MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1652 January, 1999
C.B.C.L. Memo No. 169

An accelerated Chow and Liu algorithm: fitting tree distributions to high-dimensional sparse data

Marina Meilă

Abstract

Chow and Liu [2] introduced an algorithm for fitting a multivariate distribution with a tree (i.e. a density model that assumes that there are only pairwise dependencies between variables) and that the graph of these dependencies is a spanning tree. The original algorithm is quadratic in the dimesion of the domain, and linear in the number of data points that define the target distribution P. This paper shows that for sparse, discrete data, fitting a tree distribution can be done in time and memory that is jointly subquadratic in the number of variables and the size of the data set. The new algorithm, called the acCL algorithm, takes advantage of the sparsity of the data to accelerate the computation of pairwise marginals and the sorting of the resulting mutual informations, achieving speed ups of up to 2-3 orders of magnitude in the experiments.

This report describes research done at the Dept. of Electrical Engineering and Computer Science, the Center for Biological and Computational Learning and the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Dept. of Defense and by the Office of Naval Research. The author can be reached at M.I.T., Center for Biological and Computational Learning, 45 Carleton St., Cambridge MA 02142, USA. E-mail: mmp@ai.mit.edu

1 Introduction

Chow and Liu [2] introduced an algorithm for fitting a multivariate distribution with a tree, i. e. a density model that assumes that there are only pairwise dependencies between variables and that the graph of these dependencies is a spanning tree. Recently, the interest in this algorithm, hitherto called the CL algorithm, has been revived by the introduction of several probabilistic models that extend the spanning tree model. The mixture of trees of [9, 8] is a mixture model in which each component is a spanning tree with a possibly different topology over the set of variables. The TANB (Tree Augmented Naive Bayes) model of [5] is a mixture of trees designed for classification. Each class density is modeled by a tree; these trees are combined in a mixture where the mixing proportions represent the prior probabilities of the class variable. The name of the classifier reflects the fact that it is an extension of the Naive Bayes classifier of [1]. The CL algorithm as well as its extensions for the mixture of trees and for the TANB model can handle forests (acyclic graphs that are disconnected) but for the sake of clarity throughout this paper I will assume that all the trees that are learned are spanning trees. Also, as in the original work of Chow and Liu, we consider that all the variables are discrete, with finite range.

In the framework of graphical probability models tree distribution enjoy many properties that make them attractive as modeling tools: they are intuitively appealing, have low complexity and yet a flexible topology, sampling and computing likelihoods for trees are linear time, efficient and simple algorithms for marginalizing and conditioning exist. Mixtures of trees enjoy all the computational advantages of trees and, in addition, they are universal approximators over the the space of all distributions[1].

Moreover, trees are one of the few classes of graphical models for which structure search is tractable [6]. The CL algorithm finds the structure and parameters of the tree T that best fits a given distribuition P as measured by the Kullback-Liebler (KL) divergence

$$KL(P\,||\,T) \;=\; \sum_x P(x)\log\frac{P(x)}{T(x)} \qquad (1)$$

For this purpose, the algorithm uses the mutual information I_{uv} under P between each pair of variables u, v in the domain. When P is an empirical distribution obtained from and i.i.d. set of data, the computation of all the mutual information values requires time and memory quadratic in the number of variables n and linear in the size of the dataset N. Then a Maximum Weight Spanning Tree (MWST) algorithm [3] using the computed mutual information as edge weights finds the optimal tree structure. Finally, the parameters of the distribution are assigned by copying the values of the marginals of P corresponding to each edge of the tree. The most computationally expensive step in fitting a tree to data is the first step - computing the pairwise mutual informations in the empirical distribution. The time and memory requirements of this step are acceptable for certain problems but they may become prohibitive when the dimensionality n of the domain becomes large.

An example of such a domain is information retrieval. In information retrieval, the data points are documents from a data base and the the variables are words from a vocabulary. A common representation for a document is as a binary vector whose dimension is equal to the vocabulary size. The vector component corresponding to a word v is 1 if v appears in the document and 0 otherwise. The number N of documents in the data base is of the order of $10^3 - 10^4$. Vocabulary sizes too can be in the thousands or tens of thousands. This means that fitting a tree to the data necessitates $n^2 \sim 10^6 - 10^9$ mutual information computations and $n^2 N \sim 10^9 - 10^{12}$ counting operations, a number much too large for many of todays applications.

The present work shows how to improve on the time and memory requirements of the CL algorithm in order to allow the use of tree density models for higher dimensionality domains. It will also show how this improvement can be carried over to learning mixtures of trees or TANB models.

It is obvious that in general the proportionality with the size of the data set N cannot be improved on, since one needs to examine each data point at least once. Hence the focus will be on improving on the dependency on n. Here the situation is as follows: there are several MWST algorithms, some more efficient than others, but all the algorithms that I know of run in time at least proportional to the number of candidate edges. For our problem this number is $n(n-1)/2$ which would result in an algorithm that is at least quadratic in the number of variables n. But we also have additional information: weights are not completely arbitray; each edge uv has a weight equal to the mutual information I_{uv} between the variables u and v. Moreover, in the information retrieval example described above, the domain has a particularity: each document contains only a relatively small number of words (of the order of 10^2) and therefore most of the components of its binary vector representation are null. We call this property of the data *sparsity*. Can we use these facts to do better?

The answer is yes. Remark that the only way the MWST algorithm uses the edge weights is in comparisons. The idea the present work is based on is to compare mutual informations between pairs of variables **without actually computing them** when this is possible. This will result in a partial sorting of the edges by mutual information. A second idea is to exploit the sparsity of the data to speed up the most expensive step of the algorithm: computing the marginals for all pairs

[1] over discrete domains

of variables. Combining the two will result in an algorithm that (under certain assumptions) is jointly subquadratic in N and n w.r.t. both running time and memory. This algorithm, that we call **accelerated CL** algorithm (**acCL**) will be presented and analyzed in the rest of this paper.

I will start by briefly presenting the CL algorithm and the notation that will be used (section 2). The next section, 3, introduces the assumptions underlying the acCL algorithm. Section 4 develops the algorithm proper, in the special case of binary variables. The generalization to variables of arbitrary arity, to non-integer counts (as in the case of the EM algorithm) and a discussion of the usage of the acCL algorithm in conjunction with priors follow in sections 5, 6 and 7 respectively. Finally, experimental results are presented in section 8.

2 Fitting a tree to a distribution

In this section we introduce the tree distribution, the tree learning algorithm called the CL algorithm and the notation that will be used throughout the paper. For more detail on these topics the reader is recommended to consult [2, 9, 8, 7].

Let V denote the set of variables of interest, $|V| = n$. Let v denote a variable in V, r_v its number of values , x_v a particular value of v and x an n-dimensional vector representing an assignment to all the variables in V.

2.1 Tree distributions

We use trees as graphical representations for families of probability distributions over V that satisfy a common set of independence relationships encoded in the tree topology. In this representation, an edge of the tree shows a direct dependence, or, more precisely, the absence of an edge between two variables signifies that they are independent, conditioned on all the other variables in V. We shall call a graph that has no cycles a *tree*[2] and shall denote by E its edge set.

Now we define a probability distribution T that is *conformal* with a tree. Let us denote by T_{uv} and T_v the marginals of T:

$$\begin{aligned} T_{uv}(x_u, x_v) &= \sum_{x_{V-\{u,v\}}} T(x_u, x_v, x_{V-\{u,v\}}) \\ T_v(x_v) &= \sum_{x_{V-\{v\}}} T(x_v, x_{V-\{v\}}). \end{aligned}$$

Let $\deg v$ be the *degree* of vertex v, e.g. the number of edges incident to $v \in V$. Then, the distribution T is conformal with the tree (V, E) if it can be factored as:

$$T(x) = \frac{\prod_{(u,v)\in E} T_{uv}(x_u, x_v)}{\prod_{v\in V} T_v(x_v)^{\deg v - 1}} \tag{2}$$

[2]In the graph theory literature, our definition corresponds to a *forest*. The connected components of a forest are called trees.

The distribution itself will be called a tree when no confusion is possible.

2.2 Mixtures of trees

We define a mixture of trees to be a distribution of the form

$$Q(x) = \sum_{k=1}^{m} \lambda_k T^k(x) \tag{3}$$

with

$$\lambda_k \geq 0, \; k = 1, \ldots, m; \qquad \sum_{k=1}^{m} \lambda_k = 1. \tag{4}$$

The tree distributions T^k are the *mixture components* and λ_k are called *mixture coefficients*. A mixture of trees can be viewed as a containing an unobserved choice variable z, taking value $k \in \{1, \ldots m\}$ with probability λ_k. Conditioned on the value of z the distribution of the visible variables x is a tree. The m trees may have different structures and different parameters. A mixture of trees where all the trees have the same structure is equivalent to a TANB model [5].

2.3 The CL algorithm

Let us now turn to learning trees from data. The observed data are denoted by $\mathcal{D} = \{x^1, x^2, \ldots, x^N\}$, where each x^i represents a vector of observations for all the variables in V. Learning a tree in the Maximum Likelihood framework, means finding a tree distribution T^{opt} that satisfies

$$T^{opt} = \underset{T}{\operatorname{argmax}} \sum_{i=1}^{N} \log T(x^i) \tag{5}$$

This problem is a special case of the more general task of fitting a tree to a distribution P by minimizing $KL(P\,||\,T)$. The fact that the tree model is factorable allows for an efficient and elegant algorithm for solving this problem, owed to Chow and Liu [2]. The algorithm is briefly described here.

The algorithm is based on the fact that all the information about a distribution P that is necessary to fit a tree is contained in its *pairwise marginals* P_{uv}. If the distribution is an empirical distribution defined by a data set $\mathcal{D}$, computing these marginals from data is a computationally expensive step and takes $\mathcal{O}(n^2N)$ operations.

Further, to find the tree structure, given by its set of edges E, one has to compute the mutual information between each pair of variables in V under the target distribution P

$$I_{uv} = \sum_{x_u x_v} P_{uv}(x_u, x_v) \log \frac{P_{uv}(x_u, x_v)}{P_u(x_u)P_v(x_v)}, \quad u, v \in V, u \neq v. \tag{6}$$

Since V has n variables, there are $n(n-1)/2$ mutual informations to be computed. After that, the optimal tree structure E is found by a *Maximum Weight Spanning Tree* (MWST) algorithm using I_{uv} as the weight for edge $(u,v), \forall u,v \in V$. Computing the MWST can be done in several ways [3, 11, 4, 10]. The one we present here is called the Kruskal algorithm [3]. It is essentially a greedy algorithm. The candidate edges are sorted in decreasing order of their weights (i.e. mutual informations). Then, starting with an empty E, the algorithm examines one edge at a time (in the order resulting from the sort operation), checks if it forms a cycle with the edges already in E and, if not, adds it to E. The algorithm ends when $n-1$ edges have been added to E.

Once the tree is found, its marginals $T_{uv}\ (u,v) \in E$ are exactly equal to the corresponding marginals P_{uv} of the target distribution P.

$$T_{uv}^{opt} \equiv P_{uv} \text{ for } uv \in E \tag{7}$$

They are already computed as an intermediate step in the computation of the mutual informations I_{uv} (6).

3 Assumptions

This section presents the assumptions that help us develop the accelerated CL algorithm. Of the four assumptions stated below, the one that is essential is the sparsity assumptions. The first two will be dispensed of or relaxed later on and are made here for the sake of symplifying the presentation. The last is a technical assumption.

Binary variables. All variables in V take values in the set $\{0,1\}$. When a variables takes value 1 we say that it is "on", otherwise we say it is "off". Without loss of generality we can further assume that a variable is off more times than it is on in the given dataset.

Integer counts. The target distribution P is derived from a set of observations of size N. Hence,

$$P_v(1) = \frac{N_v}{N} = 1 - P_v(0) \tag{8}$$

where N_v represents the number of times variable v is on in the dataset. According to the first assumption,

$$0 < P_v(1) \leq \frac{1}{2} \quad \text{or} \quad 0 < N_v \leq \frac{1}{2}N \tag{9}$$

We can also exclude as non-informative all the variables that are always on (or always off) thereby ensuring the strict positivity of $P_v(1)$ and N_v.

Let us denote by N_{uv} the number of times variables u and v are simultaneously on. We call each of these events a *cooccurrence* of u and v. The marginal P_{uv} of u and v is given by

$$\begin{aligned} N.P_{uv}(1,1) &= N_{uv} && (10)\\ N.P_{uv}(1,0) &= N_u - N_{uv} && (11)\\ N.P_{uv}(0,1) &= N_v - N_{uv} && (12)\\ N.P_{uv}(0,0) &= N - N_v - N_u + N_{uv} && (13) \end{aligned}$$

All the information about P that is necessary for fitting the tree is summarized in the counts N, N_v and $N_{uv},\ u,v = 1,\dots,n$ that are assumed to be non-negative integers. From now on we will consider P to be represented by these counts.

Sparse data. Let us denote by $0 \leq |x| \leq n$ the number of variables that are on in observation x. Further, define s, the *sparsity* of the data by

$$s = \max_{i=1,N} |x^i| \tag{14}$$

If, for example, the data are documents and the variables represent words from a vocabulary, then s represents the maximum number of distinct words in a document. The time and memory requirements of the accelerated CL algorithm that we are going to introduce depend on the sparsity s. The lower the sparsity, the more efficient the algorithm. From now on, s will be assumed to be a constant and

$$s << n,\ N. \tag{15}$$

Data/dimension ratio bounded The ratio of the number of data points N vs. the dimension of the domain n is bounded.

$$\frac{N}{n} \leq R \tag{16}$$

This is a technical assumption that will be useful later. It is a plausible assumption for large n and N.

4 The accelerated CL algorithm

4.1 First idea: Comparing mutual informations between binary variables

The mutual information between two binary variables $u,v \in V$ can be expressed as:

$$\begin{aligned} I_{uv} &= H_u + H_v - H_{uv} \\ &= \frac{1}{N}\{-N_u \log N_u - (N-N_u)\log(N-N_u) + N\log N \\ &\quad - N_v \log N_v - (N-N_v)\log(N-N_v) + N\log N \\ &\quad + N_{uv}\log N_{uv} + (N_u - N_{uv})\log(N_u - N_{uv}) \\ &\quad + (N_v - N_{uv})\log(N_v - N_{uv}) \\ &\quad + (N - N_u - N_v + N_{uv})\log(N - N_u - N_v + N_{uv}) \\ &\quad - \log N\} \end{aligned} \tag{17}$$

Let us focus on the pairs u,v that do not cooccur, i.e. for which $N_{uv} = 0$. For such a pair, the above expression simplifies to

$$\begin{aligned} I_{uv} &= \frac{1}{N}\{-(N-N_u)\log(N-N_u) \\ &\quad -(N-N_v)\log(N-N_v) \\ &\quad +(N-N_u-N_v)\log(N-N_u-N_v) \\ &\quad +N\log N\} \end{aligned} \tag{18}$$

Knowing that N is fixed for the dataset, it follows that I_{uv} in the 0 cooccurrence case is function of 2 variables: N_v and N_u. Let us fix u (and consequently N_u) and

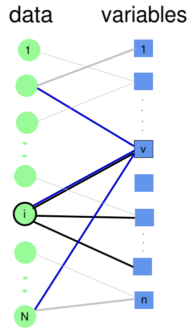

Figure 1: The bipartite graph representation of a sparse data set. Each edge iv means "variable v is on in data point i".

analyse the variation of the mutual information w.r.t. N_v:

$$\begin{aligned}\frac{\partial I_{uv}}{\partial N_v} &= \log(N - N_v) - \log(N - N_u - N_v) \\ &= \log\frac{N - N_v}{N - N_u - N_v} \;>\; 0 \qquad (19)\end{aligned}$$

It follows that if $N_{uv} = 0$ then the mutual information I_{uv} is monotonically increasing with N_v for a fixed u. Put in other terms, for a given variable u and any two variables v, v' for which $N_{uv} = N_{uv'} = 0$

$$N_v > N_{v'} \text{ implies that } I_{uv} > I_{uv'}.$$

This observation allows us to partially sort the mutual informations I_{uv} for non-cooccurring pairs u, v, without computing them. First, we have to sort all the variables by their number of occurrences N_v. This gives an ordering $\succ$ of all the variables in V.

$$v \succ u \;\Leftrightarrow\; N_v < N_u \qquad (20)$$

Then, for each u, we create the list of variable following u and not cooccurring with it:

$$V_0(u) = \{v \in V,\ v \succ u,\ N_{uv} = 0\} \qquad (21)$$

This list is sorted by decreasing N_v and therefore, implicitly, by decreasing I_{uv}. Since the data are sparse, most pairs of variables do not cooccur. Therefore, by creating the lists $V_0(u)$, a large number of the mutual informations are partially sorted. We shall show how to use this construction in section 4. Before that, let us examine an efficient way of computing the N_{uv} counts when the data are sparse.

4.2 Second idea: computing cooccurrences in a bipartite graph data representation

The bipartite graph data representation. Let $\mathcal{D} = \{x^1, \ldots x^N\}$ be a set of observations over n binary variables whose probability of being on is less than 1/2. It is efficient to represent each observation in $\mathcal{D}$ as a list of the variables that are on in the respective observation. Thus, data point $x^i, i = 1, \ldots N$ will be represented by the list $xlist^i = \text{list}\{v \in V | x^i_v = 1\}$. The name of this representation comes from its depiction in figure 1 as a bipartite graph $(V, \mathcal{D}, \mathcal{E})$ where each edge $(vi) \in \mathcal{E}$ corresponds to variable v being on in observation i. The space required by this representation is no more than

$$sN \;<<\; nN,$$

and much smaller than the space required by the binary vector representation of the same data.

Computing cooccurrences in the bipartite graph representation First, let us note that the total number N_C of cooccurrences in the dataset $\mathcal{D}$ is

$$N_C = \sum_{v \succ u} N_{uv} \;\leq\; \frac{1}{2}s^2 N \qquad (22)$$

where s is the previously defined sparsity of the data. Indeed, each data point x contains at most s variables that are on, therefore contributing at most $s(s-1)/2$ cooccurrences to the sum in (22).

As it will be shown shortly, computing all the cooccurrence counts takes the same amount of time, up to a logarithmic factor. The following algorithm not only computes all the cooccurrence numbers N_{uv} for $u, v \in V$ but also constructs a representation of the lists $V_0(u)$. Because the data are sparse, we expect the lists $V_0(u)$ to contain on average many more elements than their respective "complements"

$$\overline{V}_0(u) = \{v \in V,\ v \succ u,\ N_{uv} > 0\} \qquad (23)$$

Therefore, instead of representing $V_0(u)$ explicitly, we construct $\overline{V_0}(u)$ representing the list of all the variables that follow u and cooccur with u at least once, sorted by N_v. We assume that all the variables are already sorted by decreasing N_v, with ties broken arbitrarily and form a list L. Hence, to traverse the virtual list $V_0(u)$ in decreasing order of I_{uv}, it is sufficient to traverse the list L startin at the successor of u, skipping the variables contained in $\overline{V}_0(u)$.

For the variables that cooccur with u, a set of *cooccurrence lists* $C(u)$ is created. They that contain the same variables as $\overline{V}_0(u)$ (i.e the variables that follow u and cooccur with it) but sorted by the mutual informations I_{uv} rather then by N_v.

For each variable v we shall create a temporary storage of cooccurrences denoted by $cheap_v$ organized as a Fibonacci heap (or F-heap) [4]. Then, the cooccurrence counts N_{uv} and the lists $C(u), \overline{V}_0(u),\ u \in V$ can be computed by the following algorithm:

Algorithm **ListByCooccurrence**

Input list of variables sorted by decreasing N_u
dataset $\mathcal{D} = \{xlist^i,\ i = 1, \ldots N\}$

1.for $v = 1, \dots n$
  initialize $cheap_v$
2.for $i = 1, \dots N$
  for $u \in xlist^i$
    for $v \in xlist^i, v \succ u$
      insert u into $cheap_v$
3.for $v = 1, \dots n$ // empty $cheap_v$ and construct the lists
  $c = 0,\ u = v$
  while $cheap_v$ not empty
    u_{new} = extract max $cheap_v$
    if $(u_{new} \succ u)$ and $(u \succ v)$
      insert v in $\overline{V}_0(u)$ at the end
      insert (v, c) in $C(u)$
      $u = u_{new},\ c = 1$
    else
      $c++$
  if $(u \succ v)$ // last insertion
    insert v in $\overline{V}_0(u)$ at the end
    insert (v, c) in $C(u)$
4.for $u = 1, \dots n$
  for $(v, c) \in C(u)$
    compute I_{uv} and store it in $C(u)$ together with (v, c)
  sort $C(u)$ by decreasing I_{uv}

Output lists $\overline{V}_0(u), C(u)\ u = 1, \dots n$

The algorithm works as follows: $cheap_v$ contains one entry for each cooccurrence of u and v. Because we extract the elements in sorted order, all cooccurrences of v with u will come in a sequence. We save u and keep increasing the count c until a new value for u (denoted above by u_{new}) comes out of the F-heap. Then it is time to store the count c and v in u's lists. Since the v's are examined in decreasing order, we know that the new elements in the $\overline{V}_0(u)$ lists have to be inserted at the end. The position of the insertion in $C(u)$ is not important, since $C(u)$ will subsequently be sorted, but it is important to store the number of cooccurrences $c \equiv N_{uv}$ that will enables us to compute the mutual information I_{uv} in step 4 of the algorithm. The computed mutual informations are also stored.

Running time. As shown above, an insertion at the extremity of a list takes constant time. Extracting the maximum of an F-heap takes logarithmic time in the size of the heap. Thus, extracting all the elements of a heap $cheap_v$ of size l_u takes

$$\sum_{k=1}^{l_u} \log k = \log l_u! \leq l_u \log l_u.$$

Bounding the time to empty the heaps $cheap_v$. Extracting all elements of all heaps $cheap_v$ takes therefore less than $\tau = \sum_u l_u \log l_u$. Knowing already that

$$\sum_u l_u \ = \ N_C \ \leq \ 1/2 s^2 N \tag{24}$$

it is easy to prove that the maximum of τ is attained for all l_u equal. After performing the calculations we obtain $\tau \sim \mathcal{O}(s^2 N \log \frac{s^2 N}{n})$. The total number of list insertions is at most proportional to τ. It remains to compute the time needed to create $cheap_v$. But we know that insertion in an F-heap takes constant time and there are N_C cooccurrences to insert.

In the last step, we have to sort the lists $C(u)$. Sorting a list of length l' takes $l' \log l'$ time. The sum of all list lengths is no larger than the total number of cooccurrences N_C and by a reasoning similar to previous one we conclude that the running time of this step is also $\mathcal{O}(s^2 N \log \frac{s^2 N}{n})$. Therefore the whole algorithm runs in time of the order

$$\mathcal{O}(s^2 N \log \frac{s^2 N}{n}). \tag{25}$$

Memory requirements. The memory requirements for the temporary heaps $cheap_v$ are equal to N_C. The space required by the final lists $\overline{V}_0(u), C(u)$ is itself no more than proportional to the total number of cooccurrences N_C, namely $\mathcal{O}(s^2 N)$. Thus the total space required for this algorithm is $\mathcal{O}(s^2 N)$.

4.3 Putting it all together: the acCL algorithm and its data structures

So far, we have efficient methods for computing the cooccurrences and for partially sorting the mutual informations of the variables in $\overline{V}_0(u)$ for all $u \in V$. What we aim for is to create a mechanism that will output the edges uv in the decreasing order of their mutual information. If this is achieved, then the Kruskal algorithm can be used to construct the optimal tree.

We shall set up this mechanism in the form of a *Fibonacci heap* (F-heap) [4] called *vheap* that contains an element for each $u \in V$, represented by the edge with the highest mutual information among the edges $uv,\ v \succ u$. The maximum over this set will obviously be the maximum over the mutual informations of all the possible edges not yet eliminated. The record in *vheap* is of the form (c, u, v, I_{uv}), with $v \succ u$ and I_{uv} being the key used for sorting. Once the maximum is extracted, the used edge has to be replaced by the next largest (in terms of I_{uv}) edge in u's lists.

To perform this latter task it easy now: for each u we have the list of variables cooccurring with u, already sorted by mutual information. For the variables not cooccurring with u we have the virtual list $V_0(u)$ sorted by mutual information as well, but for which the mutual information values are not yet computed. All we have to do is to compute I_{uv_0} with v_0 being the head of $V_0(u)$. By comparing I_{uv_0} with the mutual information of the head of $C(u)$ we find

$$\max_{v \succ u} I_{uv}$$

This value, together with its corresponding u, v, N_{uv} is the value to be inserted in the F-heap. Every time an

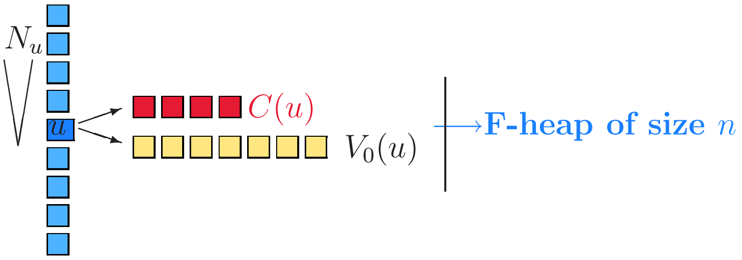

Figure 2: The acCL algorithm: the data structure that supplies the next weightiest candidate edge. Vertically to the left are the variables, sorted by decreasing N_u. For a given u, there are two lists: $C(u)$, the list of variables $v \succ u$, sorted in decreasing order of I_{uv} and (the virtual list) $V_0(u)$ sorted by decreasing N_v. The maximum of the two first elements of these lists is $\max_{v \succ u} I_{uv}$ that is inserted into an F-heap.The overal maximum of I_{uv} can then be extracted as the maximum of the F-heap.

edge (u, v, N_{uv}, I_{uv}) is inserted in *vheap*, v is deleted from its coresponding list. The data structures involved in this process are schematically shown in figure 4.3.

With this mechanism in place the Kruskal algorithm [3] can be used to construct the desired spanning tree. The outline of the algorithm is

Algorithm **acCL**

Inputvariable set V of size n
    dataset $\mathcal{D} = \{xlist^i,\ i = 1, \ldots N\}$
1. compute N_v for $v \in V$
    create *vlist*, list of variables in V sorted by decreasing N_v
2. **ListByCooccurrence**
3. create *vheap*
    for $u \in vlist$
        $v = \operatorname*{argmax}_{\text{head}C_u, \text{head}V_0(u)} I_{uv}$
        insert (c, u, v, I_{uv}) in *vheap*
4. E $=$ **KruskalMST**(*vheap*); store the $c = N_{uv}$ values for the edges added to E
5. for $uv \in E$
    compute the probability table T_{uv} using N_u, N_v, N_{uv} and N.

Output T

4.4 Time and storage requirements

Running time In the first step we have to compute the variables' frequencies N_v. This can be done by scanning trough the data points and by increasing the corresponding N_v each time v is found in $xlist^i$ for $i = 1, \ldots N$. This procedure will take at most sN operations. Adding in the time to initialize all N_v ($\mathcal{O}(n)$) and the time to sort the N_v values ($\mathcal{O}(n \log n)$) gives an upper bound on the running time for step 1 of the acCL algorithm of

$$\mathcal{O}(n \log n + sN)$$

The running time of the second step is already estimated to

$$\mathcal{O}(s^2 N \log \frac{s^2 N}{n})$$

Step 3, constructing a Fibonacci heap of size n, takes constant time per insertion, thus

$$\mathcal{O}(n)$$

An additional amount of time may be needed to extract elements from the virtual lists $V_0(u)$ by skipping the cooccurring elements, but this time will be accounted for in step 4.

Step 4 is the Kruskal algorithm. Each extraction from *vheap* is $\mathcal{O}(\log n)$. All the extractions from the virtually represented $V_0(u)$ take no more than $n_K + N_C$ time steps since there are at most N_C elements that have to be skipped. An extraction from $C(u)$ takes constant time. n_K is the number of steps taken by Kruskal's algorithm. After choosing a candidate edge, the Kruskal algorithm checks if it forms a cycle with edges already included in the tree, and if not, it includes it too. By using an efficient disjoint set representation [3] the former operation can be performed in constant time; the second operation, equivalent to insertion in a list, can be performed in constant time as well. Therefore, the running time for this step is

$$\mathcal{O}(n_K \log n + N_C)$$

The last step computes $n - 1$ probability tables, each of them taking constant time. Thus, its running time is

$$\mathcal{O}(n)$$

Adding up these five terms we obtain the upper bound for the running time of the acCL algorithm as

$$\mathcal{O}(n \log n + sn + s^2 N \log \frac{s^2 N}{n} + n_K \log n) \qquad (26)$$

Ignoring the logarithmic factors the bound becomes

$$\tilde{\mathcal{O}}(sn + s^2 N + n_K) \qquad (27)$$

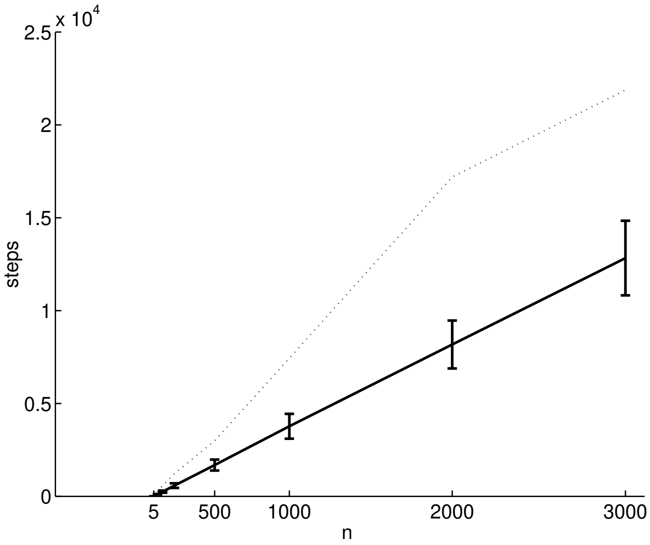

Figure 3: The mean (full line), standard deviation and maximum (dotted line) of Kruskal algorithm steps n_K over 1000 runs plotted against $n \log n$. n ranges from 5 to 3000. The edge weights were sampled from a uniform distribution.

For constant or bounded s, the above bound is a polynomial of degree 1 in the three variables n, N and n_K. However, we know that n_K the total number of edges inspected by Kruskal's algorithm has the range

$$n - 1 \;\leq\; n_K \;\leq\; \frac{n(n-1)}{2}. \tag{28}$$

Hence, in the worst case the above algorithm is quadratic in n. However, there are reasons to believe that in practice the dependence of n_K on n is subquadratic. Random graph theory suggests that if the distribution of the weight values is the same for all edges, then Kruskal's algorithm should take a number of steps proportional to $n \log n$ [12]. This result is sustained by experiments we have conducted: we ran Kruskal algorithm on sets of random weights over domains of dimension up to $n = 3000$. For each n, 1000 runs were performed. Figure 3 shows the average and maximum n_K plotted versus $n \log n$. The curves display a close to linear dependence.

Memory requirements To store data and results we need: $\mathcal{O}(sN)$ for the dataset in the bipartite graph representation, $\mathcal{O}(n)$ to store the variables and another $\mathcal{O}(n)$ to store the resulting tree structure and parametrization.

The additional storage required by the algorithm includes $\mathcal{O}(n)$ for $v - list$, then $\mathcal{O}(s^2N)$ for all the lists created in step 2 of the algorithm. In step 3, *vheap* is created taking up $\mathcal{O}(n)$ memory. The Kruskal algorithm requires $\mathcal{O}(n)$ storage to keep track of the tree topology. Computing the probability tables in step 5 requires no additional storage besides that taken up by the distribution itself.

Hence, the total space used by the algorithm is

$$\mathcal{O}(s^2N + n). \tag{29}$$

5 Generalization to discrete variables of arbitrary arity

This section will show how the Chow and Liu algorithm can be accelerated in discrete domains where the variables can take more than two values. We shall assume that a variable $v \in V$ takes r_v values, $2 \leq r_v \leq r_{MAX}$. The algorithm that we are going to develop is a simple extension of the acCL algorithm. Therefore, we shall present only the ideas that lead to extension and the modifications to the acCL algorithm that they imply.

The acCL algorithms introduced previously were exploiting the data sparsity. If they are to be generalized, it is first necessary to extend the notion of sparsity itself. Thus, in the forthcoming we shall assume that for each variable exists a special value that appears with higher frequency than all the other values. This value will be denoted by 0, without loss of generality. For example, in a medical domain, the value 0 for a variable would represent the "normal" value, whereas the abnormal values of each variable would be designated by non-zero values. Similarly, in a diagnostic system, 0 will indicate a normal or correct value, whereas the non-zero values would be assigned to the different failure modes associated with the respective variable.

An occurence for variable v will be the event $v \neq 0$ and a cooccurrence of u and v means that u and v are both non-zero in the same data point. Therefore, we define $|x|$ as the number of non-zero values in observation x

$$|x| \;=\; n - \sum_{v \in V} \delta_{x_v} \tag{30}$$

The sparsity s will be the maximum of $|x|$ over the data set, as before.

From the above it can be anticipated that the high frequency of the 0 values will help accelerate the tree learning algorithm. As before, we shall represent only the occurrences explicitly, creating thereby a compact and efficient data structure. Moreover, we shall demonstrate presorting mutual informations for non-cooccurring variables in this case can be done in the same way as before.

5.1 Computing cooccurrences

Following the previously introduced idea of not representing 0 values explicitly, each data point x will be replaced by the list *xlist* of the variables that occur in it. However, since there can be more than one non-zero value, the list has to store this value along with the variable index. Thus

$$xlist \;=\; \text{list}\{(v, x_v), v \in V, x_v \neq 0\}.$$

Similarly, a cooccurrence will be represented by the quadruple $(u, x_u, v, x_v),\; x_u, x_v \neq 0$. Counting and storing cooccurrences can be done in the same time as before and with a proportionally larger amount on memory, required by the additional need to store the (non-zero) variable values.

Instead of one cooccurrence count N_{uv} we shall now have a two-way contingency table N_{uv}^{ij}. Each N_{uv}^{ij} represents the number of data points where $u = i, v = j,\ i, j \neq 0$. This contingency table, together with the marginal counts N_v^j (defined as the number of data points where $v = j,\ j \neq 0$) and with N completely determine the joint distribution of u and v (and consequently the mutual information I_{uv}). Constructing the cooccurrence contingency tables multiplies the storage requirements of this step of the algorithm by $\mathcal{O}(r_{MAX}^2)$ but does not change the running time.

5.2 Presorting mutual informations

As in subsection 4.1, our goal is to presort the mutual informations I_{uv} for all $v \succ u$ that do not cooccur with u. We shall show that this can be done exactly as before. The derivations below will be clearer if they are made in terms of probabilities; therefore, we shall use the notations:

$$P_v(i) \;=\; \frac{N_v^i}{N}, \quad i \neq 0 \tag{31}$$

$$P_{v0} \;\equiv\; P_v(0) \;=\; 1 - \sum_{i\neq 0} P_v(i) \tag{32}$$

The above quantities represent the (empirical) probabilities of v taking value $i \neq 0$ and 0 respectively. Entropies will be denoted by H.

A "chain rule" expression for the entropy of a discrete variable. The entropy H_v of any multivalued discrete variable v can be decomposed in the following way:

$$\begin{aligned}
H_v \;=&\\
=&\; -P_{v0}\log P_{v0} - \sum_{i\neq 0} P_v(i)\log P_v(i)\\
=&\; -P_{v0}\log P_{v0} - (1-P_{v0})\sum_{i\neq 0}\frac{P_v(i)}{(1-P_{v0})}\left[\log\frac{P_v(i)}{(1-P_{v0})}\right.\\
&\; \left. + \log(1-P_{v0})\right]\\
=&\; \underbrace{-P_{v0}\log P_{v0} - (1-P_{v0})\log(1-P_{v0})}_{H_{v0}}\\
&\; -(1-P_{v0})\underbrace{\sum_{i\neq 0}\frac{P_v(i)}{(1-P_{v0})}\log\frac{P_v(i)}{(1-P_{v0})}}_{-H_{\overline{v}}}\\
=&\; H_{v0} + (1-P_{v0})H_{\overline{v}}
\end{aligned} \tag{33}$$

This decomposition represents a sampling model where first we choose whether v will be zero or not, and then, if the outcome is "non-zero" we choose one of the remaining values by sampling from the distribution $P_{v|v\neq 0}(i) = \frac{P_v(i)}{1-P_{v0}}$. H_{v0} is the uncertainty associated with the first choice, whereas $H_{\overline{v}} \equiv H_{v|v\neq 0}$ is the entropy of the outcome of the second one. The advantage of this decomposition for our purpose is that it separates the 0 outcome from the others and "encapsulates" the uncertainty of the latters in the number $H_{\overline{v}}$.

The mutual information of two non-cooccurring variables We shall use the above fact to find an expression of the mutual information I_{uv} of two non cooccurring variables u, v in terms of P_{u0}, P_{v0} and $H_{\overline{u}}$ only.

$$I_{uv} \;=\; H_u - H_{u|v} \tag{34}$$

The second term, the conditional entropy of u given v is

$$H_{u|v} \;=\; P_{v0}H_{u|v=0} + \sum_{j\neq 0} P_v(j)\underbrace{H_{u|v=j}}_{0} \tag{35}$$

The last term in the above equation is 0 because, for any non-zero value of v, the condition $N_{uv} = 0$ implies that u has to be 0. Let us now develop $H_{u|v=0}$ using the decomposition in equation (33).

$$H_{u|v=0} \;=\; H_{u0|v=0} + (1 - P_{u=0|v=0})H_{u|u\neq 0, v=0} \tag{36}$$

Because u and v are never non-zero in the same time, all non-zero values of u are paired with zero values of v. Hence, knowing that $v = 0$ brings no additional information once we know that $u \neq 0$. In probabilistic terms: $Pr[u = i|u \neq 0, v = 0] = Pr[u = i|u \neq 0]$ and

$$H_{u|u\neq 0, v=0} \;=\; H_{\overline{u}} \tag{37}$$

The term $H_{u=0|v=0}$ is the entropy of a binary variable whose probability is $Pr[u = 0|v = 0]$. This probability equals

$$\begin{aligned}
Pr[u=0|v=0] \;&=\; 1 - \sum_{i\neq 0} P_{u|v=0}(i)\\
&=\; 1 - \sum_{i\neq 0}\frac{P_u(i)}{P_{v0}}\\
&=\; 1 - \frac{1-P_{u0}}{1-P_{v0}}
\end{aligned} \tag{38}$$

Note that in order to obtain a non-negative probability in the above equation one needs

$$1 - P_{u0} \;\leq\; P_{v0}$$

a condition that is always satisfied if u and v do not cooccur.

Replacing the previous three equations in the formula of the mutual information, we get

$$\begin{aligned}
I_{uv} \;=&\; P_{u0}\log P_{u0} - P_{v0}\log P_{v0}\\
&\; + (P_{u0} + P_{v0} - 1)\log(P_{u0} + P_{v0} - 1)
\end{aligned} \tag{39}$$

an expression that, remarkably, depends only on P_{u0} and P_{v0}. Taking its partial derivative with respect to P_{v0} yields

$$\frac{\partial I_{uv}}{\partial P_{v0}} \;=\; \log\frac{P_{v0} + P_{u0} - 1}{P_{v0}} \;<\; 0 \tag{40}$$

a value that is always negative, independently of P_{v0}. This shows the mutual information increases monotonically with the "occurrence frequency" of v given by $1 - P_{v0}$. Note also that the above expression for the derivative is a rewrite of the result obtained for binary variables in (19).

We have shown that the acCL algorithm can be extended to variables taking more than two values by making only one (minor) modification: the replacement of the scalar counts N_v and N_{uv} by the vectors $N_v^j,\ j \neq 0$ and, respectively, the contingency tables $N_{uv}^{ij},\ i,j \neq 0$.

6 Using the acCL algorithm with EM

So far it has been shown how to accelerate the CL algorithm under the assumption that the target probability distribution P is defined in terms of integer counts[3] $N,\ N_v,\ N_{uv},\ u,v \in V$. This is true when fitting one tree distribution to an observed data set or in the case of classification with TANB models where the data points are partitioned according to the observed class variable. But an important application of the CL algorithm are mixtures of trees [9], and in the case of learning mixtures by the EM algorithm the counts defining P for each of the component trees are not integer.

Each Expectation step of the EM algorithm computes the posterior probability of each mixture component k of having generated data point x^i. This values is denoted by $\gamma_k(i)$. The values γ have the effect of "weighting" the points in the dataset $\mathcal{D}$ with values in $[0,1]$, different for each of the trees in the mixture. The counts N_v^k and N_{uv}^k corresponding to tree T^k are defined in terms of the γ values as

$$N^k = \sum_{i=1}^{N} \gamma_k(i) \tag{41}$$

$$N_v^k = \sum_{i:x_v^i=1} \gamma_k(i) \tag{42}$$

$$N_{uv}^k = \sum_{i:x_v^i=1 \wedge x_u^i=1} \gamma_k(i). \tag{43}$$

These counts are in general not integer numbers. Let us examine steps 1 and 2 of the acCL algorithm and modify them in order to handle weighted data.

First, remark that if the mixture has m components, step 1 will have to sort the variables m times, producing m different *vlists*, one for each of the components. Computing the N_v^k values is done similarly to the previous section; the only modification is that for each occurrence of v in a data point one adds $\gamma_k(i)$ to N_v^k instead of incrementing a counter. Remark that no operations are done for pairs of variables that do not cooccur in the original data set, preserving thereby the algorithm's guarantee of efficiency.

[3]In this and the subsequent sections we return to the binary variable assumption for the sake of notational simplicity.

For step 2, a similar approach is taken. At the time of inserting in $cheap_u^k$ one must store not only v but also $\gamma_k(i)$. When the heap is emptied, the current "count" c sums all the $\gamma_k(i)$ values corresponding to the given v.

Note also that one can use the fact that the data are sparse to accelerate the E step as well. One can precompute the "most frequent" likelihood $T_0^k \stackrel{\Delta}{=} T^k(0,\ldots,0)$ for each k. Then, if v is 1 for point x^i one multiplies T_0^k by the ratio $\frac{T^k_{v|\mathrm{pa}(v)}(1|\mathrm{pa}(v))}{T^k_{v|\mathrm{pa}(v)}(0|\mathrm{pa}(v))}$ also precomputed. This way the E step will run in $\mathcal{O}(msN)$ time instead of the previously computed $\mathcal{O}(mnN)$.

7 Factored priors and the acCL algorithm

All of the above assumes that the tree or mixture is to be fit to the data in the maximum likelihood framework. This section will study the possibility of using priors in conjunction with the acCL algorithm. The classes of priors that we shall be concerned with are the decomposable priors that preserve the parameter independence assumptions on which the CL algorithm is based. For an in-depth discussion of decomposable priors the reader is invited to consult [7]. We shall first examine priors on the tree's structure having the form

$$P(E) \ \propto\ \exp\left(-\sum_{uv\in E}\beta_{uv}\right)$$

This prior translates into a penalty on the weight of edge uv as seen by a MWST algorithm

$$W_{uv} \ \leftarrow\ I_{uv} - \frac{\beta_{uv}}{N}$$

It is easily seen that for general β_{uv} values such a modification cannot be handled by the acCL algorithm. Indeed, this would affect the ordering of the edges outgoing from u in a way that is inpredictable from the counts $N_v,\ N_{uv}$. However, if β_{uv} is constant for all pairs $u,v \in V$, then the ordering of the edges is not affected. All we need to do to use an acCL algorithm with a constant penalty β is to compare the I_{uv} of each edge, at the moment it is extracted from *vheap* by Kruskal's algorithm, with the quantity $\frac{\beta}{N}$. The algorithm stops as soon as one edge is found whose mutual information is smaller than the penalty $\frac{\beta}{N}$ and proceeds as before otherwise. Of course, in the context of the EM algorithm, N is replaced by N^k and β can be different for each component of the mixture. Remark that if all the variables are binary (or have the same number of values) an MDL type edge penalty translates into a constant β_{uv}.

Regarding Dirichlet priors on the tree's parameters, it is known [7] that they can be represented as a set of fictitious counts $N'_{uv},\ u,v \in V$ and that maximizing the posterior probability of the tree is equivalent to minimizing the KL divergence

$$KL(\tilde{P}\,||\,T)$$

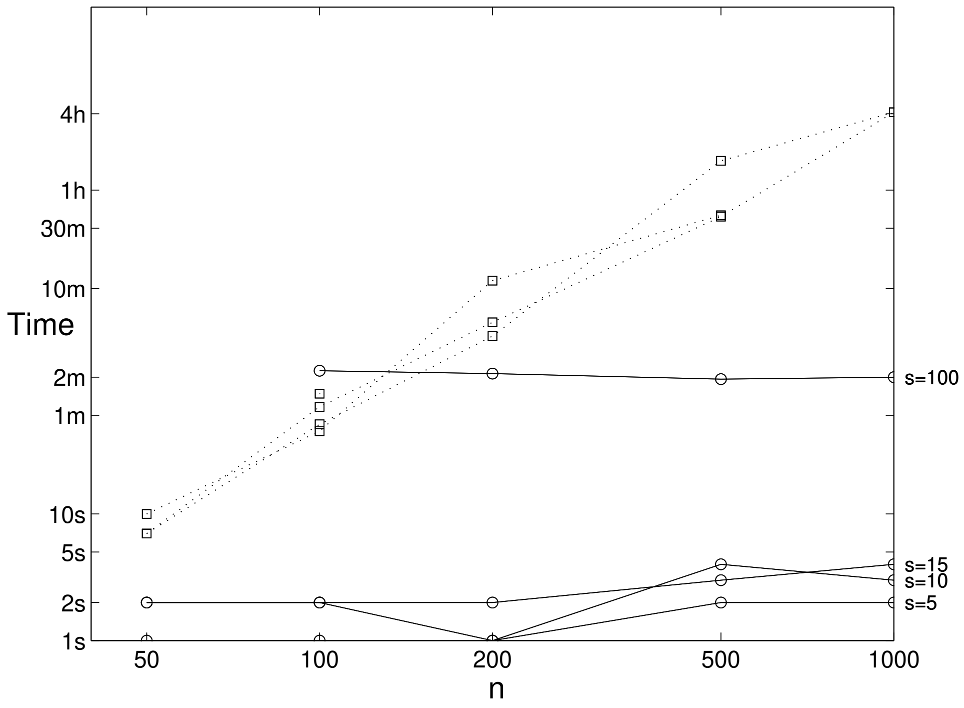

Figure 4: Real running time for the accelerated (full line) and traditional (dotted line) CL algorithms versus number of vertices n for different values of the sparsity s.

with $\tilde{P}$ a mixture between the empirical distribution P and the fictitious distribution P' defined by N'_{uv}.

$$\tilde{P} \;=\; \frac{N}{N+N'}P + \frac{N'}{N+N'}P' \qquad (44)$$

In this situation, the basic sparsity assumption that the acCL algorithm relies on may be challenged. Recall that it is important that most of the N_{uv} values are 0. If the counts N'_{uv} violate this assumption then the acCL algorithms become inefficient. In particular, the acCL algorithm degrades to a standard CL algorithm. Having many or all $N'_{uv} > 0$ is not a rare case. In particular, it is a characteristic of the non-informative priors that aim at smoothing the model parameters. This means that smoothing priors and the acCL algorithm will in general not be compatible.

Somehow suprisingly, however, the uniform prior given by

$$N'_{uv} \;=\; \frac{1}{r_u r_v} N' \qquad (45)$$

constitutes an exception: for this prior, in the case of binary variables, all the fictitious cooccurrence counts are equal to $N'/4$. Using this fact, one can prove that for very small and for large values of N' (> 8) the order of the mutual informations in $V_0(u)$ is preserved and respectively reversed. This fact allows us to run the acCL algorithm efficiently after only slight modification.

8 Experiments

The following experiments compare the (hypothesized) gain in speed of the acCL algorithm w.r.t the traditional Chow and Liu method under controlled conditions on artificial data.

The binary domain has a dimensionality n varying from 50 to 1000. Each data point has a fixed number s of variables being on. The sparsity s takes the values 5, 10, 15 and 100. The small values were chosen to gauge the advantage of the accelerated algorithm under extremely favorable conditions. The larger value will help us see how the performance degrades under more realistic circumstances. Each data point (representing a list of variables being on) was generated as follows: the first variable was picked randomly from the range $1, \ldots n$; the subsequent points were sampled from a random walk with a random step size between -4 and 4. For each pair n, s a set of 10,000 points was generated.

Each data set was used by both CL and acCL to fit one tree distribution and the running times were recorded and plotted in figure 4. The improvements over the traditional version for sparse data are spectacular: learning a tree over 1000 variables from 10,000 data points takes 4 hours by the traditional algorithm and only 4 seconds by the accelerated version when the data are sparse ($s = 15$). For $s = 100$ the acCL algorithm takes 2 minutes to complete, improving on the traditional algorithm by a factor of "only" 123.

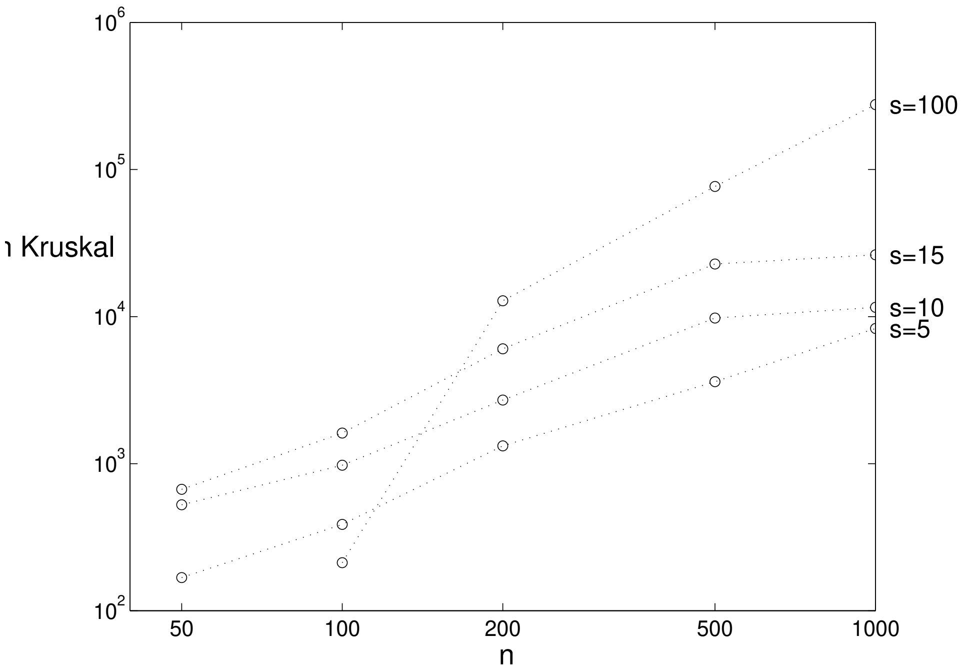

Figure 5: Number of steps of the Kruskal algorithm n_K versus domain size n measured for the acCL algorithm for different values of s.

What is also noticeable is that the running time of the accelerated algorithm seems to be almost independent of the dimension of the domain. On the other side, the number of steps n_K (figure 5) grows with n. This observation implies that the bulk of the computation lies with the steps preceding the Kruskal algorithm proper. Namely, that it is in computing cooccurrences and organizing the data that most of the time is spent. This observation would deserve further investigation for large real-world applications.

Figure 4 also confirms that the running time of the traditional CL algorithm grows quadratically with n and is independent of s.

9 Concluding remarks

This paper has presented a way of taking advantages of sparsity in the data to accelerate the tree learning algorithm.

The method achieves its performance by exploiting characteristics of the data (sparsity) and of the problem (the weights represent mutual informations) that are external to the Maximum Weight Spanning Tree algorithm proper. Moreover, it has been shown empirically that a very significant part of the algorithms' running time is spent in computing cooccurrences. This prompts future work on applying trees and mixtures of trees to high-dimensional tasks to focus on methods for structuring the data and for computing or approximating marginal distributions and mutual informations in specific domains.

Acknowledgements

I am indebted to Paul Viola for suggesting this topic and to David Karger for many stimulating discussions along the way.

References

[1] Peter Cheeseman and John Stutz. *Bayesian classification (AutoClass): Theory and results.* AAAI Press, 1995.

[2] C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *"IEEE Transactions on Information Theory"*, IT-14(3):462–467, May 1968.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[4] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Jounal of the Association for Computing Machinery*, 34(3):596–615, July 1987.

[5] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[6] David Heckerman, Dan Geiger, and David M. Chickering. Learning Bayesian networks: the com-

bination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[7] Marina Meilă. *Learning with Mixtures of Trees.* PhD thesis, Massachusetts Institute of Technology, 1999.

[8] Marina Meilă and Michael I. Jordan. Estimating dependency structure as a hidden variable. In M. I. Jordan and Sara Solla, editors, *Neural Information Processing Systems*, number 10, page (to appear). MIT Press, 1998.

[9] Marina Meilă, Michael I. Jordan, and Quaid D. Morris. Estimating dependency structure as a hidden variable. Technical Report AIM–1648,CBCL–165, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1998. (revised version of AIM–1611).

[10] R. Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16, 1973.

[11] Robert Endre Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, 1983.

[12] Douglas B. West. *Introduction to Graph Theory.* Prentice Hall, 1996.