lt}, \text{eq}\}$ and k an integer, means that the node X_j must have *less than* (when $op = \text{lt}$) or *equal to* (when $op = \text{eq}$) k parents.
- $arc(X_i, X_j)$ indicates that the node X_i must be a parent of X_j.
- Operators *or* ($\vee$) and *not* ($\neg$) are used to form the rules. The *and* operator is not explicitly used as we assume that each constraint is in disjunctive normal form.

The structural constraints can be imposed locally as long as they involve just a single node and its parents. In essence, parent sets of a node X_i that do violate a constraint are never processed nor stored, and this can be checked locally when one is about to compute the local score. On the other hand, constraints such as $(arc(X_1, X_2) \vee arc(X_2, X_3))$ cannot be imposed locally, as it defines a non-local condition (the arcs go to distinct variables, namely X_2 and X_3). In this work we assume that constraints are local. Besides constraints devised by an expert, one might use constraints to force the learning procedure to obtain specialized types of networks. The next two subsections describe (somewhat non-trivial) examples of use of constraints to learn different types of networks. Specialized networks tend to be easier to learn, because the search space is already reduced to the structures that satisfy the underlying constraints. Notwithstanding, the readers who are only interested in learning general Bayesian networks might want to skip the rest of this section and continue from Section 4.

3.1 Learning Naive and TAN structures

For example, the constraints $\forall_{i\neq c, j\neq c}\ \neg arc(X_i, X_j)$ and $indegree(X_c, 0, \text{eq})$ impose that only arcs from node X_c to the others are possible, and that X_c is a root node, that is, a Naive Bayes structure will be learned. A learning procedure would in fact act as a feature selection procedure by letting some variables unlinked. Note that the symbol $\forall$ just employed is not part of the language but is used for easy of expose (in fact it is necessary to write down every constraint defined by such construction). As another example, the constraints $\forall_{j\neq c}\ indegree(X_j, 3, \text{lt})$, $indegree(X_c, 0, \text{eq})$, and $\forall_{j\neq c}\ indegree(X_j, 0, \text{eq}) \vee arc(X_c, X_j)$ ensure that all nodes have X_c as parent, or no parent at all. Besides X_c, each node may have at most one other parent, and X_c is a root node. This learns the structure of a Tree-augmented Naive (TAN) classifier, also performing a kind of feature selection (some variables may end up unlinked). In fact, it learns a forest of trees, as we have not imposed that all variables must be linked. In Section 6 we present some experimental results which indicate that learning TANs is a much easier (still very important) practical situation.

We point out that learning structures of networks with the particular purpose of building a classifier can be also tackled by other score functions that consider conditional distributions (Pernkopf and Bilmes, 2005). Here we present a way to learn TANs considering the fit of the joint distribution, which can be done by constraints. Further discussions about learning classifiers is not the aim of this work.

3.2 Learning Dynamic Bayesian Networks

A more sophisticated application of structural constraints is presented in this section, where they are employed to translate the structure learning in Dynamic Bayesian Networks (DBNs) to a corresponding problem in Bayesian networks. While Bayesian networks are not directly related to time, DBNs are used to model temporal processes. Assuming Markovian and stationary properties, DBNs may be encoded in a very compact way and inferences are executed quickly. They are built over a collection of sets of random variables $\{\mathcal{X}^0, \mathcal{X}^1, \ldots, \mathcal{X}^T\}$ representing variables in different times $0, 1, \ldots, T$ (we assume that time is discrete). A Markovian property holds, which ensures that $p(\mathcal{X}^{t+1}|\mathcal{X}^0, \ldots \mathcal{X}^t) = p(\mathcal{X}^{t+1}|\mathcal{X}^t)$, for $0 \leq t < T$. Furthermore, because the process is assumed to be stationary, we have that $p(\mathcal{X}^{t+1}|\mathcal{X}^t)$ is independent of t, that is, $p(\mathcal{X}^{t+1}|\mathcal{X}^t) = p(\mathcal{X}^{t'+1}|\mathcal{X}^{t'})$ for any $0 \leq t, t' < T$. This means that a DBN is just as a collection of Bayesian networks that share the same structure and parameters (apart from the initial Bayesian network for time zero). If $X_i^t \in \mathcal{X}^t$ are the variables at time t, a DBN may have arcs between nodes X_i^t of the same time t and arcs from nodes X_i^{t-1} (previous time) to nodes X_i^t of time t. Hence, a DBN can be viewed as two-slice temporal Bayesian network, where at time zero, we have a standard Bayesian network as in Section 2, which we denote $\mathcal{B}^o$, and for slices 1 to T we have another Bayesian network (called *transitional* Bayesian network and denoted simply $\mathcal{B}$) defined over the same variables but where nodes may have parents on two consecutive slices, that is, $\mathcal{B}$ precisely defines the distributions $p(\mathcal{X}^{t+1}|\mathcal{X}^t)$, for any $0 \leq t < T$.

To learn a DBN, we assume that many temporal sequences of data are available. Thus, a complete data set $D = \{D_1, \ldots, D_N\}$ is composed of N sequences, where each D_u is composed of instances $D_u^t \doteq \mathbf{x_u^t} = \{x_{u,1}^t, \ldots, x_{u,n}^t\}$, for $t = 0, \ldots, T$ (where T is the total number of slices/frames apart from the initial one). Note that there is an implicit order among the elements of each D_u. We denote by $D^0 \doteq \{D_u^0 :\ 1 \leq u \leq N\}$ the data of the first slice, and by $D^t \doteq \{(D_u^t, D_u^{t-1}) :\ 1 \leq u \leq N\}$, with $1 \leq t \leq T$, the data of a slice t (note that the data of the slice $t-1$ is also included, be-

cause it is necessary for learning the transitions). As the conditional probability distributions for time $t > 0$ share the same parameters, we can unroll the DBN to obtain the factorization $p(\mathcal{X}^{1:T}) = \prod_i p^0(X_i^0|\Pi_i^0) \prod_{t=1}^{T} \prod_i p(X_i^t|\Pi_i^t)$, where $p^0(X_i^0|\Pi_i^0)$ are the local conditional distributions of $\mathcal{B}^0$, X_i^t and Π_i^t represent the corresponding variables in time t, and $p(X_i^t|\Pi_i^t)$ are the local distributions of $\mathcal{B}$.

Unfortunately learning a DBN is at least as hard as learning a Bayesian network, because the former can be viewed as a generalization of the latter. Still, we show that the same method used for Bayesian networks can be used to learn DBNs. With complete data, learning parameters of DBNs is similar to learning parameters of Bayesian networks, but we deal with counts n_{ijk} for both $\mathcal{B}^0$ and $\mathcal{B}$. The counts related to $\mathcal{B}^0$ are obtained from the first slice of each sequence, so there are N samples overall, while counts for $\mathcal{B}$ are obtained from the whole time sequences, so there are $N \cdot T$ elements to consider (supposing that each sequence has the same length T, for ease of expose). The score function of a given structure decomposes between the score function of $\mathcal{B}^0$ and the score function of $\mathcal{B}$ (because of the decomposability of score functions), so we look for graphs such that

$$(\mathcal{G}^{0*}, \mathcal{G}'^{*}) = \underset{\mathcal{G}^0, \mathcal{G}'}{\operatorname{argmax}} \left(s_{D^0}(\mathcal{G}^0) + s_{D^{1:T}}(\mathcal{G}') \right) = (\underset{\mathcal{G}^0}{\operatorname{argmax}}\, s_{D^0}(\mathcal{G}^0), \underset{\mathcal{G}'}{\operatorname{argmax}}\, s_{D^{1:T}}(\mathcal{G}')), \tag{3}$$

where G^0 is a graph over $\mathcal{X}^0$ and $\mathcal{G}'$ is a graph over variables $\mathcal{X}^t, \mathcal{X}^{t-1}$ of a generic slice t and its predecessor $t-1$. Counts are obtained from data sets with time sequences separately for the initial and the transitional Bayesian networks, and the problem reduces to the learning problem in a Bayesian network with some constraints that force the arcs to respect the DBN's stationarity and Markovian characteristics (of course, it is necessary to obtain the counts from the data in a particular way). We make use of the constraints defined in Section 3 to develop a simple transformation of the structure learning problem to a corresponding structure learning problem in an augmented Bayesian network. The steps of this procedure are as follows:

1. Learn $\mathcal{B}^0$ using the data set D^0. Note that this is already a standard Bayesian network structure learning problem, so we obtain the graph $\mathcal{G}^0$ for the first maximization of Equation (3).

2. Suppose there is a Bayesian network $\mathcal{B}' = (\mathcal{G}', \mathcal{X}', \mathcal{P}')$ with twice as many nodes as $\mathcal{B}^0$. Denote the nodes as $(X_1, \ldots, X_n, X_1', \ldots, X_n')$. Construct a new data set D' that is composed by $N \cdot T$ elements $\{D^1, \ldots, D^T\}$. Note that D' is precisely a data set over $2n$ variables, because it is formed of pairs (D_u^{t-1}, D_u^t), which are complete instantiations for the variables of $\mathcal{B}'$, containing the elements of two consecutive slices.

3. Include structural constraints as follows:

$$\forall_{1 \le i \le n}\ arc(X_i, X_i'), \tag{4}$$

$$\forall_{1 \le i \le n}\ indegree(X_i, 0, eq). \tag{5}$$

   Equation (4) forces the time relation between the same variable in consecutive time slices (in fact this constraint might be discarded if someone does not want to enforce each variable to be correlated to itself of the past slice). Equation (5) forces the variables $X_1, \ldots, X_n$ to have no parents (these are the variables that are simulating the previous slice, while the variables $\mathcal{X}'$ are simulating the current slice).

4. Learn $\mathcal{B}'$ using the data set D' with an standard Bayesian network structure learning procedure, capable of enforcing the structural constraints. Note that the parent sets of $X_1,\ldots,X_n$ are already fixed to be empty, so the output graph will maximize the scores associated only to nodes $\mathcal{X}'$: $\text{argmax}_{\mathcal{G}'} s_{D^{1:T}}(\mathcal{G}')) =$

$$\underset{\mathcal{G}'}{\text{argmax}} \left(\sum_i s_{i,D^{1:T}}(\Pi_i) + \sum_{i'} s_{i',D^{1:T}}(\Pi'_i) \right) = \underset{\mathcal{G}'}{\text{argmax}} \sum_{i'} s_{i',D^{1:T}}(\Pi'_i).$$

This holds because of the decomposability of the score function among nodes, so that the scores of the nodes $X_1,\ldots,X_n$ are fixed and can be disregarded in the maximization (they are constant).

5. Take the subgraph of $\mathcal{G}'$ corresponding to the variables $X'_1,\ldots,X'_n$ to be the graph of the transitional Bayesian network $\mathcal{B}$. This subgraph has arcs among $X'_1,\ldots,X'_n$ (which are arcs correlating variables of the same time slice) as well as arcs from the previous slice to the nodes $X'_1,\ldots,X'_n$.

Therefore, after applying this transformation, the structure learning problem in a DBN can be performed by two calls to the method that solves the problem in a Bayesian network. We point out that an expert may create her/his own constraints to be used during the learning, besides those constraints introduced by the transformation, as long as such constraints do not violate the DBN implicit constraints. This makes possible to learn DBNs together with expert's knowledge in the form of structural constraints.

4. Properties of the Score Functions

In this section we present mathematical properties that are useful when computing score functions. Local scores need to be computed many times to evaluate the candidate graphs when we look for the best graph. Because of decomposability, we can avoid to compute such functions several times by creating a cache that contains $s_i(\Pi_i)$ for each X_i and each parent set Π_i. Note that this cache may have an exponential size on n, as there are 2^{n-1} subsets of $\{X_1,\ldots,X_n\}\setminus\{X_i\}$ to be considered as parent sets. This gives a total space and time of $O(n\cdot 2^n\cdot v)$ to build the cache, where v is the worst-case asymptotic time to compute the local score function at each node.[3] Instead, we describe a collection of results that are used to obtain much smaller caches in many practical cases.

First, Lemma 1 is quite simple but very useful to discard elements from the cache of each node X_i. It holds for all score functions that we treat in this paper. It was previously stated in Teyssier and Koller (2005) and de Campos et al. (2009), among others.

Lemma 1 *Let X_i be a node of $\mathcal{G}'$, a candidate DAG for a Bayesian network where the parent set of X_i is Π'_i. Suppose $\Pi_i \subset \Pi'_i$ is such that $s_i(\Pi_i) > s_i(\Pi'_i)$ (where s is one of BIC, AIC, BD or derived criteria). Then Π'_i is not the parent set of X_i in an optimal DAG $\mathcal{G}^*$.*

Proof This fact comes straightforward from the decomposability of the score functions. Take a graph $\mathcal{G}$ that differs from $\mathcal{G}'$ only on the parent set of X_i, where it has Π_i instead of Π'_i. Note that $\mathcal{G}$

3. Note that the time to compute a single local score might be large depending on the number of parents but still asymptotically bounded by the data set size.

is also a DAG (as $\mathcal{G}$ is a subgraph of $\mathcal{G}'$ built from the removal of some arcs, which cannot create cycles) and $s(\mathcal{G}) = \sum_{j \neq i} s_j(\Pi'_j) + s_i(\Pi_i) > \sum_{j \neq i} s_j(\Pi'_j) + s_i(\Pi'_i) = s(\mathcal{G}')$. Any DAG $\mathcal{G}'$ with parent set Π'_i for X_i has a subgraph $\mathcal{G}$ with a better score than that of $\mathcal{G}'$, and thus Π'_i is not the optimal parent configuration for X_i in $\mathcal{G}^*$. ■

Unfortunately Lemma 1 does not tell us anything about supersets of Π'_i, that is, we still need to compute scores for all the possible parent sets and later verify which of them can be removed. This would still leave us with $n \cdot 2^n \cdot v$ asymptotic time and space requirements (although the space would be reduced after applying the lemma). The next two subsections present results to avoid all such computations. BIC and AIC are treated separately from BD and derivatives (reasons for that will become clear in the derivations).

4.1 BIC and AIC Score Properties

Next theorems handle the issue of having to compute scores for all possible parent sets, when one is using BIC or AIC criteria. BD scores are dealt later on.

Theorem 2 *Using BIC or AIC as score function, suppose that X_i, Π_i are such that $r_{\Pi_i} > \frac{N}{w} \frac{\log r_i}{r_i - 1}$. If Π'_i is a proper superset of Π_i, then Π'_i is not the parent set of X_i in an optimal structure.*

Proof [4] We know that Π'_i contains at least one additional node, that is, $\Pi'_i \supseteq \Pi_i \cup \{X_e\}$ and $X_e \notin \Pi_i$. Because $\Pi_i \subset \Pi'_i$, $L_i(\Pi'_i)$ is certainly greater than or equal to $L_i(\Pi_i)$, and $t_i(\Pi'_i)$ will certainly be greater than the corresponding value $t_i(\Pi_i)$ in $\mathcal{G}$. The difference in the scores is $s_i(\Pi'_i) - s_i(\Pi_i)$, which equals to (see the explanations after the formulas):

$$\begin{aligned}
\max_{\theta'_i} L_i(\Pi'_i) - t_i(\Pi'_i) - (\max_{\theta_i} L_i(\Pi_i) - t_i(\Pi_i)) \leq \\
- \max_{\theta_i} L_i(\Pi_i) - t_i(\Pi'_i) + t_i(\Pi_i) = \\
\sum_{j=1}^{r_{\Pi_i}} n_{ij} \left(- \sum_{i=1}^{r_i} \frac{n_{ijk}}{n_{ij}} \log \frac{n_{ijk}}{n_{ij}} \right) - t_i(\Pi'_i) + t_i(\Pi_i) \leq \\
\sum_{j=1}^{r_{\Pi_i}} n_{ij} H(\theta_{ij}) - t_i(\Pi'_i) + t_i(\Pi_i) \leq \\
\sum_{j=1}^{r_{\Pi_i}} n_{ij} \log r_i - r_{\Pi_i} \cdot (r_e - 1) \cdot (r_i - 1) \cdot w \leq \\
\sum_{j=1}^{r_{\Pi_i}} n_{ij} \log r_i - r_{\Pi_i} \cdot (r_i - 1) \cdot w = N \log r_i - r_{\Pi_i} \cdot (r_i - 1) \cdot w.
\end{aligned}$$

The first step uses the fact that $L_i(\Pi'_i)$ is negative, so we drop it, the second step uses the fact that $\theta^*_{ijk} = \frac{n_{ijk}}{n_{ij}}$, with $n_{ij} = \sum_{i=1}^{r_i} n_{ijk}$, the third step uses the definition of entropy $H(\cdot)$ of a discrete distribution, and the fourth step uses the fact that the entropy of a discrete distribution is less than the log of its number of categories. Finally, the last equation is negative if $r_{\Pi_i} \cdot (r_i - 1) \cdot w > N \log r_i$, which

4. Another similar proof appears in Bouckaert (1994), but it leads directly to the conclusion of Corollary 3. The intermediate result is algorithmically important.

is exactly the hypothesis of the theorem. Hence $s_i(\Pi_i') < s_i(\Pi_i)$, and Lemma 1 guarantees that Π_i' cannot be the parent set of X_i in an optimal structure. ■

Corollary 3 *Using BIC or AIC as criterion, the optimal graph $\mathcal{G}$ has at most $O(\log N)$ parents per node.*

Proof Assuming $N > 4$, we have $\frac{\log r_i}{w(r_i-1)} < 1$ (because w is either 1 or $\frac{\log N}{2}$). Take a variable X_i and a parent set Π_i with exactly $\lceil \log_2 N \rceil$ elements. Because every variable has at least two states, we know that $r_{\Pi_i} \geq 2^{|\Pi_i|} \geq N > \frac{N}{w}\frac{\log r_i}{r_i-1}$, and by Theorem 2 we know that no proper superset of Π_i can be an optimal parent set. ■

Theorem 2 and Corollary 3 ensures that the cache stores at most $O(\sum_{t=0}^{\lceil \log_2 N \rceil} \binom{n-1}{t})$ elements for each variable (all combinations up to $\lceil \log_2 N \rceil$ parents). Next lemma does not help us to improve the theoretical size bound that is achieved by Corollary 3, but it is quite useful in practice because it is applicable even in cases where Theorem 2 is not, implying that fewer parent sets need to be inspected.

Theorem 4 *Let BIC or AIC be the score criterion and let X_i be a node with $\Pi_i \subset \Pi_i'$ two possible parent sets such that $t_i(\Pi_i') + s_i(\Pi_i) > 0$. Then Π_i' and all supersets $\Pi_i'' \supset \Pi_i'$ are not optimal parent configurations for X_i.*

Proof We have that $t_i(\Pi_i') + s_i(\Pi_i) > 0 \Rightarrow -t_i(\Pi_i') - s_i(\Pi_i) < 0$, and because $L_i(\cdot)$ is a negative function, it implies

$$\Rightarrow (L_i(\Pi_i') - t_i(\Pi_i')) - s_i(\Pi_i) < 0 \Rightarrow s_i(\Pi_i') < s_i(\Pi_i).$$

Using Lemma 1, we have that Π_i' is not the optimal parent set for X_i. The result also follows for any $\Pi_i'' \supset \Pi_i$, as we know that $t_i(\Pi_i'') > t_i(\Pi_i')$ and the same argument suffices. ■

Theorem 4 provides a bound to discard parent sets without even inspecting them. The idea is to verify the assumptions of Theorem 4 every time the score of a parent set Π_i of X_i is about to be computed by taking the best score of any subset and testing it against the theorem. Only subsets that have been checked against the structural constraints can be used, that is, a subset with high score but that violates constraints cannot be used as the "certificate" to discard its supersets (in fact, it is not a valid parent set at first). This ensures that the results are valid even in the presence of constraints. Whenever the theorem can be applied, Π_i is discard and all its supersets are not even inspected. This result allows us to stop computing scores earlier than the worst-case, reducing the number of computations to build and store the cache. Π_i is also checked against Lemma 1 (which is stronger in the sense that instead of a bounding function, the actual scores are directly compared). However Lemma 1 cannot help us to avoid analyzing the supersets of Π_i.

4.2 BD Score Properties

First note that the BD scores can be rewritten as:

$$s_i(\Pi_i) = \sum_{j \in J_i} \left(\log \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} + \sum_{k \in K_{ij}} \log \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})} \right),$$

where $J_i \doteq J_i^{\Pi_i} \doteq \{1 \leq j \leq r_{\Pi_i} :\ n_{ij} \neq 0\}$, because $n_{ij} = 0$ implies that all terms cancel each other. In the same manner, $n_{ijk} = 0$ implies that the terms of the internal summation cancel out, so let $K_{ij} \doteq K_{ij}^{\Pi_i} \doteq \{1 \leq k \leq r_i :\ n_{ijk} \neq 0\}$ be the indices of the categories of X_i such that $n_{ijk} \neq 0$. Let $K_i^{\Pi_i} \doteq \cup_j K_{ij}^{\Pi_i}$ be a vector with all indices corresponding to non-zero counts for Π_i (note that the symbol $\cup$ must be seen as a concatenation of vectors, as we allow $K_i^{\Pi_i}$ to have repetitions). The counts n_{ijk} (and consequently $n_{ij} = \sum_k n_{ijk}$) are completely defined if we know the parent set Π_i. Rewrite the score as follows:

$$s_i(\Pi_i) = \sum_{j \in J_i} \left(f(K_{ij}, (\alpha_{ijk})_{\forall k}) + g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) \right),$$

with

$$f(K_{ij}, (\alpha_{ijk})_{\forall k}) = \log\Gamma(\alpha_{ij}) - \sum_{k \in K_{ij}} \log\Gamma(\alpha_{ijk}),$$
$$g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) = -\log\Gamma(\alpha_{ij} + n_{ij}) + \sum_{k \in K_{ij}} \log\Gamma(\alpha_{ijk} + n_{ijk}).$$

We do not need K_{ij} as argument of $g(\cdot)$ because the set of non-zero n_{ijk} is known from the counts $(n_{ijk})_{\forall k}$ that are already available as arguments of $g(\cdot)$. To achieve the desired theorem that will be able to reduce the computational time to build the cache, some intermediate results are necessary.

Lemma 5 *Let Π_i be the parent set of X_i, $(\alpha_{ijk})_{\forall ijk} > 0$ be the hyper-parameters, and integers $(n_{ijk})_{\forall ijk} \geq 0$ be counts obtained from data. We have that $g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) \leq -\log\Gamma(v) \approx 0.1214$ if $n_{ij} \geq 1$, where $v = \text{argmax}_{x>0} -\log\Gamma(x) \approx 1.4616$. Furthermore, $g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) \leq -\log\alpha_{ij} + \log\alpha_{ijk} - f(K_{ij}, (\alpha_{ijk})_{\forall k})$ if $|K_{ij}| = 1$.*

Proof We use the relation $\Gamma(x + \sum_k a_k) \geq \Gamma(x+1)\prod_k \Gamma(a_k)$, for $x \geq 0$, $\forall_k a_k \geq 1$ and $\sum_k a_k \geq 1$ (note that it is valid even if there is a single element in the summation). This relation comes from the Beta function inequality:

$$\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} \leq \frac{x+y}{xy} \implies \Gamma(x+1)\Gamma(y+1) \leq \Gamma(x+y+1),$$

where $x, y > 0$. Applying the transformation $y + 1 = \sum_t a_t$ (which is possible because $\sum_t a_t > 1$ and thus $y > 0$), we obtain:

$$\Gamma(x + \sum_t a_t) \geq \Gamma(x+1)\Gamma(\sum_t a_t) \geq \Gamma(x+1)\prod_t \Gamma(a_t),$$

(the last step is due to $a_t \geq 1$ for all t, so the same relation of the Beta function can be overall applied, because $\Gamma(x+1)\Gamma(y+1) \leq \Gamma(x+y+1) \leq \Gamma(x+1+y+1)$).

With the relation just devised in hands, we have

$$\frac{\Gamma(\alpha_{ij} + n_{ij})}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} = \frac{\Gamma(\sum_{1 \leq k \leq r_i} (\alpha_{ijk} + n_{ijk}))}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} =$$

$$= \frac{\Gamma(\sum_{k \notin K_{ij}} \alpha_{ijk} + \sum_{k \in K_{ij}} (\alpha_{ijk} + n_{ijk}))}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} \geq \Gamma(1 + \sum_{k \notin K_{ij}} \alpha_{ijk}),$$

obtained by renaming $x = \sum_{k \notin K_{ij}} \alpha_{ijk}$ and $a_k = \alpha_{ijk} + n_{ijk}$ (we have that $\sum_{k \in K_{ij}} (\alpha_{ijk} + n_{ijk}) \geq n_{ij} \geq 1$ and each $a_k \geq 1$). Thus

$$g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) = -\log \frac{\Gamma(\alpha_{ij} + n_{ij})}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} \leq -\log \Gamma(1 + \sum_{k \notin K_{ij}} \alpha_{ijk}).$$

Because $v = \text{argmax}_{x>0} - \log \Gamma(x)$, we have $-\log \Gamma(1 + \sum_{k \notin K_{ij}} \alpha_{ijk}) \leq -\log \Gamma(v)$.

Now, the second part of the lemma. If $|K_{ij}| = 1$, then let $K_{ij} = \{k\}$. We know that $n_{ij} \geq 1$ and thus

$$g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) = -\log \frac{\Gamma(\alpha_{ij} + n_{ij})}{\Gamma(\alpha_{ijk} + n_{ij})} = -\log \left(\frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ijk})} \prod_{t=0}^{n_{ij}-1} \frac{(\alpha_{ij} + t)}{(\alpha_{ijk} + t)} \right) =$$

$$= -f(K_{ij}, (\alpha_{ijk})_{\forall k}) - \log \frac{\alpha_{ij}}{\alpha_{ijk}} - \sum_{t=1}^{n_{ij}-1} \log \frac{(\alpha_{ij} + t)}{(\alpha_{ijk} + t)} \leq -\log \alpha_{ij} + \log \alpha_{ijk} - f(K_{ij}, (\alpha_{ijk})_{\forall k}),$$

because $\frac{(\alpha_{ij}+t)}{(\alpha_{ijk}+t)} \geq 1$ for every t. ■

Lemma 6 *Let Π_i be the parent set of X_i, $(\alpha_{ijk})_{\forall ijk} > 0$ be the hyper-parameters, and integers $(n_{ijk})_{\forall ijk} \geq 0$ be counts obtained from data. We have that $g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) \leq 0$ if $n_{ij} \geq 2$.*

Proof If $n_{ij} \geq 2$, we use the relation $\Gamma(x + \sum_k a_k) \geq \Gamma(x+2) \prod_k \Gamma(a_k)$, for $x \geq 0$, $\forall_k a_k \geq 1$ and $\sum_k a_k \geq 2$. This inequality is obtained in the same way as in Lemma 5, but using a tighter Beta function bound:

$$\mathcal{B}(x,y) \leq \frac{x+y}{xy} \left(\frac{(x+1)(y+1)}{x+y+1} \right)^{-1} \implies \Gamma(x+2)\Gamma(y+2) \leq \Gamma(x+y+2),$$

and the relation follows by using $y + 2 = \sum_t a_t$ and the same derivation as before. Now,

$$\frac{\Gamma(\alpha_{ij} + n_{ij})}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} = \frac{\Gamma(\sum_{1 \leq k \leq r_i} (\alpha_{ijk} + n_{ijk}))}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} =$$

$$= \frac{\Gamma(\sum_{k \notin K_{ij}} \alpha_{ijk} + \sum_{k \in K_{ij}} (\alpha_{ijk} + n_{ijk}))}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} \geq \Gamma(2 + \sum_{k \notin K_{ij}} \alpha_{ijk}),$$

obtained by renaming $x = \sum_{k \notin K_{ij}} \alpha_{ijk}$ and $a_k = \alpha_{ijk} + n_{ijk}$, as we know that $\sum_{k \in K_{ij}} (\alpha_{ijk} + n_{ijk}) \geq n_{ij} \geq 2$ and each $a_k \geq 1$. Finally,

$$g((n_{ijk})_{\forall k}, (\alpha_{ijk})_{\forall k}) = -\log \frac{\Gamma(\alpha_{ij} + n_{ij})}{\prod_{k \in K_{ij}} \Gamma(\alpha_{ijk} + n_{ijk})} \leq -\log \Gamma(2 + \sum_{k \notin K_{ij}} \alpha_{ijk}) \leq 0,$$

because $\Gamma(2 + \sum_{k \notin K_{ij}} \alpha_{ijk}) \geq 1$. ■

Lemma 7 *Given a BD score and two parent sets Π_i^0 and Π_i for a node X_i such that $\Pi_i^0 \subset \Pi_i$, if*

$$s_i(\Pi_i^0) > \sum_{\substack{j\in J_i^{\Pi_i}:\\ |K_{ij}^{\Pi_i}|\geq 2}} f(K_{ij}^{\Pi_i}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) + \sum_{\substack{j\in J_i^{\Pi_i}:\\ |K_{ij}^{\Pi_i}|=1}} \log \frac{\alpha_{ijk'}^{\Pi_i}}{\alpha_{ij}^{\Pi_i}},$$

then Π_i is not the optimal parent set for X_i.

Proof Using the results of Lemmas 5 and 6,

$$\begin{aligned}
s_i(\Pi_i) &= \sum_{j\in J_i} \Big(f(K_{ij}^{\Pi_i}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) + g((n_{ijk}^{\Pi_i})_{\forall k}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) \Big) \\
&\leq \sum_{j\in J_i:\ |K_{ij}^{\Pi_i}|\geq 2} \Big(f(K_{ij}^{\Pi_i}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) + g((n_{ijk}^{\Pi_i})_{\forall k}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) \Big) + \\
&\quad + \sum_{j\in J_i^{\Pi_i}:\ |K_{ij}^{\Pi_i}|=1} \Big(-\log \alpha_{ij}^{\Pi_i} + \log \alpha_{ijk'}^{\Pi_i} \Big) \\
&\leq \sum_{j\in J_i^{\Pi_i}: |K_{ij}^{\Pi_i}|\geq 2} f(K_{ij}^{\Pi_i}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) + \sum_{j\in J_i^{\Pi_i}: |K_{ij}^{\Pi_i}|=1} \log \frac{\alpha_{ijk'}^{\Pi_i}}{\alpha_{ij}^{\Pi_i}},
\end{aligned}$$

which by the assumption of this lemma, is less than $s_i(\Pi_i^0)$. Thus, we conclude that the parent set Π_i^0 has better score than Π_i, and the desired result follows from Lemma 1. ■

Lemma 8 *Given the BDeu score, $(\alpha_{ijk})_{\forall ijk} > 0$, and integers $(n_{ijk})_{\forall ijk} \geq 0$ such that $\alpha_{ij} \leq 0.8349$ and $|K_{ij}| \geq 2$ for a given j, then $f(K_{ij}, (\alpha_{ijk})_{\forall k}) \leq -|K_{ij}| \cdot \log r_i$.*

Proof Using $\alpha_{ijk} \leq \alpha_{ij} \leq 0.8349$ (for all k), we have

$$\begin{aligned}
f(K_{ij}, (\alpha_{ijk})_{\forall k}) &= \log\Gamma(\alpha_{ij}) - |K_{ij}| \log\Gamma(\frac{\alpha_{ij}}{r_i}) \\
&= \log\Gamma(\alpha_{ij}) - |K_{ij}| \log\Gamma(\frac{\alpha_{ij}}{r_i} + 1) + |K_{ij}| \log \frac{\alpha_{ij}}{r_i} \\
&= \log\Gamma(\alpha_{ij}) - |K_{ij}| \log \frac{\Gamma(\frac{\alpha_{ij}}{r_i} + 1)}{\alpha_{ij}} - |K_{ij}| \log r_i \\
&= |K_{ij}| \log \frac{\Gamma(\alpha_{ij})^{1/|K_{ij}|}\alpha_{ij}}{\Gamma(\frac{\alpha_{ij}}{r_i} + 1)} - |K_{ij}| \log r_i.
\end{aligned}$$

Now, $\Gamma(\alpha_{ij})^{1/|K_{ij}|}\alpha_{ij} \leq \Gamma(\frac{\alpha_{ij}}{r_i} + 1)$, because $r_i \geq 2$, $|K_{ij}| \geq 2$ and $\alpha_{ij} \leq 0.8349$ (this number can be computed by numerically solving the inequality for $r_i = |K_{ij}| = 2$). We point out that 0.8349 is a bound for α_{ij} that ensures this last inequality to hold when $r_i = |K_{ij}| = 2$, which is the worst-case scenario (greater values of r_i and $|K_{ij}|$ make the left-hand side decrease and the right-hand side increase). Because r_i of each node is known, tighter bounds might be possible according to the node. ■

Theorem 9 *Given the BDeu score and two parent sets Π_i^0 and Π_i for a node X_i such that $\Pi_i^0 \subset \Pi_i$ and $\alpha_{ij}^{\Pi_i} \leq 0.8349$ for every j, if $s_i(\Pi_i^0) > -|K_i^{\Pi_i}| \log r_i$ then neither Π_i nor any superset $\Pi_i' \supset \Pi_i$ are optimal parent sets for X_i.*

Proof We have that

$$s_i(\Pi_i^0) > -|K_i^{\Pi_i}| \log r_i = \sum_{j \in J_i^{\Pi_i}:\ |K_{ij}^{\Pi_i}| \geq 2} -|K_{ij}^{\Pi_i}| \log r_i + \sum_{j \in J_i^{\Pi_i}:\ |K_{ij}^{\Pi_i}| = 1} -\log r_i,$$

which by Lemma 8 is greater than or equal to

$$\sum_{j \in J_i^{\Pi_i}:\ |K_{ij}^{\Pi_i}| \geq 2} f(K_{ij}^{\Pi_i}, (\alpha_{ijk}^{\Pi_i})_{\forall k}) + \sum_{j \in J_i^{\Pi_i}:\ |K_{ij}^{\Pi_i}| = 1} -\log r_i.$$

Now, Lemma 7 suffices to show that Π_i is not a optimal parent set, because $-\log r_i = \log \frac{\alpha_{ijk}^{\Pi_i}}{\alpha_{ij}^{\Pi_i}}$ for any k. To show the result for any superset $\Pi_i' \supset \Pi_i$, we just have to note that $|K_i^{\Pi_i'}| \geq |K_i^{\Pi_i}|$ (because the overall number of non-zero counts can only increase when we include more parents), and $\alpha_{ij'}^{\Pi_i'}$ (for all j') are all less than 0.8349 (because the αs can only decrease when more parents are included), thus we can apply the very same reasoning to all supersets. ■

Theorem 9 provides a bound to discard parent sets without even inspecting them because of the non-increasing monotonicity of the employed bounding function when we increase the number of parents. As done for the BIC and AIC criteria, the idea is to check the validity of Theorem 9 every time the score of a parent set Π_i of X_i is about to be computed by taking the best score of any subset and testing it against the theorem (of course using only subsets that satisfy the structural constraints). Whenever possible, we discard Π_i and do not even look into all its supersets. Note that the assertion $\alpha_{ij} \leq 0.8349$ required by the theorem is not too restrictive, because as parent sets grow, as ESS is divided by larger numbers (it is an exponential decrease of the αs). Hence, the values α_{ij} become quickly below such a threshold. Furthermore, Π_i is also checked against Lemma 1 (although it does not help with the supersets). As we see later in the experiments, the practical size of the cache after the application of the properties is small even for considerably large networks, and both Lemma 1 and Theorem 9 help reducing the cache size, while Theorem 9 also help to reduce computations. Finally, we point out that Singh and Moore (2005) have already worked on bounds to reduce the number of parent sets that need to be inspected, but Theorem 9 provides a much tighter bound than their previous result, where the cut happens only after all $|K_{ij}^{\Pi_i}|$ go below two (or using their terminology, when *configurations are pure*).

5. Constrained B&B Algorithm

In this section we describe the branch-and-bound (B&B) algorithm used to find the best structure of the Bayesian network and comment on its complexity and correctness. The algorithm uses a B&B search where each case to be solved is a relaxation of a DAG, that is, the cases may contain cycles. At each step, a graph is picked up from a priority queue, and it is verified if it is a DAG. In such case, it is a feasible structure for the network and we compare its score against the best score so

far (which is updated if needed). Otherwise, there must be a directed cycle in the graph, which is then broken into subcases by forcing some arcs to be absent/present. Each subcase is put in the queue to be processed (these subcases cover all possible subgraphs related to the original case, that is, they cover all possible ways to break the cycle). The procedure stops when the queue is empty. Note that every time we break a cycle, the subcases that are created are independent, that is, their sets of graphs are disjoint. We obtain this fact by properly breaking the cycles to avoid overlapping among subcases (more details below). This is the same idea as in the inclusion-exclusion principle of combinatorics employed over the set of arcs that formed the cycle and ensures that we never process the same graph twice, and also ensures that all subgraphs are covered.

The initialization of the algorithm is as follows:

- $C : (X_i, \Pi_i) \rightarrow \mathcal{R}$ is the cache with the scores for all the variables and their possible parent configurations. This is constructed using a queue and analyzing parent sets according to the properties of Section 4, which saves (in practice) a large amount of space and time. All the structural constraints are considered in this construction so that only valid parent sets are stored.

- $\mathcal{G}$ is the graph created by taking the best parent configuration for each node without checking for acyclicity (so it is not necessarily a DAG), and s is the score of $\mathcal{G}$. This graph is used as an upper bound for the best possible graph, as it is clearly obtained from a relaxation of the problem (the relaxation comes from allowing cycles).

- $\mathcal{H}$ is an initially empty matrix containing, for each possible arc between nodes, a mark stating that the arc must be present, or is prohibited, or is free (may be present or not). This matrix controls the search of the B&B procedure. Each branch of the search has a $\mathcal{H}$ that specifies the graphs that still must be searched within that branch.

- Q is a priority queue of triples $(\mathcal{G}, \mathcal{H}, s)$, ordered by s (initially it contains a single triple with $\mathcal{G}$, $\mathcal{H}$ and s as mentioned. The order is such that the top of the queue contains always the triple of greatest s, while the bottom has the triple of smallest s.

- $(\mathcal{G}_{best}, s_{best})$ keeps at any moment the best DAG and score found so far. The value of s_{best} could be set to $-\infty$, but this best solution can also be initialized using any inner approximation method. For instance, we use a procedure that guesses an ordering for the variable, then computes the global best solution for that ordering, and finally runs a hill climbing over the resulting structure. All these procedures are very fast (given the small size of the pre-computed cache that we obtain in the previous steps). A good initial solution may significantly reduce the search of the B&B procedure, because it may give a lower bound closer to the upper bound defined by the relaxation $(\mathcal{G}, \mathcal{H}, s)$.

- *iter*, initialized with zero, keeps track of the iteration number. *bottom* is a user parameter that controls how frequent elements will be picked from the bottom of the queue instead of the usual removal from the top. For example, a value of 1 means to pick always from the bottom, a value of 2 alternates elements from the top and the bottom evenly, and a large value makes the algorithm picks always from the top.

The main loop of the B&B search is as follows:

- While Q is not empty, do

  1. Increment *iter*. If $\frac{iter}{bottom}$ is not an integer, then remove the top of Q and put into $(\mathcal{G}_{cur}, \mathcal{H}_{cur}, s_{cur})$. Otherwise remove the bottom of Q into $(\mathcal{G}_{cur}, \mathcal{H}_{cur}, s_{cur})$. If $s_{cur} \leq s_{best}$ (worse than an already known solution), then discard the current element and start the loop again.
  2. If $\mathcal{G}_{cur}$ is a DAG, then update $(\mathcal{G}_{best}, s_{best})$ with $(\mathcal{G}_{cur}, s_{cur})$, discard the current element and start the loop again (if $\mathcal{G}_{cur}$ came from the top of Q, then the algorithm stops—no other graph in the queue can be better than $\mathcal{G}_{cur}$).
  3. Take a cycle of $\mathcal{G}_{cur}$ (one must exist, otherwise we would have not reached this step), namely $v = (X_{a_1} \rightarrow X_{a_2} \rightarrow \ldots \rightarrow X_{a_{q+1}})$, with $a_1 = a_{q+1}$.
  4. For $y = 1, \ldots, q$, do
     (a) Mark on $\mathcal{H}_{cur}$ that the arc $X_{a_y} \rightarrow X_{a_{y+1}}$ is prohibited. This implies that the branch we are going to create will not have this cycle again.
     (b) Recompute $(\mathcal{G}, s)$ from $(\mathcal{G}_{cur}, s_{cur})$ such that the new parent set of $X_{a_{y+1}}$ in $\mathcal{G}$ complies with this new $\mathcal{H}_{cur}$. This is done by searching in the cache $C(X_{a_{y+1}}, \Pi_{a_{y+1}})$ for the best parent set. If there is a parent set in the cache that satisfies $\mathcal{H}_{cur}$, then
        – Include the triple $(\mathcal{G}, \mathcal{H}_{cur}, s)$ into Q.[5]
     (c) Mark on $\mathcal{H}_{cur}$ that the arc $X_{a_y} \rightarrow X_{a_{y+1}}$ must be present and that the sibling arc $X_{a_{y+1}} \rightarrow X_{a_y}$ is prohibited, and continue the loop of step 4. (*Step 4c forces the branches that we create to be disjoint among each other.*)

There are two considerations to show the correctness of the method. First, we need to guarantee that all the search space is considered, even though we do not explicitly search through all of it. Second, we must ensure that the same part of the search space is not processed more than once, so we do not lose time and know that the algorithm will finish with a best global graph. The search is conducted over all possible graphs (not necessarily DAGs). The queue Q contains the subspaces (of all possible graphs) to be analyzed. A triple $(\mathcal{G}, \mathcal{H}, s)$ indicates, through $\mathcal{H}$, which is this subspace. $\mathcal{H}$ is a matrix containing an indicator for each possible arc. It says if an arc is allowed (meaning it might or might not be present), prohibited (it cannot be present), or demanded (it must be present) in the current subspace of graphs. Thus, $\mathcal{H}$ completely defines the subspaces. $\mathcal{G}$ and s are respectively the best graph inside $\mathcal{H}$ (note that $\mathcal{G}$ might have cycles) and its score value (which is an upper bound for the best DAG in this subspace).

In the initialization step, Q begins with a triple where $\mathcal{H}$ indicates that every arc is allowed,[6] so all possible graphs are within the subspace of the initial $\mathcal{H}$. In this moment, the main loop starts and the only element of Q is put into $(\mathcal{G}_{cur}, \mathcal{H}_{cur}, s_{cur})$ and s_{cur} is compared against the best known score. Note that as $\mathcal{G}_{cur}$ is the graph with the greatest score that respects $\mathcal{H}_{cur}$, any other graph within the subspace defined by $\mathcal{H}_{cur}$ will have worse score. Therefore, if s_{cur} is less than the best known score, all this branch represented by $\mathcal{H}_{cur}$ may be discarded (this is the *bound* step). Certainly no graph within that subspace will be worth checking, because their scores are less than s_{cur}.

5. One may check the acyclicity of the graph before including the triple in the queue. We analyze this possibility later on.
6. In fact, the implementation may set $\mathcal{H}$ with possible known restrictions of arcs, that is, those that are known to be demanded or prohibited by structural constraints may be included in the initial $\mathcal{H}$.

If $\mathcal{G}_{cur}$ has score greater than s_{best}, then the graph $\mathcal{G}_{cur}$ is checked for cycles, as it may or may not be acyclic (all we know is that $\mathcal{G}_{cur}$ is a relaxed solution within the subspace $\mathcal{H}_{cur}$). If it is acyclic, then $\mathcal{G}_{cur}$ is the best graph so far. Moreover, if the acyclic $\mathcal{G}_{cur}$ was extracted from the top of Q, then the algorithm may stop, as all the other elements in the queue have lower score (this is guaranteed by the priority of the queue). Otherwise we restart the loop, as we cannot find a better graph within this subspace (the acyclic $\mathcal{G}_{cur}$ is already the best one by definition). On the other hand, if $\mathcal{G}_{cur}$ is cyclic, then we need to divide the space $\mathcal{H}_{cur}$ into smaller subcases with the aim of removing the cycles of $\mathcal{G}_{cur}$ (this is the *branch* step). Two characteristics must be kept by the branch step: (i) $\mathcal{H}_{cur}$ must be fully represented in the subcases (so we do not miss any graph), and (ii) the subcases must be disjoint (so we do not process the same graph more than once). A possible way to achieve these two requirements is as follows: let the cycle $v = (X_{a_1} \to X_{a_2} \to \ldots \to X_{a_{q+1}})$ be the one detected in $\mathcal{G}_{cur}$. We create q subcases such that

- The first subcase does not contain $X_{a_1} \to X_{a_2}$ (but may contain the other arcs of that cycle, that is, we do not prohibit the others).
- The second case certainly contains $X_{a_1} \to X_{a_2}$, but $X_{a_2} \to X_{a_3}$ is prohibited (so they are disjoint because of the difference in the presence of the first arc).
- (And so on such that) The y-th case certainly contains $X_{a_{y'}} \to X_{a_{y'+1}}$ for all $y' < y$ and prohibits $X_{a_y} \to X_{a_{y+1}}$. This is done until the last element of the cycle.

This is the same idea as the inclusion-exclusion principle, but applied here to the arcs of the cycle. It ensures that we never process the same graph twice, and also that we cover all the graphs, as by the union of the mentioned sets we obtain the original $\mathcal{H}$. Because of that, the algorithm runs at most $\prod_i |C(X_i)|$ steps, where $|C(X_i)|$ is the size of the cache for X_i (there are not more ways to combine parent sets than that number). In practice, we expect the *bound* step to be effective in dropping parts of the search space in order to reduce the total time cost.

The B&B algorithm as described alternately picks elements from the top and from the bottom of the queue (the percentage of elements from the bottom is controlled by the user parameter *bottom*). In terms of covering all search space, we have to ensure that all elements of the queue are processed, no matter the order we pick them, and that is enough to the correctness of the algorithm. However, there is an important difference between elements from the top and the bottom: top elements improve the upper bound for the global score, because we know that the global score is less than or equal to the highest score in the queue. Still, the elements from the top cannot improve the lower bound, as lower bounds are made of valid DAGs, and the first found DAG from the top is already the global optimal solution (by the priority of the queue). In order to update also the lower bound, elements from the bottom can be used, as they have low score with (usually) small subspaces, making easier to find valid DAGs. In fact, we know that an element from the bottom, if not a DAG, will generate new elements of the queue whose subspaces have upper bound score less than that of the originating elements, which certainly put them again in the bottom of the queue. This means that processing elements from the bottom is similar to perform a depth-first search, which is likely to find valid DAGs. Hence, we guarantee to have both lower and upper bounds converging to the optimal solution.

In the experiments of Section 6, we have chosen the parameter *bottom* such that one in three iterations picks an element from the bottom of the queue. This choice has not been tuned and has been taken with the aim of increasing the chance of finding valid DAGs. Note that every element

from the top will certainly decrease the upper bound, while the elements from the bottom may or may not increase the lower bound. There is no obvious choice here: if we use fewer elements from the bottom, then we improve the upper bound faster, but we possibly have a worse lower bound, which implies in less chance of *bounding* regions of the search space (which would help to improve the upper bound in a faster way as well); on the other hand, if we use many elements from the bottom, then we increase the chance (even if there is no guarantee) of improving the lower bound, but we spend less time improving the upper bound, which ultimately has to be tightened until it meets the lower bound. In other words, if the current best solution is already very good (in the sense of being optimal or almost optimal—note that we do not know it when the method is running), then it is useless to pick elements from the bottom. Therefore, a possible (heuristic) approach is to adaptively select the percentage of elements to pick from the bottom: in the very beginning of the algorithm, more elements are picked from the bottom. As time passes, as the upper bound gets closer to the best current solution (it also becomes less likely to find better solutions because the chance that the current solution is already good gets higher with time), so the percentage of elements picked from the bottom should keep reducing until it reaches zero (or almost zero). Currently we have not implemented any strategy to modify the percentage of elements that are picked from top and bottom of the queue.

Two other ideas are worth mentioning regarding the B&B algorithm: (i) if we periodically perform local searches within subspaces using distinct starting points, the lower bound can be improved (still this has its own computational cost, so it must be selectively done); (ii) if we do check for acyclicity in the step 4b before inserting the triple into the queue, then it is possible to update the current best solution earlier, and the algorithm still works. In this case, step 2 is unnecessary because DAGs will never be inserted into the queue (given that we check if the initial graph is not already a DAG before starting the main loop). Still, we need to find the cycle to be used in step 3, so to save computations we need to spend memory to store the cycle (previously found in step 4b) together with the triples of the queue. Hence, this idea trades some computational time (or memory usage) by a speed-up in finding some DAGs to improve the lower bound. Note that, in most cases, the graph that is checked in step 4b will not be a DAG anyway. While this modification benefits the improvement of the lower bound by spending some additional computation/memory, some preliminary experiments have not shown any significant gain. However, this is still to be better analyzed, as it may vary depending on implementation details.

The B&B can be stopped at any time and the current best solution as well as an upper bound for the global best score are available. This stopping criterion might be based on number of steps, time and/or memory consumption, percentage of error (difference between upper and lower bounds). This is an important property of this method. For example, if we are just looking for an improving solution, we may include in the loop an *if* to check if the current best solution is already better than some threshold, which would save computational time. Still, if we run it until the end, we are ensured to have a global optimum solution.

The algorithm can also be easily parallelized. We can split the content of the priority queue into many different tasks. No shared memory needs to exist among tasks if each one has its own version of the cache. The only data structure that needs consideration is the queue, which from time to time must be balanced between tasks. With a message-passing idea that avoids using *locks*, the gain of parallelization is linear in the number of tasks.

Some particular cases of the algorithm are worth mentioning. If we fix an ordering for the variables such that all the arcs must link a node towards another non-precedent in the ordering (this

is a common idea in many approximate methods), the proposed algorithm does not perform any branch, as the ordering implies acyclicity, and so the initial solution is already the best (only for that ordering—recall that the number of possible orderings is exponential in n). The performance would be proportional to the time to create the cache. Another important case is when one limits the maximum number of parents of a node. This is relevant for hard problems with many variables, as it would imply in a bound on the cache size.

	ESS	adult	breast	car	letter	lung	mush	nurse	wdbc	zoo
	0.1	6.2	0.0	0.1	3.7	1699.6	7.5	0.9	221.2	0.4
Memory	1	6.2	0.0	0.1	3.7	1150.1	5.9	0.8	204.6	0.4
(in MB)	10	6.3	0.0	0.1	3.8	812.3	5.4	0.7	206.2	0.3
	BIC	1.8	0.0	0.0	2.3	0.3	0.5	0.4	5.3	0.1
	0.1	89.3	0.0	0.0	429.4	2056	357.9	0.7	2891	1.7
Time	1	91.6	0.0	0.0	440.4	1398	278.7	0.7	2692	1.7
(in sec.)	10	91.6	0.0	0.0	438.1	1098	268.9	0.7	2763	1.7
	BIC	67.4	0.0	0.1	859.6	1.3	72.1	1.4	351	0.3
	0.1	$2^{17.4}$	$2^{10.5}$	$2^{8.8}$	$2^{20.1}$	$2^{30.8}$	$2^{24.0}$	$2^{11.2}$	$2^{27.9}$	$2^{19.8}$
Number	1	$2^{17.4}$	$2^{10.5}$	$2^{8.8}$	$2^{20.1}$	$2^{30.2}$	$2^{23.6}$	$2^{11.2}$	$2^{27.8}$	$2^{19.7}$
of Steps	10	$2^{17.4}$	$2^{10.4}$	$2^{8.8}$	$2^{20.1}$	$2^{29.8}$	$2^{23.5}$	$2^{11.2}$	$2^{27.9}$	$2^{19.6}$
	BIC	$2^{14.8}$	$2^{7.3}$	$2^{8.4}$	$2^{19.0}$	$2^{15.4}$	$2^{17.1}$	$2^{10.9}$	$2^{20.7}$	$2^{13.1}$
Worst-case		$2^{17.9}$	$2^{12.3}$	$2^{8.8}$	$2^{20.1}$	$2^{31.1}$	$2^{26.5}$	$2^{11.2}$	$2^{28.4}$	$2^{20.1}$

Table 1: Memory, time and number of steps (local score evaluations) used to build the cache. Results for BIC and BDeu with ESS varying from 0.1 to 10 are presented.

6. Experiments

We perform experiments to show the benefits of the reduced cache and search space. Later we show some examples of the use of constraints.[7] First, we use data sets available at the UCI repository (Asuncion and Newman, 2007). Lines with missing data are removed and continuous variables are discretized over the mean into binary variables. The data sets are: *adult* (15 variables and 30162 instances), *breast* (10 variables and 683 instances), *car* (7 variables and 1728 instances) *letter* (17 variables and 20000 instances), *lung* (57 variables and 27 instances), mushroom (23 variables and 1868 instances, denoted by *mush*), nursery (9 variables and 12960 instances, denoted by *nurse*), Wisconsin Diagnostic Breast Cancer (31 variables and 569 instances, denoted by *wdbc*), *zoo* (17 variables and 101 instances). The number of categories per variables varies from 2 to dozens in some cases (we refer to UCI for further details).

Table 1 presents the used memory in MB (first block), the time in seconds (second block) and number of steps in local score evaluations (third block) for the cache construction, using the properties of Section 4. Each column presents the results for a distinct data set. In different lines we show results for BDeu with ESS equals to 0.1, 1, 10, and for BIC. The line *worst-case* presents the number of steps to build the cache without using Theorems 4 (for BIC/AIC) and 9 (for BDeu), which are the theorems that allow the algorithm to avoid computing every subset of parents. As we see through the log-scale in which they are presented, the reduction in number of steps has not been

7. The software is available online in the web address `http://www.ecse.rpi.edu/~cvrl/structlearning.html`.

exponential, but still saves a good amount of computations (roughly half of the work). In the case of the BIC score, the reduction is more significant. In terms of memory, the usage clearly increases with the number of variables in the network (lung has 57 and wdbc has 31 variables).

	ESS	adult	breast	car	letter	lung	mush	nurse	wdbc	zoo
Max.	0.1	2.1(4)	1.0(1)	0.7(1)	4.5(5)	0.1(2)	4.1(5)	1.2(3)	1.3(2)	1.4(3)
Number	1	2.4(4)	1.0(1)	1.0(2)	5.2(6)	0.4(2)	4.4(7)	1.7(3)	1.7(3)	1.9(4)
of Parents	10	3.3(5)	1.0(1)	1.9(2)	5.9(6)	3.0(4)	4.8(8)	2.1(3)	3.1(4)	3.4(4)
	BIC	2.8(5)	1.0(1)	1.3(2)	6.3(7)	2.1(3)	4.1(4)	1.8(3)	2.7(3)	2.8(3)
Worst-case		14.0	9.0	6.0	16.0	6.0^{*}	22.0	8.0	8.0^{*}	16.0
Final Size	0.1	$2^{4.2}$	$2^{1.5}$	$2^{1.1}$	$2^{8.2}$	$2^{0.2}$	$2^{8.5}$	$2^{1.9}$	$2^{3.6}$	$2^{3.3}$
of the	1	$2^{4.8}$	$2^{1.9}$	$2^{1.6}$	$2^{9.0}$	$2^{0.8}$	$2^{8.9}$	$2^{2.4}$	$2^{4.9}$	$2^{4.4}$
Cache	10	$2^{6.3}$	$2^{3.3}$	$2^{3.0}$	$2^{10.5}$	$2^{10.7}$	$2^{9.8}$	$2^{3.5}$	$2^{12.1}$	$2^{8.9}$
	BIC	$2^{9.3}$	$2^{4.7}$	$2^{4.5}$	$2^{15.3}$	$2^{11.5}$	$2^{13.0}$	$2^{5.6}$	$2^{12.9}$	$2^{10.9}$
Worst-case		$2^{17.9}$	$2^{12.3}$	$2^{8.8}$	$2^{20.1}$	$2^{31.1*}$	$2^{26.5}$	$2^{11.2}$	$2^{28.4*}$	$2^{20.1}$
Implied	0.1	$2^{54.1}$	$2^{13.3}$	$2^{6.3}$	$2^{129.0}$	$2^{8.2}$	$2^{175.7}$	$2^{11.6}$	$2^{90.3}$	$2^{39.3}$
Search	1	$2^{62.1}$	$2^{17.1}$	$2^{8.3}$	$2^{144.8}$	$2^{33.1}$	$2^{186.0}$	$2^{15.4}$	$2^{132.7}$	$2^{60.3}$
Space	10	$2^{91.6}$	$2^{33.2}$	$2^{20.6}$	$2^{176.1}$	$2^{612.0}$	$2^{221.8}$	$2^{27.3}$	$2^{375.1}$	$2^{150.7}$
(approx.)	BIC	2^{71}	2^{23}	2^{10}	2^{188}	2^{330}	2^{180}	2^{17}	2^{216}	2^{111}
Worst-case		2^{210}	2^{90}	2^{42}	2^{272}	2^{1441*}	2^{506}	2^{72}	2^{727*}	2^{272}

Table 2: Final cache characteristics: maximum number of parents (average by node; between parenthesis is presented the actual maximum number), actual cache size, and (approximate) search space implied by the cache. Worst-cases are presented for comparison (those marked with a star are computed using the constraint on the number of parents that was applied to *lung* and *wdbc*). Results of BIC and BDeu with ESS from 0.1 to 10 are presented.

The benefits of the application of these results imply in performance gain for many algorithms in the literature to learn Bayesian network structures, as long as they only need to work over the (already precomputed) small cache. In Table 2 we present the final cache characteristics, where we find the most attractive results, for instance, the small cache sizes when compared to the worst case. The first block contains the maximum number of parents per node (averaged over the nodes, and the actual maximum between parenthesis). The worst-case is the total number of nodes in the data set minus one, apart from *lung* (where we have set a limit of at most six parents) and *wdbc* (with at most eight parents). The second block shows the cache size for each data set and distinct values of ESS. We also show the results of the BIC score and the worst-case values for comparison. We see that the actual cache size is smaller (in orders of magnitude) than the worst-case situation. It is also possible to analyze the search space reduction implied by these results by looking the implications to the search space of structure learning. We must point out that by search space we mean all the possible combinations of parent sets for all the nodes. Eventually some of these combinations are not DAGs, but are still being counted. However, there are two considerations: (i) the precise counting problem is harder to solve (in order to give the exact search space size), and (ii) many structure learning algorithms run over more than only DAGs, because they need to look at the graphs (and thus combinations of parents) to decide if they are acyclic or not. In these cases, the actual search space is not simply the set of possible DAGs, even though the final solution will be a DAG. Still, some algorithms might do a better job by using other ideas of searching for

the best structure instead of looking to possible DAGs, which might imply in a smaller worst-case complexity (for instance, the dynamic programming method runs over subsets of variables, which are in number 2^n).

		B&B			DP		OS		HC	
	network	Score	gap	time	score	time	score	time	score	time
BIC	adult	-286902.8	5.5%	150.3	0.0%	0.77	0.1%	0.17	0.5%	0.30
	breast	-8254.8	0.0%	0.01	0.0%	0.01	0.0%	0.01	0.0%	0.00
	car	-13100.5	0.0%	0.01	0.0%	0.01	0.0%	0.01	0.2%	0.00
	letter	-173716.2	8.1%	574.1	-0.6%	22.8	1.0%	0.75	3.7%	0.30
	lung	-1146.9	2.5%	907.1	*Fail*	*Fail*	1.0%	0.13	0.7%	0.05
	mushroom	-12834.9	15.3%	239.8	*Fail*	*Fail*	1.0%	0.12	4.8%	0.05
	nursery	-126283.2	0.0%	0.04	0.0%	0.04	0.0%	0.04	0.03%	0.06
	wdbc	-3053.1	13.6%	333.5	*Fail*	*Fail*	0.8%	0.13	0.9%	0.02
	zoo	-773.4	0.0%	5.2	0.0%	3.5	1.0%	0.03	0.6%	0.00
ESS=0.1	adult	-288591.2	0.0%	92.1	0.0%	0.75	0.1%	0.21	0.3%	0.32
	breast	-8635.1	0.0%	0.02	0.0%	0.01	0.0%	0.01	0.0%	0.00
	car	-13295.0	0.0%	0.01	0.0%	0.00	0.0%	0.00	0.1%	0.01
	letter	-181941.5	5.7%	375.75	-0.1%	7.6	0.1%	0.27	2.1%	0.27
	lung	-1731.9	0.0%	0.22	*Fail*	*Fail*	0.0%	0.11	0.0%	0.05
	mushroom	-12564.2	14.7%	382.4	*Fail*	*Fail*	0.2%	0.15	5.3%	0.05
	nursery	-126660.4	0.0%	0.06	0.0%	0.04	0.0%	0.04	0.1%	0.06
	wdbc	-3558.6	4.4%	494.1	*Fail*	*Fail*	1.4%	0.05	1.3%	0.01
	zoo	-1024.5	0.0%	0.09	0.0%	3.1	0.8%	0.01	1.0%	0.00
ESS=1	adult	-286695.2	4.5%	203.0	0.0%	0.76	0.1%	0.22	0.3%	0.34
	breast	-8254.3	0.0%	0.02	0.0%	0.01	0.0%	0.01	0.0%	0.00
	car	-13145.3	0.0%	0.01	0.0%	0.00	0.0%	0.00	0.05%	0.00
	letter	-178635.2	6.7%	520.2	-0.7%	9.9	0.0%	0.34	2.1%	0.27
	lung	-1249.7	0.0%	0.61	*Fail*	*Fail*	0.1%	0.12	0.1%	0.05
	mushroom	-12097.0	16.7%	381.5	*Fail*	*Fail*	0.2%	0.19	4.2%	0.05
	nursery	-126212.7	0.0%	0.06	0.0%	0.04	0.0%	0.04	0.1%	0.05
	wdbc	-3175.9	11.2%	471.1	*Fail*	*Fail*	0.7%	0.06	1.0%	0.02
	zoo	-794.1	0.0%	1.4	0.0%	3.4	1.1%	0.02	8.7%	0.00
ESS=10	adult	-285014.5	11.8%	213.8	-0.1%	0.88	0.04%	0.24	0.5%	0.33
	breast	-8130.2	0.0%	0.04	0.0%	0.01	0.0%	0.00	0.3%	0.00
	car	-13038.6	0.0%	0.03	0.0%	0.00	0.0%	0.00	0.03%	0.00
	letter	-174111.8	8.7%	1250	-0.4%	22.3	0.1%	0.84	1.8%	0.32
	lung	-957.2	11.7%	2118	*Fail*	*Fail*	3.3%	1.38	2.3%	0.1
	mushroom	-11924.0	22.7%	587.8	*Fail*	*Fail*	0.1%	0.43	2.4%	0.07
	nursery	-125846.5	0.0%	0.14	0.0%	0.04	0.0%	0.04	0.1%	0.06
	wdbc	-2986.2	22.2%	1938	*Fail*	*Fail*	0.6%	2.8	1.4%	0.23
	zoo	-697.2	13.2%	367.7	-0.3%	5.0	1.4%	0.1	0.9%	0.00

Table 3: Comparison of scores among B&B, DP, OS and HC. *Fail* means that it could not solve the problem within 10 million steps or because of memory limit (4GB). DP, OS and HC scores are in percentage w.r.t. the score of B&B (positive means worse than B&B and negative means better). Each entry with a 0.0% means that the result, in that instance, was exactly equal to the B&B result (in terms of the score). Times are given in seconds.

An expected but important point to emphasize is the correlation of the prior with the time and memory to build the cache. It would be expected that, as larger ESS (and thus the prior towards the uniform) as slower and more memory consuming is the method. That is because smoothing the different parent sets by the stronger prior makes harder to see large differences in scores, and consequently the properties that would reduce the cache size are less effective. However, this is not quite evident from the results, where the relation between ESS and time/memory is not clear. Yet it must be noted that the two largest data sets in terms of number of variables (*lung* and *wdbc*) were impossible to be processed without setting up other limits such as maximum number of parents or maximum number of free parameters in the node (we have not used any limit for the other data sets). We used an upper limit of six parents per node for *lung* and eight for *wdbc*. This situation deserves further study so as to clarify whether it is possible to run these computations on large data sets and large ESS. It might be necessary to find tighter bounds if at all possible, that is, stronger results than Theorem 9 to discard unnecessary score evaluations earlier in the computations. Nevertheless, the main goal of this present work is not to study the impact of ESS on learning, but to present properties that improve the performance of learning methods.

In Table 3, we show results of four distinct algorithms: the B&B described in Section 5, the dynamic programming (DP) idea of Silander and Myllymaki (2006), the hill-climbing (HC) method starting with an empty structure, and an algorithm that picks variable orderings randomly and then find the best structure satisfying that ordering, that is, DAGs where arcs respect the ordering of the variables (there is no arc connecting a node to its predecessors in the ordering). This algorithm (named OS) is similar to K2 algorithm with random orderings, but it is always better because a global optimum is found for each ordering (we use one million of orderings). Note that OS performs better than HC in almost all test cases. We have chosen to analyze the BIC scores (given that the properties have provided greater reduction in the search space in this case) and BDeu with ESS equals to 0.1, 1 and 10. It is clear from the results of ESS equals to 10 that the B&B procedure struggles with very large search spaces, and the same might happen for even larger ESS.

The scores obtained by each algorithm (in percentage against the value obtained by B&B) and the corresponding time are shown in Table 3 (excluding the cache construction). A limit of ten million steps is given to each method (steps here are considered as the number of queries to the cache). It is also presented the reduced space where B&B performs its search, as well as the maximum gap of the solution. This gap is obtained by the relaxed version of the problem. We can guarantee that the global optimal solution is within this gap (even though the solution found by the B&B may already be the best, as it happens, for example, in the first line of the table). With the reduced cache presented here, finding the best structure for a given ordering is very fast, so it is possible to run OS over millions of orderings in a short period of time. Some additional comments are worth. DP could not solve *wdbc* or *lung* even without the limit in number of steps, because it has exhausted 16GB of memory. Hence, we cannot expect to obtain answers in larger cases. However, it is clear that (in a worst case sense) the number of steps of DP is smaller than that of B&B, and this behavior can be seen in data sets with small number of variables. Nevertheless, B&B eventually bounds some regions without processing them, provides an upper bound at each iteration, and does not suffer from memory exhaustion as DP. It is true that B&B also uses memory increasingly if there are not good bounds, but this case can be tackled by (automatically) switching between the described B&B and a full depth-first search.[8] This makes the method applicable even to very large settings. When

8. Our implementation is able to stop the B&B and to switch to a full depth-first search, but this behavior was not necessary in the experiments because the memory requirements were not too intense.

n is large (more than 35), DP will not finish in reasonable time, and hence will not provide any solution, while B&B still gives an approximation and a bound to the global optimum. About OS, if we sample even more orderings, then its results improve and the global optimum is found also for the *adult* data set. Still, OS provides no guarantee or estimation about how far is the global optimum (here we know that the optimum has been achieved because of the solution of the exact methods). It is worth noting that both DP and OS are also benefited by the smaller cache. Although we are discussing only four algorithms, performance gain from the application of the properties in other algorithms is expected as well.

network	time(s)	cache size	space
adult	0.26	114	2^{39}
car	0.01	14	$2^{6.2}$
letter	0.32	233	2^{61}
lung	0.26	136	2^{51}
mushroom	0.71	398	2^{88}
nursery	0.06	26	2^{12}
wdbc	361.64	361	2^{99}
zoo	8.4	1697	2^{111}

Table 4: B&B procedure learning TANs using BIC. Time (in seconds) to find the global optimum, cache size (number of stored scores) and (reduced) space for the search.

The last part of this section is dedicated to some test cases with constraints. Table 4 shows the results when we employ constraints to force the final network to be a Tree-augmented Naive Bayes. Here the class variable is isolated in the data set and constraints are included as described in Section 3. Note that the cache size, the search space and consequently the time to solve the problems have substantially decreased. Finally, Table 5 has results for random data sets with predefined number of nodes and instances using the BIC score. A randomly created Bayesian network with at most $3n$ arcs (where n is the number of nodes) is used to sample the data. Because of that, we are able to generate random structural constraints that are certainly valid for this *true* Bayesian network (approximately n constraints for each case). The table contains the total time to run the problem and the size of the cache, together with the results when using constraints. Note that the code was run in parallel with a number of tasks equals to n, otherwise an increase by a factor of n must be applied to the results in the table. Each line contains the mean and standard deviation of ten executions (using random generated networks) for time and cache size with and without constraints (using the same data sets in order to compare them). We can see that the gain is recurrent in all cases. The B&B method was able to find a global optimal solution in all but the cases with one hundred nodes, where it has achieved an approximate solution with error always less than 0.1% (this amounts to 40% of the test cases with 100 nodes). We point out that the other exact method we have analyzed based on dynamic programming cannot deal with such large networks because of both memory and time costs. There is an increase in computational time from 30 to 100 nodes, but even more from 100 to 500 instances (considering the data sets with 70 and 100 nodes). This happens because the properties that reduce the cache size and search space are much more effective under small-sized data sets. However, we are not considering the improvement in accuracy when using constraints, but just the computational gain. It is not trivial to measure the quality of a learned structure, because the target of the methods is the underlying probability distribution, and distinct structures may lead

to good results in fitting such distribution. For instance, comparing number of matching arcs has only meaning if one is interested in the structure by itself, and not in the fitness of the underlying distribution. This topic deserves attention, but it would bring us far from the focus of this study.

	unconstrained				constrained			
nodes(n)/	time(sec)		cache size		time(sec)		cache size	
instances	mean	std.dev.	mean	std.dev.	mean	std.dev.	mean	std.dev.
30/100	0.07	0.02	49.6	9.1	0.04	0.01	44.3	8.98
30/500	3.70	1.18	75.6	16.6	2.33	0.73	61.4	17.7
50/100	0.31	0.08	77.9	9.6	0.20	0.04	66.1	6.71
50/500	37.1	10.8	102.5	23.0	23.2	6.86	83.0	17.7
70/100	1.91	0.82	127.5	18.1	0.97	0.32	108.3	13.6
70/500	293.3	99.5	137.3	22.2	176.3	62.6	111.8	14.5
100/100	85.0	29.3	253.4	27.7	4.44	1.06	199.5	21.1
100/500	2205.6	534.4	204.6	32.1	1414.8	419.2	168.0	21.3

Table 5: Results on ten data sets per line generated from random networks. Both mean and standard deviation of time to solve (with an upper limit of 20 million steps) and size of the cache (in number of scores) are presented for the *normal* unconstrained case and for the constrained cases (over the same data sets).

7. Conclusions

This paper describes novel properties of decomposable score functions to learn Bayesian network structure from data. Such properties allow the construction of a cache with all possible local scores of nodes and their parents without large memory consumption, which can later be used by searching algorithms. For instance, memory consumption was a bottleneck for some algorithms in the literature, see, for example, Parviainen and Koivisto (2009). This implies in a considerable reduction of the search space of graphs without losing the global optimal structure, that is, it is ensured that the overall best graph remains in the reduced space. In fact the reduced memory and search space potentially benefits many structure learning methods in the literature, and we have conducted experiments with some of them.

An algorithm based on a branch-and-bound technique is described, which integrates structural constraints with data. The procedure guarantees global optimality with respect the score function. It is an anytime procedure in the sense that the error of the current solution is constantly reduced either by finding a better solution or by reducing the upper bound for the global optimum. If stopped early, the method provides the current solution and its maximum error. This can be useful if one wants to integrate it with an expectation-maximization (EM) method to treat incomplete data sets, and such characteristic is usually not present in other exact structure learning methods. In the EM method, the global structure learning procedure ensures that the maximization step is never trapped by a local solution, and the anytime property allows the use of a generalized EM to reduce considerably the computational cost.

Because of the properties and the characteristics of the B&B method, it is more efficient than dynamic programming state-of-the-art exact methods for large domains. We show through experiments with randomly generated data and public data sets that problems with up to 70 nodes can

be exactly processed in reasonable time, and problems with 100 nodes are handled within a small worst-case error. Dynamic programming methods are able to treat less than 35 variables. Described ideas may also help to improve other approximate methods and may have interesting practical applications. We show through experiments with public data sets that requirements of memory are small, as well as the resulting reduced search space. Of course we do not expect to exactly solve problems for considerably large networks, still the paper makes a relevant step towards solving larger instances. We can summarize the comparison with the dynamic programming idea as follows: if the problem has few variables, dynamic programming is probably the fastest method (the branch-and-bound method will also be reasonably fast); if the problem has medium size, the branch-and-bound method might solve it exactly (dynamic programming will mostly fail to answer); finally, if the problem is large, the branch-and-bound method will eventually give an approximation (and its worst-case error), while the standard dynamic programming idea will fail.

There is certainly much further to be done. One important question is whether the bounds of the theorems in Section 4 (more specifically Theorem 9) can be improved or not. We are actively working on this question. Furthermore, the experimental analysis can be extended to further clarify the understanding of the problem, for instance how the ESS affects the results. It is clear that, for considerably large domains, none of the exact methods are going to suffice by themselves. Besides developing ideas and algorithms for dealing with large domains, the comparison of structures and what define them to be good is an important topic. For example, accuracy of the generated networks can be evaluated with real data. On the other hand, it does not ensure that we are finding the true links of the underlying structure, but a somehow similar graph that produces a close joint distribution. For that, one could use generated data and compare the structures against the one data were generated from it. A study on how the properties may help fast approximate methods is also a desired goal.

Acknowledgments

This work is supported in part by the grant W911NF-06-1-0331 from the U.S. Army Research Office, and by the *Computational Life Sciences (CLS)* program phase II, canton Ticino, Switzerland.

References

H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974.

D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Univ. Press, 2006.

A. Asuncion and D.J. Newman. UCI machine learning repository, 2007. URL `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

R. Bouckaert. Properties of Bayesian belief network learning algorithms. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, UAI'94, pages 102–109, San Francisco, CA, 1994. Morgan Kaufmann.

W. Buntine. Theory refinement on Bayesian networks. In *Proceedings of the 8th Annual Conference on Uncertainty in Artificial Intelligence*, UAI'92, pages 52–60, San Francisco, CA, 1991. Morgan Kaufmann.

D. M. Chickering. Optimal structure identification with greedy search. *Journal of Machine Learning Research*, 3:507–554, 2002.

D. M. Chickering, C. Meek, and D. Heckerman. Large-sample learning of Bayesian networks is np-hard. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, UAI'03, pages 124–13, San Francisco, CA, 2003. Morgan Kaufmann.

G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

C. P. de Campos, Z. Zeng, and Q. Ji. Structure learning of Bayesian networks using constraints. In *Proceedings of the 26th International Conference on Machine Learning*, ICML'09, pages 113–120, Montreal, 2009. Omnipress.

D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: the combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

T. Jaakkola, D. Sontag, A. Globerson, and M. Meila. Learning Bayesian Network Structure using LP Relaxations. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, AISTATS'10, pages 358–365, 2010.

M. Koivisto. Advances in exact Bayesian structure discovery in Bayesian networks. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, UAI'06, pages 241–248. AUAI Press, 2006.

M. Koivisto and K. Sood. Exact Bayesian Structure Discovery in Bayesian Networks. *Journal of Machine Learning Research*, 5:549–573, 2004.

K. Kojima, E. Perrier, S. Imoto, and S. Miyano. Optimal search on clustered structural constraint for learning Bayesian network structure. *Journal of Machine Learning Research*, 11:285–310, 2010.

S. Ott and S. Miyano. Finding optimal gene networks using biological constraints. *Genome Informatics*, 14:124–133, 2003.

P. Parviainen and M. Koivisto. Exact structure discovery in Bayesian networks with less space. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, UAI'09, pages 436–443, Arlington, Virginia, United States, 2009. AUAI Press.

F. Pernkopf and J. Bilmes. Discriminative versus generative parameter and structure learning of Bayesian network classifiers. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML'05, pages 657–664, New York, NY, USA, 2005. ACM.

E. Perrier, S. Imoto, and S. Miyano. Finding optimal Bayesian network given a super-structure. *Journal of Machine Learning Research*, 9:2251–2286, 2008.

G. Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

T. Silander and P. Myllymaki. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, UAI'06, pages 445–452, Arlington, Virginia, 2006. AUAI Press.

A. P. Singh and A. W. Moore. Finding optimal Bayesian networks by dynamic programming. Technical report, Carnegie Mellon University, 2005. CMU-CALD-05-106.

P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction and Search*. Springer-Verlag, New York, 1993.

J. Suzuki. Learning Bayesian belief networks based on the minimum description length principle: An efficient algorithm using the B&B technique. In *Proceedings of the 13th International Conference on Machine Learning*, ICML'96, pages 462–470, 1996.

M. Teyssier and D. Koller. Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence*, UAI'05, pages 584–590, 2005.

I. Tsamardinos, L. E. Brown, and C. Aliferis. The max-min hill-climbing Bayesian network structure learning algorithm. *Machine Learning*, 65(1):31–78, 2006.