Computing the NML for Bayesian Forests via Matrices and Generating Polynomials

Tommi Mononen, Petri Myllymäki
Helsinki Institute for Information Technology, Finland

***Abstract*— The *Minimum Description Length* (MDL) is an information-theoretic principle that can be used for model selection and other statistical inference tasks. One way to implement this principle in practice is to compute the *Normalized Maximum Likelihood* (NML) distribution for a given parametric model class. Unfortunately this is a computationally infeasible task for many model classes of practical importance. In this paper we present a fast algorithm for computing the NML for the model class of Bayesian forests, which are graphical dependency models for multi-dimensional domains with the constraint that each node (variable) has at most one predecessor. The resulting algorithm has the time complexity of $\mathcal{O}(n^{2\mathcal{K}+\mathcal{L}-3})$, where n is the number of data vectors, and $\mathcal{K}$ and $\mathcal{L}$ are the maximal number of values (alphabet sizes) of different types of variables in the model.**

I. Introduction

Let us consider an i.i.d. sample of n vectors $\mathbf{x}^n = \mathbf{x}_1, \ldots, \mathbf{x}_n$, where each vector $\mathbf{x}_i$ consists of s discrete symbols taken from some alphabet (with each of the s components having its own alphabet). Given a parametric model $\mathcal{M}$, the normalized maximum likelihood (NML) distribution (or code) [7], [9] is defined by

$$P_{NML}(\mathbf{x}^n|\mathcal{M}) = \frac{P(\mathbf{x}^n|\hat{\theta}(\mathbf{x}^n, \mathcal{M}))}{\sum_{\mathbf{y}^n} P(\mathbf{y}^n|\hat{\theta}(\mathbf{y}^n, \mathcal{M}))}, \qquad (1)$$

where the numerator is the maximum likelihood for the observed data $\mathbf{x}^n$ within $\mathcal{M}$. The *normalizing term* in the denominator is the sum over maximum likelihoods of all possible data sets of size n, with respect to the model class. The NML offers a theoretically appealing, minmax optimal criterion for model selection and other statistical inference tasks, but is typically hard to compute.

Bayesian forests are tree-structured graphical Bayesian network models [6], [3]: a tree is a connected directed acyclic graph where each node has at most one parent, and a forest is a set of trees. In a Bayesian tree, each node corresponds to a random variable (a column in $\mathbf{x}^n$), and the joint probability distribution of a vector $\mathbf{v} = (v_1, \ldots, v_s)$ factorizes as

$$P(\mathbf{v}) = \prod_{i=1}^{s} P(v_i|v_{f(i)}), \qquad (2)$$

where $f(i)$ is the index of the parent node of node i. So each node has a local probability distribution conditioned on its parent node and the joint distribution is a product of these local distributions. Trees are independent of each other, so the joint probability of a forest is a product of joint probabilities of trees. In this paper we use multinomial local distributions, and hence the joint distribution is a product of multinomials.

For Bayesian forests, computing the numerator of (1) is trivial [10], while computing the denominator is computationally very demanding. In the following we represent a novel algorithm for computing the normalizing term of (1) in the case of Bayesian forests. The efficiency of the algorithm depends on the number of values of the variables (the sizes of the column alphabets).

II. Matrix representation

Computing the NML normalizing term for a Bayesian forest is quite complicated: the basic task is to take a sum over all possible data sets of size n and compute the maximum likelihood for each data. However, note that this is equivalent to summing over all possible sufficient statistics related to the model and computing each maximum likelihood multiplied by a factor which counts the number of data sets producing the same sufficient statistics. As we saw earlier, Bayesian multinomial forests have a nice property that the global distribution decomposes into local probability distributions. Now the main idea is to claim that we can use a similar decomposition for computing our NML normalizing term more efficiently. For accomplishing this, we define three different type of local *linear components*: a root node component, an inner node component and a leaf node component. We can represent these components as vectors and matrices. The normalizing term corresponding to the whole forest can be then computed using simple matrix operations between the precomputed components. As we compute over all possible local sufficient statistics in every component, we get the wanted result: the value of the normalizing term for an forest. For further investigations of the sufficient statistics related to this problem, see [10].

A. General algorithm

Let us first have a look at the two-step matrix algorithm on a general level without defining exactly the components. The algorithm needs three different type of linear components: horizontal vectors for root nodes, matrices for inner nodes and vertical vectors for leaf nodes. In the first step the algorithm computes required components. Identical components need to be computed just once. Only the number of values of a variable and its parent is essential: if there are identical numbers of outcomes in several locations in the forest, components are identical.

Let us denote the *root node component* with R_X, where X is the name of the node. The *inner node component* is denoted

with M_X^Y, where X is the name of the inner node and Y is the name of the parent node, and finally L_X^Y is the *leaf node component*. The operation between siblings (or between trees) is the entrywise product (Hadamard product) and denoted by symbol $'\odot'$. The operation between a parent and a child is the ordinary matrix multiplication. In the second step the algorithm executes these matrix computations. Notice that there is no need to compute matrix-matrix-operations. Computation proceeds always from leaves to root and the most demanding operations are therefore matrix-vector-multiplications. We can write computations using the matrix notation: take for example the forest ($B \leftarrow A \rightarrow C \rightarrow D$, $E \rightarrow F$) with six nodes. Now it is easy to write the computation using the matrix notation as $(R_A(L_B^A \odot (M_C^A L_D^C))) \odot (R_E L_F^E)$.

Next we proceed by specifying in subsections C, D and E the individual components in such a manner that the above algorithm provably computes the desired normalization term, but first we have to shortly discuss k-compositions, k-partitions and their relations to each other.

B. k-compositions and k-partitions

A single multinomially distributed variable forms the *multinomial model*. The sufficient statistics for this model is a k-composition, where k is the number of values in a variable. The *k-composition* of n can be represented as a k-tuple $\mathbf{x} = (x_1, \ldots, x_k)$, where $x_1 + \cdots + x_k = n$, $x_i \in \mathbb{N}$ and n is the number of data points [8]. For brevity, let us in the sequel denote the multinomial probability with maximum likelihood parameters for the observed counts by $c((x_1, \ldots, x_k))$:

$$c((x_1, \ldots, x_k)) = \frac{(\sum_{i=1}^k x_i)!}{\prod_{i=1}^k (x_i!)} \prod_{j=1}^k \left(\frac{x_j}{\sum_{i=1}^k x_i} \right)^{x_j}, \quad (3)$$

and hence the NML normalizing term for the multinomial model class is

$$\mathcal{C}_{MN}(k, n) = \sum_{x_1 + \cdots + x_k = n} c((x_1, \ldots, x_k)), \quad (4)$$

where the sum goes over the set of all k-compositions of the data size n [5]. But note that we can reduce the number of terms in the previous sum by using k-partitions instead of k-compositions: A *k-partition* is an unordered set of counts defined in a similar way as k-compositions. However, for simplicity, we represent each k-partition as a k-tuple with a standard decreasing order of counts.

We denote all k-partitions of a variable X by $q^X = [q_1^X, \ldots, q_p^X]$, where k is the number of values in variable X and p is the number of k-partitions. The number of data vectors is omitted from this notation, because we can determine it from k and p. To be sure that we multiply right elements with each other in the second step of the algorithm, we have to select also an fixed ordering among the set of k-partitions. We use the following ordering (the rationale of which becomes apparent in in section III-D): First order all k-partitions so that the first one is the one with most zero counts. Then comes the ones with one zero count less, and so on. After this, order each of these sub-blocks with a same number of zero counts using inverse lexicographic ordering.

When two k-compositions $\mathbf{x}$ and $\mathbf{y}$ have the same ordered k-partition, then the maximum likelihoods are also the same, i.e., $c(\mathbf{x}) = c(\mathbf{y})$. Consequently, if we use k-partitions instead k-compositions, we have to use an extra multiplier in the computations to take into account the fact that many k-compositions map to the same k-partition. Formally, the *multiplier function* is given by

$$m(\mathbf{x}) = \frac{k!}{\prod_{w \in \mathbf{x}} \mu_w(\mathbf{x})!}, \quad (5)$$

where $\mu_w(\mathbf{x}) = |u : x_u = w|$ tells how many times a value w appears in a k-partition $\mathbf{x}$.

C. Root node component

Root nodes have no parents, so the sufficient statistics of each root node goes simply through all k-partitions, where k is the number of values of the root node variable. The root node component is defined as a horizontal vector of multinomial probabilities with maximum likelihood parameters for all sufficient statistics of given data size n:

$$R_X = \begin{bmatrix} m(q_1^X) \cdot c(q_1^X) & \cdots & m(q_p^X) \cdot c(q_p^X) \end{bmatrix}, \quad (6)$$

where p is the number of k-partitions in the root node X. The multipliers collect all identical values into the same vector element. We naturally could have used k-compositions instead of k-partitions, but then the length of the vector would have increased radically: when the number of data vectors is large, there are $k!$-times more k-compositions than k-partitions.

D. Inner node component

The inner node component is a matrix. We define the matrix here, but all computational issues will be handled in section III.

Unlike in the case of root nodes, with an inner node we have to take into account the sufficient statistics of the parent node, and for that we introduce the *conditional term* $c(q_i^X | q_i^Y)$, where q_i^X is the sufficient statistics of a node and q_i^Y is the sufficient statistics of the parent node. In the root node case we can think that the missing parent node is a one valued node (a constant). The conditional term has in this case the form $c(q_i^X | (n)) = c(q_i^X)$. Hence we can split the data between different outcomes of the root node and get the correct result.

In the inner node case we can to think that the data is originally in l separate bins (we use the ball analogy here), where l is a number of possible outcomes of the parent node. We spread the data from each parent bin to the inner node's bins, but unfortunately we cannot do this independently, because we have to achieve the given k-partition, where k is the number of the inner node's possible values. As there are many different valid paths to get the right k-partition, our conditional term gets the form

$$c((x_1, \ldots, x_k) | (y_1, \ldots, y_l)) = \sum_{Z \in \mathcal{Z}} \prod_{i=1}^{l} c(z_{1i}, \ldots, z_{ki}), \quad (7)$$

where Z is a matrix with marginals $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_l)$, and the terms z_{ij} are elements of the matrix Z. All elements are positive integer values. An alternative representation for the matrix Z and its marginal sums is given by

$$Z = \begin{bmatrix} z_{11} & \cdots & z_{1l} \\ \vdots & \ddots & \vdots \\ z_{k1} & \cdots & z_{kl} \\ y_1 & \cdots & y_l \end{bmatrix} \begin{matrix} x_1 \\ \vdots \\ x_k \\ \end{matrix} . \tag{8}$$

Set $\mathcal{Z}$ is the set of those matrices Z which satisfy the given marginals:

$$\begin{aligned} \mathcal{Z} = \{Z | & z_{11} + \cdots + z_{k1} = y_1, \ldots, z_{1l} + \cdots + z_{kl} = y_l, \\ & z_{11} + \cdots + z_{1l} = x_1, \ldots, z_{k1} + \cdots + z_{kl} = x_k\}. \end{aligned} \tag{9}$$

To compute the conditional term, we have to form all the matrices which satisfy the given marginals. This is a tremendously heavy operation. Given the defined conditional terms, the general form of the inner node matrix is

$$M_X^Y = \begin{bmatrix} m(q_1^X) \cdot c(q_1^X | q_1^Y) & \cdots & m(q_p^X) \cdot c(q_p^X | q_1^Y) \\ \vdots & \ddots & \vdots \\ m(q_1^X) \cdot c(q_1^X | q_d^Y) & \cdots & m(q_p^X) \cdot c(q_p^X | q_d^Y) \end{bmatrix}, \tag{10}$$

where X is a node and Y is its parent node. Dimensions of this matrix are defined by the number of l-partitions and the number of k-partitions ($d \times p$). As before, multipliers exist as we are using k-partitions, not k-compositions.

E. Leaf node component

Leaf nodes are easier to handle than inner nodes, although they also have parent nodes. In the inner node case we have the target k-partition, but in the case of leaf nodes, any k-partition is valid. As we do not have a fixed target partition, we can just marginalize over all individual targets and the result is a product of ordinary multinomial normalizing terms defined in the formula (4). So we can separately compute each parent node bin, without any restrictions:

$$\begin{aligned} & \sum_{i=1}^{p} m(q_i^X) \cdot c(q_i^X | q_j^Y) \\ & \quad = \sum_{i=1}^{p} m((x_1, \ldots, x_k)_i) \cdot c((x_1, \ldots, x_k)_i | (y_1, \ldots, y_l)_j) \\ & \quad = \mathcal{C}_{MN}(k, y_1) \cdots \mathcal{C}_{MN}(k, y_l), \end{aligned} \tag{11}$$

where $(x_1, \ldots, x_l)_i$ is the ith k-partition and $(y_1, \ldots, y_l)_j$ is some l-partition. Efficient computation of the term $\mathcal{C}_{MN}(k, y_j)$ is not trivial, but fortunately there is a recent result showing how to use a simple recurrence formula to compute the term in linear time with respect to n [4].

The leaf node component is a vertical vector

$$L_X^Y = \begin{bmatrix} \sum_{i=1}^{p} m(q_i^X) \cdot c(q_i^X | q_1^Y) \\ \vdots \\ \sum_{i=1}^{p} m(q_i^X) \cdot c(q_i^X | q_d^Y) \end{bmatrix}. \tag{12}$$

The vector has d elements, where d is the number of l-partitions. Although the leaf vector can also be seen as an inner node matrix where we take a sum over the other dimension, luckily it is much easier to compute than the inner node matrix.

III. Efficient computation of inner node matrices

In the previous section we defined the inner node component, but we did not yet give an algorithm for computing it. One could of course make the obvious brute-force algorithm for finding the set of all matrices Z, and then compute each of the $c(q_i^X | q_j^Y)$ terms. However, this approach is clearly infeasible, as the number of matrices Z grows very fast when the number of data vectors increases. Also the size of the inner node component is growing at the same time.

Fortunately it is possible to use generating functions [2] and other computational simplifications for calculating the inner node components, as we will shortly see. These modifications make the computations feasible for small data sets, if the maximum number of values of the variables is relatively small. First we introduce generating polynomials, which form the very basis of our efficient computation.

A. Generating polynomials

A *multivariate generating polynomial* is a finite multivariate series of the form

$$\sum_{\mathbf{x} \in \mathcal{X}} a(x_1, \ldots, x_k) z_1^{x_1} z_2^{x_2} \cdots z_k^{x_k}, \tag{13}$$

where $\mathbf{x} = (x_1, \ldots, x_k)$ are vectors in a set of integer-valued vectors $\mathcal{X}$ and each variable $z_i \in \mathbb{C}$. The coefficients $a(x_1, \ldots, x_k)$ encode some relevant information. When we take a product of these generating polynomials, the coefficients of the resulting polynomial then correspond to higher level convolution operations. In the following we define our generating polynomials in such a manner that the result corresponds to the convolution operations we need.

Let us now have a polynomial of the form (13) with the sum going over all k-compositions of size u. A natural choice for the coefficients is the function $c((x_1, \ldots, x_k))$, in which case the resulting generating polynomial is

$$P_k^0 = 1, \quad \text{and} \tag{14}$$

$$P_k^u = \sum_{x_1 + x_2 + \cdots + x_k = u} c((x_1, \ldots, x_k)) z_1^{x_1} z_2^{x_2} \cdots z_k^{x_k}. \tag{15}$$

The coefficients of this polynomial describe the value of $c((x_1, \ldots, x_k) | (u))$, which is a root vector element without the multiplier. Now the trick is just to multiply these generating polynomials with respect to a parent node l-partition:

$$T_k^{(y_1, \ldots, y_l)} = P_k^{y_1} P_k^{y_2} \cdots P_k^{y_l}. \tag{16}$$

The coefficients of this polynomial now correspond to the needed conditional terms $c((x_1, \ldots, x_k) | (y_1, ..., y_l))$. In fact one can read all the k-partitions $(x_1, \ldots, x_k)$ for a given parent node l-partition from this new generating polynomial. So each product polynomial gives us a whole row of the inner

node matrix. Denoting the coefficient extraction by standard notation, we can write this as

$$c((x_1,\ldots,x_k)|(y_1,...,y_l)) = [z_1^{x_1}\cdots z_k^{x_k}]T_k^{(y_1,\ldots,y_l)}. \quad (17)$$

The above scheme gives us a concrete way to compute the conditional terms. The only remaining question is how to do the multiplication of these multivariate polynomials efficiently. For this we will not use normal multiplication methods for computing all the terms of the $T_k^{(y_1,\ldots,y_l)}$-polynomial: as we only need the coefficients of those terms which correspond to k-partitions, we can use the concept of a polytope to bound the set of computationally relevant terms.

B. Convex polytopes

As polygon is the name for a figure on a plane bounded by a finite number of line segments that form a closed path, a *polytope* is the name for a similar object in any dimension. The bounding line segments are called *facets*: facets of a k-dimensional polytope are $(k-1)$-dimensional and are itself polytopes. We can represent a *convex polytope* as an intersection of half spaces. *Integer lattice points* of a convex polytope are all the points $(x_1,\ldots,x_k)$ which belong to the polytope and have $x_i \in \mathbb{Z}$ for all i [1].

Now we define the set of all terms that we need for computing $T_k^{(y_1,\ldots,y_l)}$. First we need all those terms that correspond to our k-partitions of n. Second, because we are taking a product of polynomials, we need also all those terms that can produce a k-partition of n -terms via multiplication process. For example, $x^2y^2z^0$ times $x^2y^0x^2$ gives us the term $x^4y^2z^2$ which corresponds to a 3-partition $(4,2,2)$ of 8. These two sets of terms are all we need. We could map these points to a k-dimensional space, but we can in fact map these points also into a $(k-1)$-dimensional space, because one of the parameters is redundant. The redundancy is caused by the fact that the total degree of all the terms is the same in our case, as the polynomials we multiply are homogeneous. When we map our terms into a $(k-1)$-dimensional space, we drop the first count and say that each k-composition $(x_1,x_2,\ldots,x_k)$ corresponds to the point $(x_2,\ldots,x_k)$. It is most useful to drop the first count with the biggest value, because then we need to compute the least number of terms during the polynomial multiplications.

As we have now defined the set of all relevant terms, we want a compact representation for the set. This set happens to be a $(k-1)$-dimensional convex polytope, which we call *k-multiplication polytope*. We define the convex k-multiplication polytope using half spaces. There are two different kind of inequalities which define our polytope. First inequalities are of type

$$0 \le x_i \le \left\lfloor \frac{n}{i} \right\rfloor, \quad (18)$$

where x_i is value in the ith bin. There is one inequality for every bin except the first one, because it is redundant. Inequalities give the lowest and the highest count for every bin. These inequalities form a hyper rectangle. But there are still unnecessary terms inside this polytope, corresponding to invalid counts that are already too big to produce any valid k-partition of n. Therefore we need a second type of inequalities in addition.

To achieve the second type of inequalities, we make splits between the bins of a k-partition while preserving the order of bins. Notice that as the count x_1 is redundant, there cannot be a split between x_1 and x_2, so x_1 and x_2 are in the same group. For example, 4-partitions have three different splittings $\{x_1x_2|x_3x_4, x_1x_2|x_3|x_4,\ x_1x_2x_3|x_4\}$, where the vertical bar means the border of two groups. Now the new set of inequalities can be written using these formed groups. We name each group using the name of the last count in that group, and a group is then multiplied by the number of members in that group. We get three inequalities:

$$2x_2 + 2x_4 \le n,\ 2x_2 + x_3 + x_4 \le n,\ 3x_3 + x_4 \le n. \quad (19)$$

These inequalities describe situations where there are several bins with a same value. In general we get the second type of inequalities by finding all possible splittings and writing the inequalities in a similar manner.

Our polytope is now kind of a cage: we must compute all the terms that are inside the cage, but none of the outside ones. Next we will define how to do the polynomial multiplications.

C. Restricted multiplication of multivariate polynomials

We start by computing all those terms of a polynomial P_k^u that correspond to lattice points of a k-multiplication polytope. Notice that the polytope is defined by the data size n, not by the number of data points u in some parent bin. We assign coefficients $c((x_1,\ldots,x_k))$ to lattice point $(x_2,\ldots,x_k)$. This means that there will be many lattice points which will remain zero, as the polynomial P_k^u does not have the corresponding terms.

As we take a product of several multivariate polynomials, it is wise to multiply first the smaller ones, because then the number of resulting polynomial terms is minimized, as well as is the number of multiplications. This means that the multiplication order must be $P_k^{y_l}P_k^{y_{l-1}}\cdots P_k^{y_1}$ when computing the $T_k^{(y_1,\ldots,y_l)}$ polynomial.

Now we can define the actual operation between the values of lattice points of polytopes, so that the operation corresponds to the multiplication of multivariate polynomials. The value of the resulting polytope lattice point $(v_1,\ldots,v_r)$ is computed by

$$\mathcal{P}(v_1,\ldots,v_r) = \sum_{w_1=0}^{v_1}\cdots\sum_{w_r=0}^{v_r}\mathcal{P}_1(w_1,\ldots,w_r)\cdot\mathcal{P}_2(v_1-w_1,\ldots,v_r-w_r), \quad (20)$$

where $\mathcal{P}_1$ and $\mathcal{P}_2$ are the polytopes to be multiplied and $r = k-1$. We set previously the coefficients of a multivariate polynomial to lattice points of the polytope. Therefore, when we do the above operation for lattice points of the first two

polytopes, we see that the computed value of a lattice point is

$$c((h_1,\ldots,h_k)|(y_1,y_2)) = \sum_{Z\in\mathcal{W}} c(z_{11},\ldots,z_{1k})\cdot c(z_{21},\ldots,z_{2k}), \quad (21)$$

where $\mathcal{W}$ is the set of all matrices Z with marginals $(h_1,\ldots,h_k)$ and (y_1,y_2). From this we can see directly that when we do several multiplications, we get the term $c((x_1,\ldots,x_k)|(y_1,\ldots,y_l))$ defined in formula (7).

We explained how to compute the term $T_k^{(y_1,\ldots,y_l)}$ efficiently. Next we show that there is no need to compute the inner node matrices separately, as they all have common terms, which is a property we can utilize to make the computation even more efficient.

D. Core inner node matrix

Computation of the inner node matrices is still a very slow operation, but we can avoid unnecessary computational work by exploiting the particular order of partitions we earlier chose. Namely, if we use the given order, we can first compute a matrix, which we will name a *core inner node matrix*, as follows:

$$CM = \begin{bmatrix} c(q_1^{\max(X)}|q_1^{\max(Y)}) & \cdots & c(q_P^{\max(X)}|q_1^{\max(Y)}) \\ \vdots & \ddots & \vdots \\ c(q_1^{\max(X)}|q_D^{\max(Y)}) & \cdots & c(q_P^{\max(X)}|q_D^{\max(Y)}) \end{bmatrix}, \quad (22)$$

where $q_i^{\max(X)}$ and $q_j^{\max(Y)}$ are $\mathcal{K}$- and $\mathcal{L}$-partitions. Value D is the number of $\mathcal{L}$-partitions, where $\mathcal{L}$ is the maximum number of values that any inner node's parent has in the forest and P is the number of $\mathcal{K}$-partitions, where $\mathcal{K}$ is the maximum number of values any inner node has in the forest. Ignoring the multipliers, we notice that every inner node matrix is just a section from the core matrix. As we ordered the partitions so that we first have embedded 1-partitions, then embedded 2-partitions and so on, we can just make the following operation to the core matrix to get an inner matrix: If the number of values of a node is less than $\mathcal{K}$, we drop the last invalid columns from the right, and if the number of values of a parent is less than $\mathcal{L}$, we drop the last invalid rows from the bottom of the core matrix.

The reason why all inner node matrices have the same $c(q_i^X|q_j^Y)$ elements is easy to see: the bins with zero counts do not affect the value of the conditional term, so we can add as many zero counts as we want and the terms still remain same.

E. Efficiency and the time complexity

The time complexity of the whole algorithm reduces to one question: how much time does it take to compute the core matrix? We get a rough approximation for the time complexity in the following way: A size of a $\mathcal{K}$-multiplication polytope with given n is $\mathcal{O}(n^{\mathcal{K}-1})$. The maximum time that a polytope element multiplication takes is also $\mathcal{O}(n^{\mathcal{K}-1})$. We have to compute D-times these polytopes, where D is the number of $\mathcal{L}$-partitions of n. The complexity of this term is $\mathcal{O}(n^{\mathcal{L}-1})$. Therefore the time complexity of the whole algorithm is $\mathcal{O}(n^{2\mathcal{K}+\mathcal{L}-3})$. However, if we have precomputed the core matrix, then the time complexity of computing the NML for any forest (compatible to the core matrix) is $\mathcal{O}(Hn^{\mathcal{K}+\mathcal{L}-2})$, where H is the number of inner nodes in the forest. This means that for example with two-valued nodes, the time complexity is $O(n^3)$, but if we have precomputed the core matrix, we can compute the NML in time $O(Hn^2)$ for any forest structure. Note that this time complexity applies for all structures with any number of values in the leaf nodes, because the number of values in a leaf node does not affect the inner node computation.

We have run some tests for examining how large n is still computationally feasible in practice. The algorithm is coded using Perl and it utilizes some additional tricks. With 3-valued nodes, computing the core matrix for $n = 200$ took about 14 hours using a 3GHz computer, while with 4-valued nodes it took 34 hours to compute the core matrix for $n = 75$. Note also that the core matrix can be efficiently computed in parallel using multiple processors, as the rows of the core inner node matrix can be computed separately using different processor for each row. Core matrices can also be stored for later use.

IV. Conclusion

We presented an algorithm for computing the normalized maximum likelihood (NML) for tree structured Bayesian network models. The efficiency of the algorithm depends on the sizes of the alphabets used, and it is significantly more efficient than the only existing alternative reported in [10]. The algorithm offers us an opportunity to empirically compare the behavior of the NML approach to other graphical model selection methods in many non-trivial cases. Furthermore, as the algorithm computes exact results, this gives us also an opportunity to empirically validate approximative methods for computing the NML.

References

[1] M. Beck and S. Robins. *Computing the Continuous Discretely: Integer-point Enumeration in Polyhedra.* Springer, 2007.

[2] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. In preparation.

[3] D. Heckerman, D. Geiger, and D.M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, September 1995.

[4] P. Kontkanen and P. Myllymäki. A linear-time algorithm for computing the multinomial stochastic complexity. *Information Processing Letters*, 103(6):227–233, 2007.

[5] P. Kontkanen, P. Myllymäki, W. Buntine, J. Rissanen, and H. Tirri. An MDL framework for data clustering. In P. Grünwald, I.J. Myung, and M. Pitt, editors, *Advances in Minimum Description Length: Theory and Applications*. The MIT Press, 2006.

[6] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[7] J. Rissanen. Fisher information and stochastic complexity. *IEEE Transactions on Information Theory*, 42(1):40–47, January 1996.

[8] F. Ruskey. *Combinatorial Generation*. In preparation.

[9] Yu.M. Shtarkov. Universal sequential coding of single messages. *Problems of Information Transmission*, 23:3–17, 1987.

[10] H. Wettig, P. Kontkanen, and P. Myllymäki. Calculating the normalized maximum likelihood distribution for Bayesian forests. In *Proc. IADIS International Conference on Intelligent Systems and Agents*, Lisbon, Portugal, July 2007.