Chapter 5

Accelerating the tree learning algorithm

"So do something! Fast!" said the Fish.
Dr. Seuss
–*The Cat in the Hat*

5.1 Introduction

The Chow and Liu (CL) algorithm presented in chapter 2 as the **TreeLearn** algorithm as well as the **MixTree** algorithm that builds on it are quadratic in the dimension of the domain n. This is due to the fact that to minimize the the KL divergence

$$KL(P\,||\,T) \;=\; \sum_x P(x)\log\frac{P(x)}{T(x)} \tag{5.1}$$

the **TreeLearn** algorithm needs 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. It has been shown (section 3.3.2) that this is the most computationally expensive step in fitting a tree to data. 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 $n^2N \sim 10^9 - 10^{12}$ counting operations! However, the problem has a particularity: each document contains only a relatively small number of words (of the order of 10^2) and therefore most of the component values of its corresponding binary vector are null. We call this property of the data *sparsity*. Can we use sparsity to improve on the time (and memory) requirements of the **TreeLearn** algorithm?

The rest of this chapter shows that the answer to this question is yes. It also shows how this improvement can be carried over to learning mixtures of trees or TANB models. We call the algorithms obtained *accelerated Chow and Liu* algorithms (**aCL**). The table below offers a preview of the results that will be achieved. The symbol $s \leq n$ represents a measure of the data sparsity, and N_K is the number of steps of the Kruskal algorithm.

	TreeLearn	**aCL**
E step	$\mathcal{O}(mnN)$	$\mathcal{O}(msN)$
M step	$\mathcal{O}(mn^2N)$	$\tilde{\mathcal{O}}[m(s^2N + sn + n_k)]$

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 dependence 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 n_e. For the tree learning problem $n_e = n(n-1)/2$ which results in algorithms that are at least quadratic in the number of variables n. But we also have additional information: weights are not completely arbitrary; each edge (uv) has a weight equal to the mutual information I_{uv} between the variables u and v. Can we use this fact to do better? 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** whenever this is possible. This way, by partially sorting the edges before running the MWST algorithm significant savings in running time can be achieved. A second but no less important idea is to exploit the sparsity of the data in computing the pairwise marginals P_{uv}. Combining the two will result in algorithms for fitting a tree to a distribution that (under certain assumptions) are jointly subquadratic in N and n w.r.t. both running time and memory. Let us start by stating the assumption underlying them.

5.2 Assumptions

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. The binary variables assumption will be eliminated later on (section 5.4).

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

$$P_v(1) \;=\; \frac{N_v}{N} = 1 - P_v(0) \tag{5.2}$$

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

$$0 \;<\; P_v(1) \leq \frac{1}{2} \quad \text{or} \quad 0 \;<\; N_v \leq \frac{1}{2}N \tag{5.3}$$

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

$$N.P_{uv}(1,1) \;=\; N_{uv} \tag{5.4}$$

$$N.P_{uv}(1,0) \;=\; N_u - N_{uv} \tag{5.5}$$

$$N.P_{uv}(0,1) \;=\; N_v - N_{uv} \tag{5.6}$$

$$N.P_{uv}(0,0) \;=\; N - N_v - N_u + N_{uv} \tag{5.7}$$

All the information about P that is necessary for fitting the tree is summarized in the counts N, N_v and N_{uv}, $u, v = 1, \ldots, n$ that are assumed to be non-negative integers. From now on we will consider P to be represented by these counts. Later on (section 5.3.5) this assumption will be considerably relaxed.

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

$$s = \max_{i=1,N} |x^i| \tag{5.8}$$

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

$$s << n,\ N. \tag{5.9}$$

As we shall see, of the four assumptions introduced here, the sparsity assumption is the only assumption that we cannot dispense with in our program of accelerating the tree learning algorithm.

Data/dimension ratio bounded The ratio of the number of data points N vs. the dimension of the domain n is bounded and bounded away from 0.

$$0 \ < \ R_{min} \ \leq \frac{N}{n} \ \leq \ R_{max} \tag{5.10}$$

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

5.3 Accelerated CL algorithms

5.3.1 First idea: Comparing mutual informations between binary variables

The mutual information between two variables $u, v \in V$ can be expressed with the above notations as:

$$\begin{aligned} I_{uv} &= H_u + H_v - H_{uv} \\ &= \frac{1}{N}\{-N_u \log N_u - (N-N_u)\log(N-N_u) + N\log N \\ &\quad - N_v \log N_v - (N-N_v)\log(N-N_v) + N\log N \\ &\quad + N_{uv}\log N_{uv} + (N_u - N_{uv})\log(N_u - N_{uv}) + (N_v - N_{uv})\log(N_v - N_{uv}) \\ &\quad + (N - N_u - N_v + N_{uv})\log(N - N_u - N_v + N_{uv}) - \log N\} \end{aligned} \tag{5.11}$$

Knowing that N is fixed for the dataset, it follows that I_{uv} is a function of 3 variables: N_v, N_u and N_{uv}. Let us fix N_{uv} and N_u and analyze the variation of the mutual information w.r.t. N_v:

$$\begin{aligned} \frac{\partial I_{uv}}{\partial N_v} &= \log N_v - \log(N-N_v) - \log(N_v - N_{uv}) + \log(N - N_u - N_v + N_{uv}) \\ &= \log \frac{N_v(N - N_u - N_v + N_{uv})}{(N_v - N_{uv})(N - N_v)} \end{aligned} \tag{5.12}$$

Equating the above derivative to 0 one obtains the extremum:

$$N_v^* \;=\; \frac{N_{uv}N}{N_u} \tag{5.13}$$

It is easy to verify that N_v^* is a minimum and that it corresponds to mutual independence between u and v. Note that N_{uv}^* is not necessarily an integer.

But more importantly, this calculation has the following practical implication: assume that u is fixed and thereby N_u is also fixed. Assume also that the variables in V are sorted by decreasing N_v and that we are interested only in the variables following u in this ordering. We denote this latter fact by $v \succ u$. To express it in an intuitive way, assume that edge uv is directed from the lower ranking variable toward the higher ranking one. In this case the mutual informations between u and the variables v following u correspond to the weights of the edges outgoing from u. Partition this set of variables (or edges) into sets $V_c, c = 0, 1, \ldots$ according to the number $c = N_{uv}$ of cooccurrences with u. Further partition the sets $V_c,\ c > 0$ into

$$\begin{aligned} V_c^+ \;&=\; \{v \in V | v \succ u \text{ and } N_{uv} = c \text{ and } N_v > N_v^*\} && (5.14)\\ V_c^- \;&=\; \{v \in V | v \succ u \text{ and } N_{uv} = c \text{ and } N_v \leq N_v^*\} && (5.15) \end{aligned}$$

For $c = 0$ $N_v^* = 0$ so that $V_0 \equiv V_0^+$. For each of these subsets I_{uv} varies monotonically (increasing or decreasing) with N_v. It means that if V_c^- and V_c^+ are sorted by increasing respectively decreasing N_v then the variables within each set are sorted in the decreasing order of their mutual information with u. This way, we can achieve a partial sorting of the mutual informations I_{uv} for a given u without computing any of the mutual informations involved. To obtain $v = \underset{v \succ u}{\operatorname{argmax}}\, I_{uv}$ we only need to calculate and compare the first elements of the lists $V_c^\pm,\ c = 0, 1, \ldots$. This procedure can save substantial amounts of computation provided that the total number of values of $c = N_{uv}$ is small for each u.

If the data are sparse, then most pairs of variables do not cooccur. Consequently, the list $V_c^\pm(u)$ for $c > 0$ have, together, much fewer elements than the $V_0(u)$ lists. One can pool the former together, for each u, without significant cost in computation time. Of the two accelerated algorithms that are introduced here, the first costructs and uses all the $V_c^\pm$ lists, whereas the second one uses only V_0.

5.3.2 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 $xlist^i = \text{list}\{v \in V | x_v^i = 1\}$. The name of this representation comes from its depiction in figure 5-1 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.

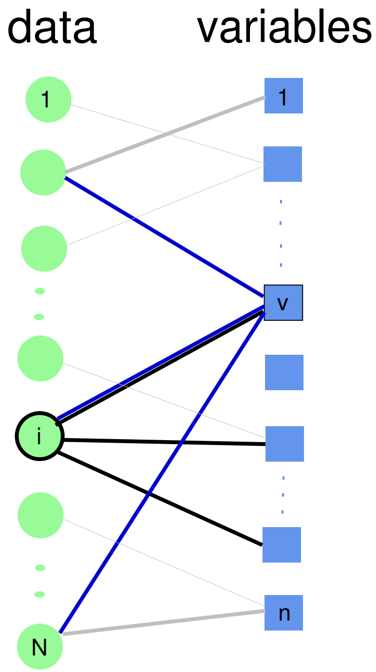

Figure 5-1: The bipartite graph representation of a sparse data set. Each edge iv means "variable v is on in data point i".

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

$$N_C \;=\; \sum_{v \succ u} N_{uv} \;\leq\; \frac{1}{2} s^2 N \tag{5.16}$$

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 (5.16). Hence, the above result.

Therefore, 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 the lists $V_c^{\pm},\ c > 0$. Because the data are sparse, we expect the lists $V_0(u)$ to be on average much larger that the other $V_c^{\pm}(u)$ lists. Therefore, instead of representing $V_0(u)$ explicitly, we construct its complement $\overline{V_0}(u)$ representing the sorted union of all the lists $V_c^{\pm}(u),\ c > 0$ or, in other words, the list of all the variables that cooccur with u at least once. We assume that the variables are already sorted by decreasing N_v, with ties broken arbitrarily.

This algorithm will serve as a building block for the accelerated Chow and Liu (**aCL**) algorithms that the next subsections will introduce. One of them, **aCL-I**, uses all the $V_c^{\pm}$ lists as shown below. The second one, **aCL-II** simplifies this process by creating only the $\overline{V}_0(u)$ lists and pooling the contents of the rest of the lists (i.e. the variables that cooccur with u) in one list C_u sorted by the mutual information I_{uv}.

For each variable u we shall initialize a temporary storage of cooccurrences denoted by $cheap_u$ organized as a Fibonacci heap (or F-heap) [20]. Then, the lists $V_c^{\pm}(u),\ c > 0,\ u \in V$ can be computed as follows:

Algorithm **ListByCooccurrence**

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

1.for $u = 1, \ldots n$
    initialize $cheap_u$
2. for $i = 1, \ldots N$
    for $u \in xlist^i$
        for $v \in xlist^i, v \succ u$
            insert v into $cheap_u$
3.for $u = 1, \ldots n$
    construct lists $V_c^{\pm}(u), c > 0$
    construct list $\overline{V_0}(u)$

Output lists $V_c(u),\ \overline{V_0}(u)\ u = 1, \ldots n,\ c > 0$

To construct the set of lists $V_c(u), c > 0$ we proceed as follows:
    ConstructLists(u)

compute N_v^*
$c = 0$
$v = u$
while $cheap_u$ not empty
    $vnew$ = extract max $cheap_u$
    if $vnew \succ v$ // new set of cooccurrences
        Insert $v,\ c$ *in corresponding lists*
        $v = vnew$
        $c = 1$
    else // one more cooccurence for the same v
        c++
Insert $v,\ c$ *in corresponding lists* //insert last set of cooccurrences

To insert $v,\ c$ in corresponding lists we do:

if $v \succ u$ // not first step
    if $N_v < N_v^*$
        insert v in $V_c^-(u)$ at the beginning
    else
        insert v in $V_c^+(u)$ at the end
    insert v in $\overline{V_0}(u)$ at the end

The algorithm works as follows: $cheap_u$ 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 store v and keep increasing the count c until a new value for v comes out of the F-heap. Then it is time to store the previous variable in its corresponding $V_c(u)$. Since the different v's come in decreasing order, we know that in V_c^+ we have to insert at the end, whereas in V_c^- we have to insert at the beginning to obtain the desired sorting of the lists. $\overline{V_0}(u)$ is the list of all variables that cooccur with u, sorted by decreasing N_v. Since it is assumed that most variables do not cooccur with u, it is more efficient to store $\overline{V_0}(u)$ than its complement $V_0(u)$.

Insertion in each V_c assumes implicitly that we test if the list was created and create it if necessary. To handle insertion in $V_c^\pm$ efficiently, we store the current maximum value of c for each u in the variable $c_{max}(u)$. We maintain a vector of pointers to the V_c lists with dimension $2c_{max}(u)$. If a list is empty its corresponding pointer is a null pointer. This way, each insertion in a $V_c^\pm$ list takes constant time. This time includes checking that the list is initialized. Each insertion in one of the $V_c^\pm$ lists is followed by the insertion of the same variable at the end of the list $\overline{V_0}(u)$, which is also a constant time operation. The amount of additional storage incurred by this method will be discussed later on.

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 $cheap_u$ of size L_u takes $\sum_{l=1}^{L_u} \log l = \log L_u! \leq L_u \log L_u$.

Bounding the time to empty the heaps $cheap_u$**.** Extracting all elements of all heaps $cheap_u$ takes therefore less than $\tau = \sum_u L_u \log L_u$. Knowing already that $\sum_u L_u = N_C \leq 1/2s^2N$ it is easy to prove that the maximum of τ is attained for all L_u equal. After performing the calculations we obtain $\tau \sim \mathcal{O}(s^2 N \log \frac{N}{n})$. The total number of list insertions is at most proportional to τ. It remains to compute the time needed to create $cheap_u$. But we know that insertion in a F-heap takes constant time and there are N_C cooccurrences to insert. Therefore the whole algorithm runs in $\mathcal{O}(s^2 N \log \frac{N}{n})$ time.

Memory requirements. The memory requirements for the temporary heaps $cheap_u$ are equal to N_C. The space required by the final lists $V_c^\pm(u)$ is no more than proportional to the total number of cooccurrences, namely $\mathcal{O}(s^2N)$. It remains to bound the space taken up by the vectors of pointers to the lists. This space is no larger than

$$\sigma_1 \;=\; \sum_{u \in V} 2c_{max}(u).$$

The total number of cooccurrences N_C is

$$\sum_{u \in V} \sum_{c=1}^{\infty} c[\text{length}(V_c^+(u)) + \text{length}(V_c^-(u))] \;=\; N_C \;\leq\; s^2N/2 \tag{5.17}$$

Under this constraint, σ_1 is maximized when $\text{length}(V_c^\pm(u)) = 0$ for all but one of the lists with $c = c_{max}(u)$ for which it is equal to 1. Then the l.h.s. of (5.17) becomes

$$\sum_{u \in V} c_{max}(u).$$

It follows that $\sigma_1 \leq s^2N$. Thus the total space required for this algorithm is $\mathcal{O}(s^2N)$.

5.3.3 Putting it all together: the aCL-I algorithm and its data structures

So far, we have an efficient method of partially sorting the mutual informations of all the candidate edges. What we aim for is to create a mechanism that will output the edges uv in the decreasing order of their mutual information. We shall set up this mechanism in the form of an F-heap called *vheap* that contains an element for each $u \in V$, represented by the edge with the highest mutual information among the edges outgoing from 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 weightiest edge in u's lists.

With this in place the Kruskal algorithm can by used to construct the desired spanning tree. The outline of the algorithms is

Algorithm **aCL-I** - outline

$n_e \;=\; 0$ // the number of tree edges
$E \;=\; \Phi$
while $n_e < n - 1$
  $(uv) \;= \text{extract max}(vheap)$
  if (uv) does not create a cycle with edges already in E
    add (uv) to E
    $n_e + +$
  get a new (uv') and insert it in *vheap*
Output E

It remains to show how to efficiently construct a suitable *vheap*. This is not hard, once we have the lists $V_c^{\pm}(u)$, $\overline{V_0}(u)$ and *vlist* the sorted list of all variables. It suffices to take the first element of each list, let it be called $v_c^{\pm},\ c > 0$, compute the respective mutual information I_{uv} and insert the triple $(c, v_c^{\pm}, I_{uv})$ in a F-heap $iheap_u$. How to extract the first element of the implicitly represented $V_0(u)$ will be discussed later. Extracting the maximum of $iheap(u)$ will provide us with the desired maximum over all edges originating in u. The quadruple (c, u, v, I_{uv}) is inserted in *vheap* and c is also used to replace the eliminated edge with one from the same list.

It remains to show how to handle the variables that do not cooccur with u. For this we maintain a pointer p_u into *vlist*. Initially p_u points to the successor of u in *vlist*. We compare p with the first element of $\overline{V_0}(u)$ and if they are equal we increment p by one and delete the head of $\overline{V_0}(u)$ recursively, until we find a v in *vlist* that does not cooccur with u. For this v we compute the mutual information and insert the triple $(0, v, I_{uv})$ in *ilist*. To get the next variable not cooccurring with u we increment p and repeat recursively the above comparison procedure with the head of $\overline{V_0}(u)$. Now we can summarize the algorithm as

Algorithm **aCL-I**

Input variable 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** // partial sort the mutual informations
3. create *vheap*
  for $u \in vlist$
    create $iheap_u$
    $(c, v, I_{uv}) = \text{extract max with replacement from } iheap_u$

insert (c, u, v, I_{uv}) in *vheap*
4. E $=$ **KruskalMST**(*vheap*) storing 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

5.3.4 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 **aCL** algorithm of

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

The running time of the second step is already estimated to

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

Step 3 takes one mutual information computation for the head of each $V_c^{\pm}(u)$ list, plus $n \log C_{max}$ operations to extract the maximum from each $iheap_u$ and insert it into *vheap* (the reader is reminded that insertion in an F-heap is constant time). By C_{max} we denote the maximum over u of the number of non-empty $V_c^{\pm}(u)$ lists associated to a variable $u \in V$. The total number of lists is denoted by N_L. With these notations the total time for this step is

$$\mathcal{O}(N_L + n \log C_{max})$$

Step 4 is the Kruskal algorithm. For each edge extracted from *vheap* another one has to be inserted. Extraction takes $\mathcal{O}(\log n)$ because the size of *vheap* is always $\leq n$; a new insertion involves an extraction from an *iheap*, done in $\mathcal{O}(\log C_{max})$ and one computation of a mutual information, taking constant time. Checking that the current edge (uv) does not create a cycle and adding it to E can be done in constant time. Thus, if we denote by n_K the total number of edges examined by the Kruskal algorithm this step takes a number of operations of the order

$$\mathcal{O}(n_K \log nC_{max})$$

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 aCL algorithm as

$$\mathcal{O}(n \log n + s^2 N \log \frac{N}{n} + N_L + n \log C_{max} + n_K \log(nC_{max})) \tag{5.18}$$

Let us now further refine this result, by bounding N_L and C_{max}. Appendix 5.9 computes an upper bound on the number of lists N_L. If our third assumption holds (N/n bounded away from 0 and ∞) then this bound reduces to $\mathcal{O}(sn)$. For C_{max} we shall use the obvious upper bound

$$C_{max} \;\leq\; N_C.$$

Notice that since C_{max} influences the running time only by its logarithm, any bound that is polynomial in s, N and n will do. Now the expression of the running time for the aCL algorithm becomes

$$\mathcal{O}(n \log n + s^2 N \log \frac{N}{n} + sn + n \log s^2 N + n_K \log(s^2 nN)) \tag{5.19}$$

which further simplifies to

$$\mathcal{O}(n \log n + s^2 N \log \frac{N}{n} + sn + n_K \log(s^2 nN)) \;=\; \tilde{\mathcal{O}}(n + s^2 N + sn + n_K) \tag{5.20}$$

This bound, ignoring the logarithmic factors, 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

$$n - 1 \;\leq\; n_K \;\leq\; \frac{n(n-1)}{2}. \tag{5.21}$$

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 $n \log n$ [71]. 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 5-2 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 *vlist*, then $\mathcal{O}(s^2 N)$ for all the lists created in step 2 of the algorithm. In step 3, *vheap* is created taking up $\mathcal{O}(n)$ memory. The space occupied by all the *ilist*s is proportional to the number of their elements, namely N_L. As we have already seen, this number is $\mathcal{O}(sn)$.

The last two steps do not use auxiliary storage so that the total space used by the algorithm is

$$\mathcal{O}(s^2 N + n + N_L) \tag{5.22}$$

or, using the bound in appendix 5.9

$$\mathcal{O}(s^2 N + sn) \tag{5.23}$$

5.3.5 The aCL-II algorithm

When the data are sparse enough, one can simplify the **aCL-I** algorithm. The $V_c^{\pm}(u)$ lists for $c > 0$ can be replaced by a single list C_u of all the variables cooccurring with u. The difference between C_u and the list $\overline{V}_0(u)$ introduced previously is that the records in C_u contain as an additional field the mutual information I_{uv} and are sorted thereby. The list

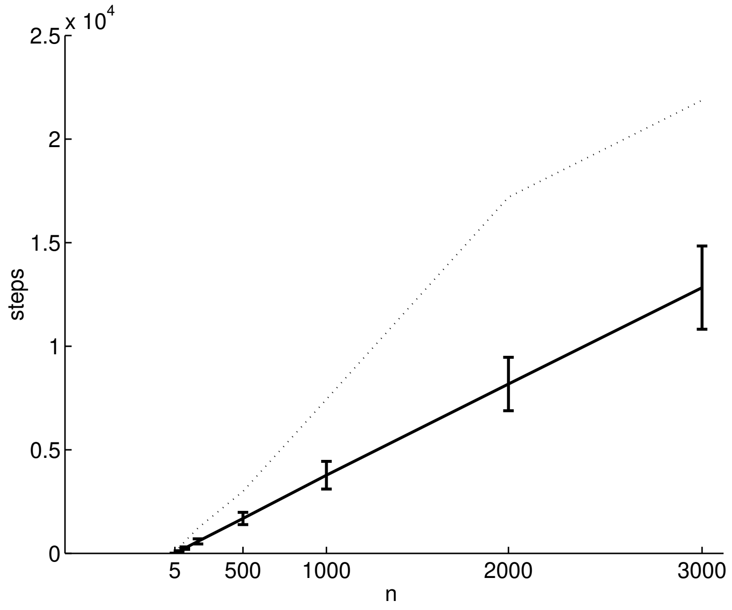

Figure 5-2: The mean (full line), standard deviation and maximum (dotted line) of Kruskal algorithm steps n_K over 1000 runs plotted against $n \log n$. n ranges from 5 to 3000. The edge weights were sampled from a uniform distribution.

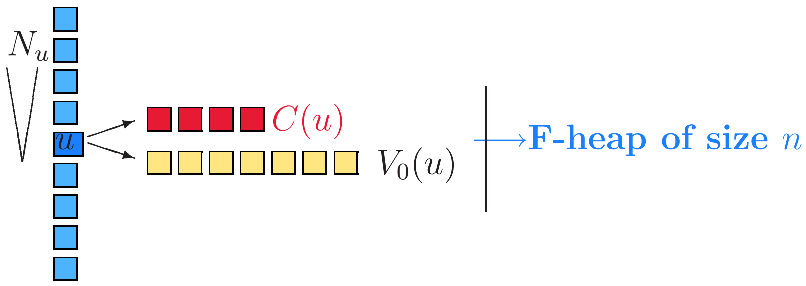

Figure 5-3: The **aCL-II** algorithm: the data structure that supplies the next weightiest candidate edge. Vertically to the left are the variables, sorted by decreasing N_u. For a given u, there are two lists: C_u, the list of variables $v \succ u$, sorted in decreasing order of I_{uv} and (the virtual list) $V_0(u)$ sorted by decreasing N_v. The maximum of the two first elements of these lists is inserted into an F-heap.The overal maximum of I_{uv} can then be extracted as the maximum of the F-heap.

$\overline{V}_0(u)$ preserves its function of supplying (together with *vlist*) the variables not cooccurring with u in decreasing order of their mutual information with u.

This implies computing all $I_{uv},\ v \in C_u,\ u \in V$ ahead of time, but they are no more than $N_C < 1/2s^2N$. The maximum weight edge outgoing from u is obtained by comparing the two mutual informations corresponding to the current heads of lists C_u and $V_0(u)$. The data structure used is shown schematically in figure 5-3 and the algorithm is described below.

Algorithm **aCL-II**

Input variable 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. construct lists $\overline{V}_0(u),\ u \in V$
  compute mutual informations $I_{uv},\ v \in \overline{V}_0(u),\ u \in V$.
  create and sort $C_u,\ u \in V$
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*) storing 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

5.3.6 Time and memory requirements for aCL-II

Steps 1, 4 and 5 of the above algorithm copy the corresponding steps of **aCL-I**. Hence we analyze only steps 2 and 3.

Constructing $\overline{V}_0$ has been discussed previously and found to take $\mathcal{O}(s^2 N \log \frac{N}{n})$ time and $\mathcal{O}(s^2N)$ memory. Then we need to compute no more than $\mathcal{O}(s^2N)$ mutual informations and to sort the resulting lists. The worst case for the latter is when all the lists are equal. Then, the total time is bounded by

$$n(\frac{s^2N}{n} \log \frac{s^2N}{n}) \;=\; s^2N \log \frac{s^2N}{n}$$

a result encountered before.

Constructing *vheap* takes $\mathcal{O}(n)$ units of time and memory. Each extraction from it 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. n_K is again the number of steps taken by Kruskal's algorithm. Therefore, the running time of the **aCL-II** algorithm is

$$\mathcal{O}(n \log n + sn + s^2 N \log(s^2 nN) + n_K \log n) \tag{5.24}$$

The additional memory requirement are bounded by

$$\mathcal{O}(n + s^2N). \tag{5.25}$$

Comparing (5.24) above to (5.20) it appears that it is not possible to determine which algorithm is asymptotically faster from the bounds alone. The algorithms have large parts that are identical and the factor $\log s^2 nN$ is a loose overestimate in both formulas; thus, it may be that the running times of the two algorithms are close in reality. But more important is the fact that by the **aCL-II** algorithm we have managed to relax the one of the assumptions made in the previous section: the latter algorithm does not rely on integer counts. This difference will become essential in the context of the EM algorithm.

5.4 Generalization to discrete variables of arbitrary arity

This section will show how the **TreeLearn** algorithm can be accelerated in discrete domains where the variables can take more than two values. The algorithm that we are going to develop is a simple extension of the **aCL-II** algorithm. Therefore, we shall present only the basic ideas of the extension and the modifications to the **aCL-II** algorithm that they imply.

The **aCL** 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

$$|x| \;=\; n - \sum_{v \in V} \delta_{x_v} \tag{5.26}$$

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 a way to presort mutual informations for non-cooccurring variables similar to that used by the **aCL-II** algorithm.

5.4.1 Computing cooccurrences

Following the previously introduced idea of not representing explicitly 0 values, 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 also 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.

5.4.2 Presorting mutual informations

As in subsection 5.3.1, 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:

$$P_v(i) \;=\; \frac{N_v^i}{N}, \quad i \neq 0 \tag{5.27}$$

$$P_{v0} \;\equiv\; P_v(0) \;=\; 1 - \sum_{i \neq 0} P_v(i) \tag{5.28}$$

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:

$$\begin{aligned}
H_v &= -P_{v0} \log P_{v0} - \sum_{i \neq 0} P_v(i) \log P_v(i) \\
&= -P_{v0} \log P_{v0} - (1 - P_{v0}) \sum_{i \neq 0} \frac{P_v(i)}{(1 - P_{v0})} \left[\log \frac{P_v(i)}{(1 - P_{v0})} + \log(1 - P_{v0}) \right] \\
&= \underbrace{-P_{v0} \log P_{v0} - (1 - P_{v0}) \log(1 - P_{v0})}_{H_{v0}} - (1 - P_{v0}) \underbrace{\sum_{i \neq 0} \frac{P_v(i)}{(1 - P_{v0})} \log \frac{P_v(i)}{(1 - P_{v0})}}_{-H_{\overline{v}}} \\
&= H_{v0} + (1 - P_{v0}) H_{\overline{v}}
\end{aligned} \tag{5.29}$$

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.

$$I_{uv} \;=\; H_u - H_{u|v} \tag{5.30}$$

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

$$H_{u|v} \;=\; P_{v0} H_{u|v=0} + \sum_{j \neq 0} P_v(j) \underbrace{H_{u|v=j}}_{0} \tag{5.31}$$

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 (5.29).

$$H_{u|v=0} \;=\; H_{u0|v=0} + (1 - P_{u=0|v=0}) H_{u|u \neq 0, v=0} \tag{5.32}$$

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

$$H_{u|u \neq 0, v=0} \;=\; H_{\overline{u}} \tag{5.33}$$

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

$$\begin{aligned} Pr[u = 0 | v = 0] &= 1 - \sum_{i \neq 0} P_{u|v=0}(i) \\ &= 1 - \sum_{i \neq 0} \frac{P_u(i)}{P_{v0}} \\ &= 1 - \frac{1 - P_{u0}}{1 - P_{v0}} \end{aligned} \tag{5.34}$$

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

$$I_{uv} \;=\; P_{u0} \log P_{u0} - P_{v0} \log P_{v0} + (P_{u0} + P_{v0} - 1) \log(P_{u0} + P_{v0} - 1) \tag{5.35}$$

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

$$\frac{\partial I_{uv}}{\partial P_{v0}} \;=\; \log \frac{P_{v0} + P_{u0} - 1}{P_{v0}} \;<\; 0 \tag{5.36}$$

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 (5.12) in the case $N_{uv} = 0$.

We have shown that the **aCL-II** 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}, $i, j \neq 0$.

5.5 Using the aCL algorithms 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 $N,\ N_v,\ N_{uv},\ u, v \in V$. 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, and in the case of learning mixtures by the EM algorithm the counts defining P for each of the component trees are not integer.

Recall that each E step of the EM algorithm computes the posterior probability of each mixture component k of having generated data point x^i. This is $\gamma_k(i)$ defined in equation (3.15). 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 k trees in the mixture. The counts N_v^k and N_{uv}^k corresponding to tree k are defined in terms of the γ values as

$$N^k \equiv \Gamma_k = \sum_i \gamma_k(i) \tag{5.37}$$

$$N_v^k = \sum_{i:x_v^i=1} \gamma_k(i) \tag{5.38}$$

$$N_{uv}^k = \sum_{i:x_v^i=1 \wedge x_u^i=1} \gamma_k(i). \tag{5.39}$$

These counts are in general not integer numbers. Therefore, for learning mixtures of trees the **aCL-II** algorithm is recommended.

Now we shall examine steps 1 and 2 of the **aCL-II** algorithm and show how to modify them in order to handle weighted data.

First, remark that 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.

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 $\mathcal{O}(mnN)$.

5.6 Decomposable priors and the aCL 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 aCL algorithm. The classes of priors that we shall be concerned with are the priors discussed in chapter 4. We shall first examine priors on the tree's structure having the form

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

As shown in section 4.2.1, 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 **aCL** algorithms. 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 **aCL** 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 $N^k \equiv \Gamma_k$ 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 has already been shown 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

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

with $\tilde{P}$ a mixture between the empirical distribution P and the fictitious distribution P' defined by N'_{uv}. This challenges two of the assumptions that the accelerated algorithms are based upon. First, the counts N_v, N_{uv} cease to be integers. This affects only the **aCL-I** algorithm.

Second, both algorithms rely on the fact that most of the N_{uv} values are 0. If the counts N'_{uv} violate this assumption then the **aCL** algorithms become inefficient. In particular, the **aCL-II** algorithm degrades to a standard **TreeLearn** 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 aCL algorithms will in general not be compatible.

Somehow suprisingly, the uniform prior (4.35) 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 **aCL-II** algorithm efficiently after only slight modification.

5.7 Experiments

The following experiments compare the (hypothesized) gain in speed of the accelerated **TreeLearn** algorithm w.r.t the traditional version presented in chapter 2 under controlled conditions on artificial data.

The binary domain had a dimensionality n varying from 50 to 1000. Each data point had a fixed number s of variables being on. The sparsity s took the values 5, 10, 15

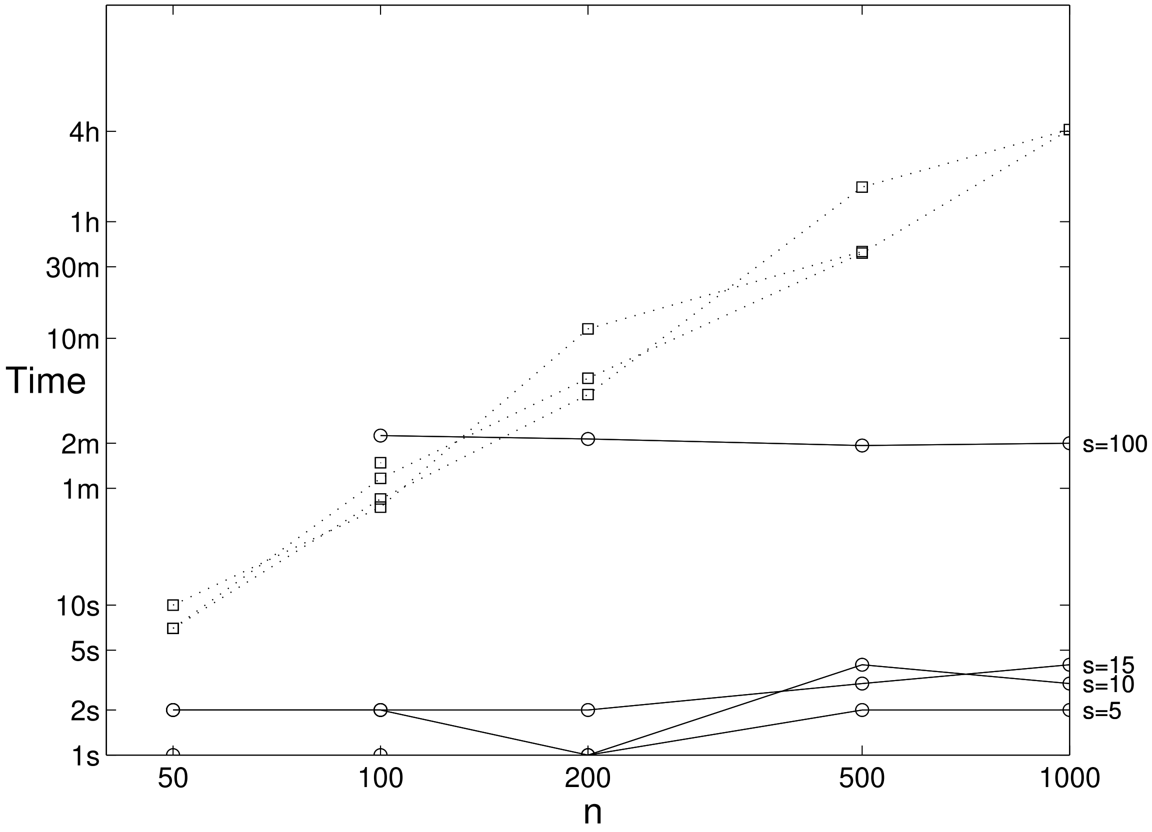

Figure 5-4: Real running time for the accelerated (full line) and traditional (dotted line) **TreeLearn** algorithm versus number of vertices n for different values of the sparsity s.

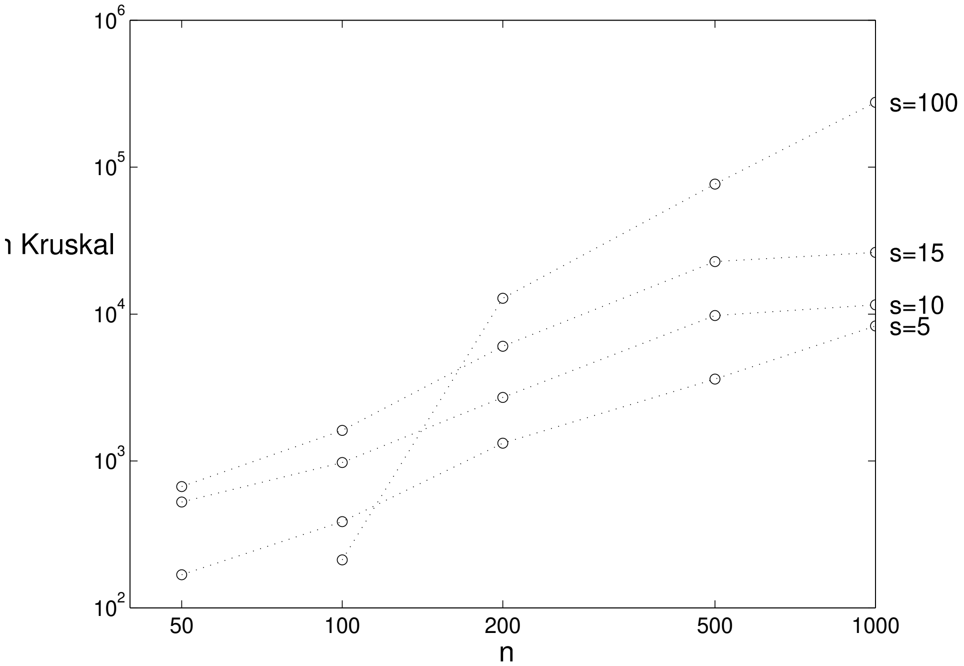

Figure 5-5: Number of steps of the Kruskal algorithm n_K versus domain size n measured for the **aCL-II** algorithm for different values of s

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 **TreeLearn** and **aCL-II** to fit one tree distribution and the running times were recorded and plotted in figure 5-4. 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 **aCL-II** algorithm takes 2 minutes to complete, improving on the traditional algorithm by a factor of “only” 123.

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 5-5) 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 5-4 also confirms that the running time of the **TreeLearn** algorithm grows quadratically with n and is independent of s.

5.8 Concluding remarks

This chapter has presented a way of taking advantages of sparsity in the data to accelerate the tree learning algorithm. The ideas introduced have been developed into the two algorithms presented without exhausting the number of possible variants.

The methods achieve their performance by taking advantage of 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. Here the algorithms have been evaluated considering that the data sparsity s is a constant. If this is not the case, however, the actual complexity of the algorithms can be computed easily from the bounds provided here, since the dependence on s is made explicit in each of them.

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.

The **aCL-II** algorithm relies heavily on the fact that most cooccurrences between variables are zero. Given this fact, it can smoothly handle real values for the non-zero cooccurrence counts and even some classes of priors. The **aCL-I** algorithm is more restrictive in the sense that it requires integer counts and from the present perspective, it is at clear disadvantage w.r.t **aCL-II** both in simplicity and in versatility.

5.9 Appendix: Bounding the number of lists N_L

Let us denote the total number of nonempty lists $V_c^{\pm}(u), c > 0, u \in V$ by N_L. It is obvious N_L is maximized when all list lengths are 1 for $c \leq C$, 0 for $c > C+1$ and 0 or 1 for $c = C$, where C is a constant to be determined. In this case, the number of lists can be expressed successively as

$$\begin{aligned} N_L &= n\sum_{c=1}^{C} 2c + \mathcal{O}(n)(C+1) \\ &= n(C+C^2) + \mathcal{O}(n)(C+1) \end{aligned}$$

Therefore

$$n(C+C^2) \;\leq\; N_C \;\leq\; s^2N/2 \tag{5.40}$$

Solving for C we obtain

$$C \;\leq\; -\frac{1}{2} + \sqrt{\frac{s^2N}{2n} + 1} \tag{5.41}$$

which amounts to

$$\begin{aligned} N_L &= nC + \mathcal{O}(n) \\ &= \mathcal{O}\left(sn\sqrt{\frac{N}{n}}\sqrt{\frac{1}{2} + \frac{n}{s^2N}}\right) \end{aligned} \tag{5.42}$$

Under the assumption that $\frac{N}{n}$ is bounded above and below the equation simplifies to

$$N_L \;=\; \mathcal{O}(sn) \tag{5.43}$$