The VLDB Journal manuscript No.
(will be inserted by the editor)

Dan Pelleg · Andrew Moore

Dependency Trees in Sub-linear Time and Bounded Memory

Received: date / Accepted: date

Abstract We focus on the problem of efficient learning of dependency trees. Once grown, they can be used as a special case of a Bayesian network, for PDF approximation, and for many other uses. Given the data, a well-known algorithm can fit an optimal tree in time that is quadratic in the number of attributes and linear in the number of records. We show how to modify it to exploit partial knowledge about edge weights. Experimental results show running time that is near-constant in the number of records, without significant loss in accuracy of the generated trees.

Keywords Data Mining · Probably Approximately Correct Learning · Fast Algorithms · Dependency Trees

1 Introduction

Bayesian networks are a popular class of very general models. They are widely used for data modeling, for inference, and for PDF approximation. They are also appealing from the cognitive aspect as their structure can often be visualized and easily understood. However, because of their expressiveness, they are hard to fit from data, requiring search in a super-exponential space of possible graph structures. Despite recent advances (Friedman et al., 1999; Goldenberg & Moore, 2004), learning network structure from big data sets demands huge computational resources.

Our approach restricts the search space to a more tractable one by considering only a simpler sub-class of graphical models. Specifically, we focus on trees. For trees, the well-known Chow and Liu (1968) algorithm can find optimal solutions in polynomial time. As an added feature, the trees can be described more simply to human users. Below, we show how to modify the known algorithm so it runs in time that is sub-linear in the input size, using a user-specified amount of memory. Empirical evidence shows run time which is linear in the number of attributes, and constant in the input size. The constant depends only on intrinsic properties of the data. This allows processing of very large data sets. We also examine and quantify the possible loss of accuracy and show it is negligible for most practical purposes.

More precisely, dependency trees are belief networks that satisfy the additional constraint that each node has at most one parent. It has been shown (Chow & Liu, 1968) that finding the tree that maximizes the data likelihood can be performed as follows. First, construct a full graph where each node corresponds to an attribute in the input data. Next, assign edge weights; these are derived from the mutual information values of the corresponding attribute pairs. Finally, run a minimum[1] spanning tree algorithm on the weighted graph. The output tree is the desired one.

Besides being a "lighter" version of Bayesian networks, dependency trees are also interesting in their own right. They form a complete representation (Meila, 1999b). Additionally, they can act as initializers for search, as mixture components (Meila, 1999b), or as components in classifiers (Friedman et al., 1998).

Once the weight matrix is constructed, executing a minimum spanning tree (MST) algorithm is fast. The time-consuming part is the population of the weight matrix, which takes time quadratic in the number of attributes and linear in the number of records. This becomes expensive when considering datasets with hundreds of thousands of records and hundreds of attributes.

To overcome this problem, we propose a new way of interleaving the spanning tree construction with the operations needed to compute the mutual information coefficients. We develop a new spanning-tree algorithm, based solely on Tarjan's (1983) red-edge rule. This algorithm is capable of using partial knowledge about edge weights and of signaling

Work done at Carnegie-Mellon university. This research was sponsored by the National Science Foundation (NSF) under grant no. ACI-0121671 and no. DMS-9873442.

Dan Pelleg
IBM Haifa Labs, Haifa, Israel. E-mail: dpelleg@cs.cmu.edu

Andrew Moore
Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA.

[1] To be precise, we will use it as a *maximum* spanning tree algorithm. The two are interchangeable, requiring just a reversal of the edge weight comparison operator. Historically, minimum has been far more popular a name.

the need for more accurate information regarding a particular edge. The partial information we maintain is in the form of probabilistic confidence intervals on the edge weights; an interval is derived by looking at a sub-sample of the data for a particular attribute pair. Whenever the algorithm signals that a currently-known interval is too wide, we inspect more data records in order to shrink it. Once the interval is small enough, we may be able to prove that the corresponding edge is *not* a part of the tree. Whenever such an edge can be eliminated without looking at the full data set, the work associated with the remainder of the data is saved. This is where performance is gained.

We have implemented the algorithm for numeric and categorical data and tested it on real and synthetic data sets containing hundreds of attributes and millions of records. We show experimental results of up to 5,000-fold speed improvements over the traditional algorithm. The resulting trees are, in most cases, of near identical quality to the ones grown by the naive algorithm.

Use of probabilistic bounds to direct structure-search appears in Maron and Moore (1994) for classification and in Moore and Lee (1994) for model selection. In a sequence of papers, Domingos et al. have demonstrated the usefulness of this technique for decision trees (Domingos & Hulten, 2000), K-means clustering (Domingos & Hulten, 2001a), and EM for mixtures of Gaussians (Domingos & Hulten, 2001b). In the context of dependency trees, Meila (1999a) discusses the discrete case that frequently comes up in text-mining applications, where the attributes are sparse in the sense that only a small fraction of them are true for any record. In this case it is possible to exploit the sparseness and accelerate the Chow-Liu algorithm.

Throughout the chapter we use the following notation. The number of data records is R, the number of attributes M. When x is an attribute, x_i is the value it takes for the i-th record. We denote by ρ_{xy} the correlation coefficient between attributes x and y, and omit the subscript when it is clear from the context. H_x is the entropy of an attribute or an attribute set x.

2 A Slow Minimum-Spanning Tree Algorithm

We begin by describing our MST algorithm. Although in its given form it can be applied to any graph, it is asymptotically slower than established algorithms (as predicted in Tarjan (1983) for all algorithms in its class). We then proceed to describe its use in the case where some edge weights are known not exactly, but rather only to lie within a given interval. In Section 4 we will show how this property of the algorithm interacts with the data-scanning step to produce an efficient dependency-tree algorithm.

In the following discussion we assume we are given a complete graph with n nodes, and the task is to find a tree connecting all of its nodes such that the total tree weight (defined to be the sum of the weights of its edges) is minimized. This problem has been extremely well studied and numerous efficient algorithms for it exist.

1. $T \leftarrow$ an arbitrary spanning set of $n-1$ edges.
   $L \leftarrow$ empty set.
2. While $|\bar{L}| > n-1$ do:
   - Pick an arbitrary edge $e \in \bar{L} \setminus T$.
   - Let e' be the heaviest edge on the path in T between the endpoints of e.
   - If e is heavier than e':
     - $L \leftarrow L \cup \{e\}$
   - otherwise:
     - $T \leftarrow T \cup \{e\} \setminus \{e'\}$
     - $L \leftarrow L \cup \{e'\}$
3. Output T.

Fig. 1 The MIST algorithm. At each step of the iteration, T contains the current "draft" tree. L contains the set of edges that have been proven to *not* be in the MST and so $\bar{L}$ contains the set of edges that still have some chance of being in the MST. T never contains an edge in L.

We start with a rule to *eliminate* edges from consideration for the output tree. Following Tarjan (1983), we state the so-called "red-edge" rule:

Theorem 1 *The heaviest edge in any cycle in the graph is not part of the minimum spanning tree.*

Traditionally, MST algorithms use this rule in conjunction with a greedy "blue-edge" rule, which chooses edges for *inclusion* in the tree. In contrast, we will repeatedly use the red-edge rule until all but $n-1$ edges have been eliminated. The proof that this results in a minimum-spanning tree follows from Tarjan (1983).

Let E be the original set of edges. Denote by L the set of edges that have already been eliminated, and let $\bar{L} = E \setminus L$. As a way to guide our search for edges to eliminate we maintain the following invariant:

Invariant 1: At any point there is a spanning tree T, which is composed of edges in $\bar{L}$.

In each step, we arbitrarily choose some edge e in $\bar{L} \setminus T$ and try to eliminate it using the red-edge rule. Recall that the rule needs a cycle to act on. Let P be the path in T between the endpoints of e. The cycle we will apply the red-edge rule to will be composed of e and P. It is clear we only need to compare e with the heaviest edge in P. If e is heavier, we can eliminate it by the red-edge rule. However, if it is lighter, then we can eliminate the tree edge by the same rule. If this is indeed the case, we do so and add e to the tree to preserve Invariant 1. The algorithm, which we call Minimum Incremental Spanning Tree (MIST), is listed in Figure 1. Figures 2-5 illustrate how it may run on an example graph.

The MIST algorithm can be applied directly to a graph where the edge weights are known exactly. And like many other MST algorithms, it can also be used in the case where

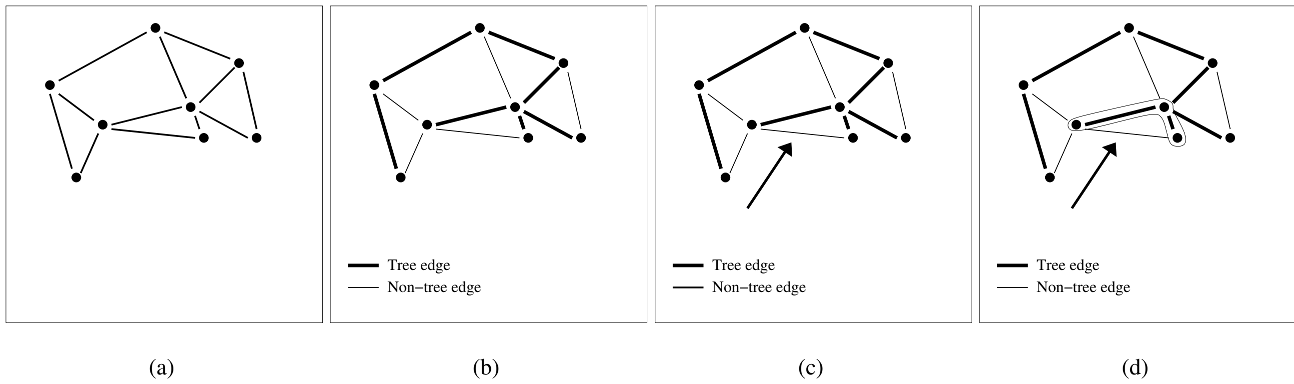


Fig. 2 Walkthrough of the MIST algorithm. The original graph (a). An arbitrary spanning tree is chosen (b). An arbitrary edge is chosen for elimination (c). The tree path completes the edge to a cycle (d). (Continued)

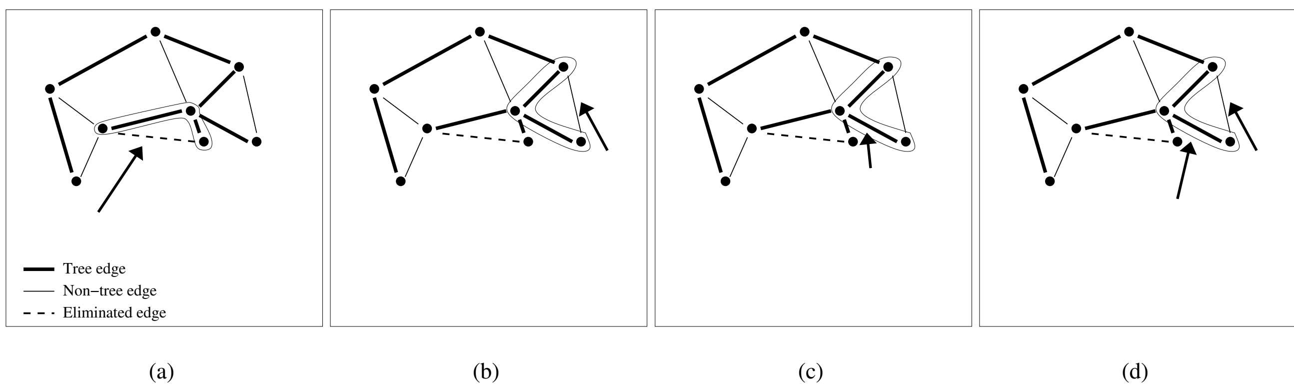


Fig. 3 Walkthrough of the MIST algorithm (cont'd). The edge is discovered to be the heaviest on the cycle and eliminated (a). Another edge is chosen (b). On completing the cycle, some other edge in it is discovered to be heaviest (c). The tree edge is eliminated, and the non-tree edge swapped in (d). (Continued)

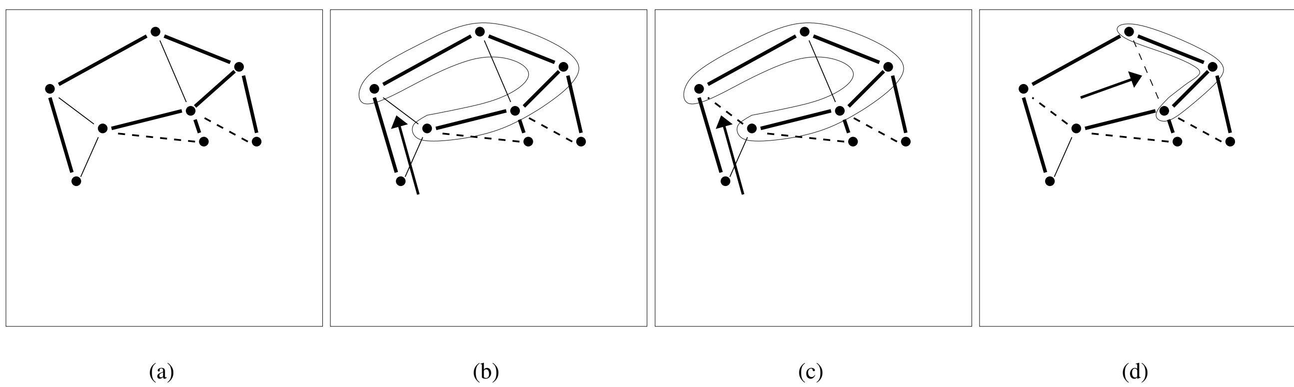
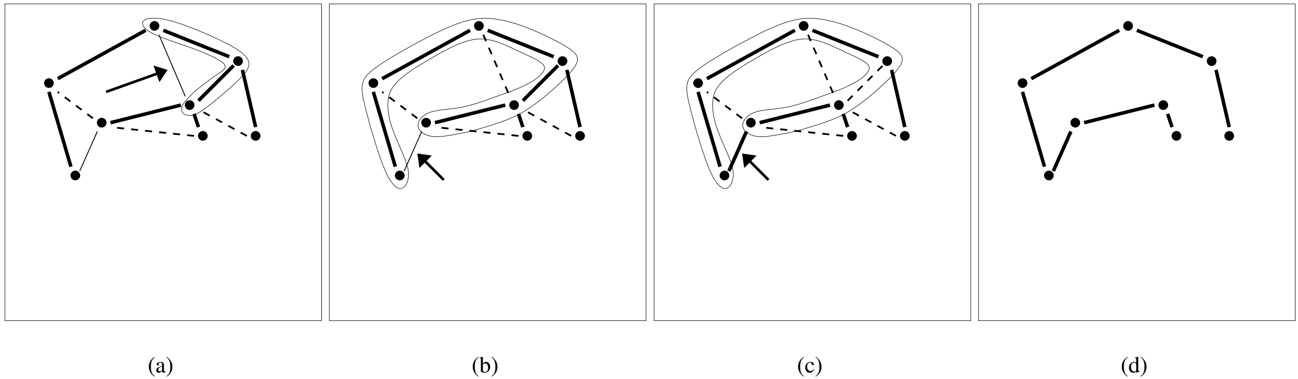


Fig. 4 Walkthrough of the MIST algorithm (cont'd). The updated tree (a). Another edge is chosen (b). The tree cycle is completed and the non-tree edge eliminated (c). The next edge is chosen (d). (Continued)



(a) (b) (c) (d)

Fig. 5 Walkthrough of the MIST algorithm (cont'd). The edge is eliminated (a). The last remaining non-tree edge is chosen (b) and swapped in (c). The output tree (d).

just the relative order of the edge weights (as opposed to the exact value) is given. Now imagine a different setting, where edge weights are not given, and instead an oracle exists, which knows the exact values of the edge weights. When asked about the relative order of two edges, it may either respond with the correct answer, or it may give an inconclusive answer. Furthermore, a constant fee is charged for each query. In this setup, MIST is still suited for finding a spanning tree while minimizing the number of queries issued. In step 2, we go to the oracle to determine the order. If the answer is conclusive, the algorithm proceeds as described. Otherwise, it just ignores the "if" clause altogether and iterates (possibly with a different edge e).

For the moment, this setting may seem contrived, but in Section 4, we go back to the MIST algorithm and put it in a context very similar to the one described here.

3 Probabilistic Bounds on Mutual Information

We now concentrate once again on the specific problem of determining the mutual information between a pair of attributes. We show how to compute it given the complete data, and how to derive probabilistic confidence intervals for it, given just a sample of the data.

As shown in Reza (1994), the mutual information for two jointly Gaussian[2] numeric attributes X and Y is:

$$I(X;Y) = -\frac{1}{2}\ln(1-\rho^2)$$

where the correlation coefficient $\rho = \rho_{XY} =$

$$\frac{\sum_{i=1}^{R}\left((x_i-\bar{x})(y_i-\bar{y})\right)}{\hat{\sigma}_X^2\hat{\sigma}_Y^2}$$

[2] If the data is not Gaussian, we can make no claims. However, there are several possible extensions to the dependency tree model which are flexible enough to accommodate such data (Pelleg, 2004; Davies, 2002).

with $\bar{x}, \bar{y}, \hat{\sigma}_X^2$ and $\hat{\sigma}_Y^2$ being the sample means and variances for attributes X and Y. In practice, we standardize the data in a pre-processing step to have zero mean and unit variance. This leaves $x_i \cdot y_i$ as the only unknown.

Since the log function is monotonic, $I(X;Y)$ is also monotonic in $|\rho|$. This is a sufficient condition for the use of $|\rho|$ as the edge weight in a MST algorithm. Consequently, the sample correlation can be used in a straightforward manner when the complete data is available. Now consider the case where just a sample of the data has been observed.

Let x and y be two data attributes. We are trying to estimate $\sum_{i=1}^{R} x_i \cdot y_i$ given the partial sum $\sum_{i=1}^{r} x_i \cdot y_i$ for some $r < R$. To derive a confidence interval, we use the Central Limit Theorem.[3] It states that given samples of the random variable Z (where for our purposes $Z_i = x_i \cdot y_i$), the sum $\sum_i Z_i$ can be approximated by a Normal distribution with mean and variance closely related to the distribution mean and variance. Furthermore, for large samples, the sample mean and variance can be substituted for the unknown distribution parameters. Note, in particular, that the central limit theorem *does not require us to make any assumption about the Gaussianity of* Z. We thus can derive a two-sided confidence interval for $\sum_i Z_i = \sum_i x_i \cdot y_i$ with probability $1-\delta$ for some user-specified δ, typically 1%. Given this interval, computing an interval for ρ is straightforward.

In the case of binary categorical data, we follow Meila (1999b) and write:

$$\begin{aligned}I(X;Y) &= H_X + H_Y - H_{XY} \\ &= \frac{1}{R}[-\mathrm{zlogz}(N_X) - \mathrm{zlogz}(N-N_X) \\ &- \mathrm{zlogz}(N_Y) - \mathrm{zlogz}(N-N_Y) \\ &+ \mathrm{zlogz}(N_{XY}) + \mathrm{zlogz}(N_X-N_{XY}) \\ &+ \mathrm{zlogz}(N_Y-N_{XY}) \\ &+ \mathrm{zlogz}(R-N_X-N_Y+N_{XY}) \\ &+ \mathrm{zlogz}(R)] \qquad (1)\end{aligned}$$

[3] One can use the weaker Hoeffding bound instead, and our implementation supports it as well, although it is generally much less useful.

where $\text{zlogz}(z)$ is shorthand for $z \log z$, and N_z denotes the number of times an attribute or a set of attributes are observed all true. As before, N_{XY} is the quantity we are deriving a probabilistic estimate for, which we do from the counts in a sample, and application of the CLT.

Now, observe that:

$$\begin{aligned}\frac{d(zlogz(y))}{dx} &= \frac{dy\log y}{dx} \\ &= \frac{dy}{dx}\log y + y\cdot\frac{1}{y}\cdot dy/dx \\ &= (1+\log y)\frac{dy}{dx}\,.\end{aligned}$$

We take a derivative of $I(X;Y)$ with respect to the measured quantity N_{XY}. The only terms in Equation 1 which do not cancel out are the ones containing N_{XY}:

$$\begin{aligned}\frac{d(I(X;Y))}{dN_{xy}} &= (1+\log N_{xy}) - (1+\log(N_x - N_{xy})) \\ &\quad -(1+\log(N_y - N_{xy})) \\ &\quad +(1+\log(R - N_x - N_y + N_{xy})) \\ &= \log N_{xy} - \log(N_x - N_{xy}) \\ &\quad -\log(N_y - N_{xy}) \\ &\quad +\log(R - N_x - N_y + N_{xy}) = \\ &= \log\frac{N_{xy}(R - N_x + N_y + N_{xy})}{(N_x - N_{xy})(N_y - N_{xy})}\,. \end{aligned} \tag{2}$$

Let $N_{\bar{x}\bar{y}}$ be the number of records for which both attributes were false, and similarly for $N_{x\bar{y}}$ and $N_{\bar{x}y}$. Immediately we get:

$$\begin{aligned}N_{x\bar{y}} &= N_x - N_{xy} \\ N_{\bar{x}y} &= N_y - N_{xy} \\ N_{\bar{x}\bar{y}} &= R - N_x + N_y + N_{xy}\,.\end{aligned}$$

Then, equality with zero in Equation 2 above is obtained when:

$$N_{xy}N_{\bar{x}\bar{y}} = N_{x\bar{y}}N_{\bar{x}y}$$

or:

$$N_{xy} = \frac{N_{x\bar{y}}N_{\bar{x}y}}{N_{\bar{x}\bar{y}}} \tag{3}$$

Therefore, to determine minimum and maximum values for $I(X;Y)$ at the interval, we evaluate it at the endpoints. Additionally, we evaluate at the extreme point, if it happens to be included in the interval. To summarize, we derive the upper (resp. lower) bounds by taking the maximum (resp. minimum) over the set containing the endpoints derived from the CLT, and optionally the extreme point from Equation 3.

4 The Full Algorithm

As we argued, the MIST algorithm is capable of using partial information about edge weights. We have also shown how to derive confidence intervals on edge weights. We now combine the two and give an efficient dependency-tree algorithm.

We largely follow the MIST algorithm as listed in Figure 1. We initialize the tree T in the following heuristic way: first we take a small sub-sample of the data, and derive point estimates for the edge weights from it. Then feed the point estimates to any MST algorithm and obtain a "draft" tree T.

When we come to compare edge weights, we generally need to deal with two intervals. If they do not intersect, then the points in one of them are all smaller in value than any point in the other, in which case we can determine which represents a heavier edge. We apply this logic to all comparisons, where the goal is to determine the heaviest path edge e' and to compare it to the candidate e. If we are lucky enough that all of these comparisons are conclusive, then the amount of work we save is related to how much data was used in computing the confidence intervals — the rest of the data for the attribute-pair that is represented by the eliminated edge can be ignored.

However, there is no guarantee that the intervals are separated and allow us to draw meaningful conclusions. If they do not, then we have a situation similar to the inconclusive oracle answers in Section 2. The price we need to pay here is looking at more data to shrink the confidence intervals. We do this by choosing one edge — either a tree-path edge or the candidate edge — for "promotion", and increasing the sample size used to compute the sufficient statistics for it[4]. After doing so we try to eliminate again (since we can do this at no additional cost). If we fail to eliminate we iterate, possibly choosing a different candidate edge (and the corresponding tree path) this time.

The choice of which edge to promote is heuristic, and depends on the expected success of resolution once the interval has shrunk. This is estimated by first defining a cost measure for a set of tree edges and a candidate edge. It is the sum of the sizes of intersections of the tree edges with the candidate edge, plus the size of intersections between the worst tree edge (as defined by the mid-points of the intervals) and the other tree edges. We now go over the tree edges, in turn, and for each one estimate the expected size of its interval, if given more data. This estimate depends on the measured variance in the observed data. We record the cost for each of these speculative edges. The one associated with the lowest cost is chosen, unless the expected difference from the current cost is below a threshold. If this holds, we pick an edge at random from the set of edges that define the boundary of the union of the tree edges which intersects with the candidate edge. If this is impossible (for example, all of these edges are already saturated), we choose some tree-path edge at random.

[4] Our implementation doubles the sample, up to a limit of 64 times the original size.

Another heuristic we employ goes as follows. Consider the comparison of the path-heaviest edge to an estimate of a candidate edge. The interval for the candidate edge may be very small, and yet still intersect the interval that is the heavy edge's weight (this would happen if, for example, both attribute-pairs have the same distribution). We may be able to reduce the amount of work by pretending the interval is narrower than it really is. We therefore trim the interval by a constant, parameterized by the user as ε, before performing the comparison. This use of δ and ε is analogous to their use in "Probably Approximately Correct" analysis: on each decision, with high probability $(1-\delta)$ we will make at worst a small mistake (ε).

Above, we stated that we can examine more data for any given edge at will. The tacit assumption is that the data is stored on random-access media. But in practice, many interesting data sets cannot fit in RAM. We return to this point below, and show how to implement the algorithm so it uses only as much memory as the user specifies.

4.1 Algorithm Complexity

We now discuss the theoretical complexity of the proposed algorithm. Refer to Figure 1. In theory, the first step can be done by choosing edges at random. In practice, it is built by sampling some number S of records from the input, and running a Chow-Liu algorithm on the sample. The complexity of obtaining the sample is $O(SM^2)$. If M is very large then this can dominate the run time. A possible countermeasure is to choose S proportional to M^{-2}. However this is not always possible: if M^2 is in the order of R or greater, this will result in a sample size smaller than one. Therefore the worst case time here is $O(M^2)$. Finding the actual minimum spanning tree on the sample can be done in time $O(M^2 + M \log M)$ by Prim's algorithm using Fibonacci heaps (Cormen et al., 1989).

Step 2 in Figure 1 requires $O(M^2)$ successful elimination steps. Each step requires, aside from the work required to read more data, finding a tree path between two nodes. In the worst case, this can take $O(M)$ work since the current tree contains $M-1$ edges. Therefore the cost for this step is $O(M^3)$. It is possible that the cost of finding and updating tree paths can be amortized (Tarjan, 1983). But more importantly, elimination of tree edges is a rare occurrence, so the tree structure is generally static. Therefore tree paths can be recorded and re-used instead of discovered. This is done in the current implementation.

Below, we present empirical results showing that, for the data sets in question, the worst case is an overestimate. In particular, Figure 7 shows that for synthetic sets with $M < 160$, performance is still comfortably in the linear range.

4.2 Bounding Memory Usage

One advantage the original Chow and Liu algorithm has is that it is easily executed with a single sequential scan of the data file. Conversely, the algorithm as described above assumes it is easy to examine any given datum, and might process records out of order. Suppose, for example, that an edge between two specific attributes is promoted several times, and the first $10,000$ records for it are read. Now, it is still possible that another edge, containing just one of these attributes, will be promoted in a future step, and the range of records for this promotion will be, say, $1000-2000$. That is, the algorithm may need to revisit some data cells, and will generally do this in an unpredictable way.

This kind of access pattern is generally expensive to do if the data is on disk. Storing the whole set in RAM might solve the problem, but that is not always feasible. Below, we describe an access scheme that allows the data to reside in a standard SQL database, and only retrieves data in contiguous blocks of a single attribute. We have fully implemented the ideas below, and the results in Section 5.2 were obtained by running our implementation. Under reasonably weak assumptions, access to the data through a database is fast. Additionally, we make use of a local cache of data, and let the user specify its size. The management policy for the cache takes into account algorithm-specific information that is not generally available to the database.

Our scheme can be thought of as an access layer through which the described algorithm reads data. Whenever more data for a given edge is required, the task of the access layer is to provide two contiguous blocks, one from each of the corresponding data columns (attributes).

The local cache is divided into buffers of equal size. Each contains a contiguous block of data from a given column. Note that these may be smaller than the amount requested for examination. In this case each request will require data from multiple buffers for each of the two columns. If the requested data is in the cache, we simply transfer it to the main algorithm. Otherwise, we locate an appropriate buffer to store the data, and make a database request for it. The database request should be handled efficiently by any SQL implementation. In practice, this is true if the buffers are big enough and the data is indexed appropriately.

As with any cache, the case where all buffers are in use is handled by an eviction policy. We normally evict the buffer that has data which was accessed least recently. However, we can optimize further. Note that given a particular edge, data used for the edge is read in-order. In other words, each data cell will only be accessed at most once per edge. Therefore, for a specific cell and its associated column, there will be at most $(M-1)$ accesses to it — one for each of the other columns. Our cache exploits this fact by spontaneously evicting buffers that have already been accessed $(M-1)$ times.

Below, we present empirical results exploring different aspects of the caching scheme. In essence, they prove that a fixed-size buffer cache provides a way to handle arbitrary amounts of data, if it is stored appropriately in a standard database. As an anecdote, we created a dependency-tree for SDSS data consisting of 446 attributes and $200,000$ records, or around 700MB of raw data, using a buffer cache of 48

MB. It took 19 minutes to complete on a 2.8 Ghz dual Intel xeon which also hosted the database.

5 Experimental Results

In the following description of experiments, we vary different parameters for the data and the algorithm. Unless otherwise specified, these are the default values for the parameters. We set δ to 1% and ε to 0.05 (on either side of the interval, totaling 0.1). The initial sample size is fifty records. There are $100,000$ records and 100 attributes. The data is real-valued. The data-generation process first generates a random tree, then draws points for each node from a normal distribution with the node's parent's value as the mean. In addition, any data value is set to random noise with probability 0.15.

To construct the correlation matrix from the full data, each of the R records needs to be considered for each of the $\binom{M}{2}$ attribute pairs. We evaluate the performance of our algorithm by adding the number of records that were actually scanned for all the attribute-pairs, and dividing the total by $R\binom{M}{2}$. We call this number the "data usage" of our algorithm. The closer it is to zero, the more efficient our sampling is, while a value of one means the same amount of work as for the full-data algorithm (possibly more, when considering the overhead).

We first demonstrate the speed of our algorithm as compared with the full $O(RM^2)$ scan. Figure 6 shows that the amount of data the algorithm examines is a constant that does not depend on the size of the data set. This translates to relative run-times of 0.7% (for the $37,500$-record set) to 0.02% (for the $1,200,000$-record set) as compared with the full-data algorithm. The latter number translates to a $5,000$-fold speedup. Note that the reported usage is an average over the number of attributes. However, this does not mean that the same amount of data was inspected for every attribute-pair — the algorithm determines how much effort to invest in each edge separately. We return to this point below.

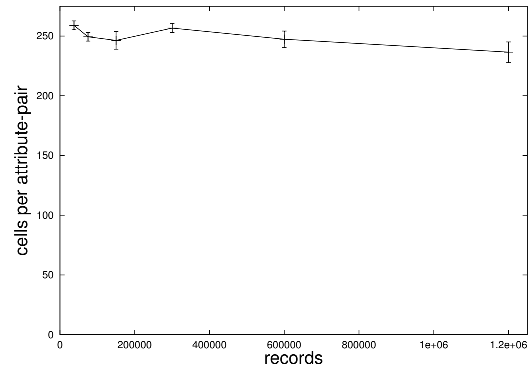


Fig. 6 Amount of data read (indicative of absolute running time), in attribute-pair units per attribute.

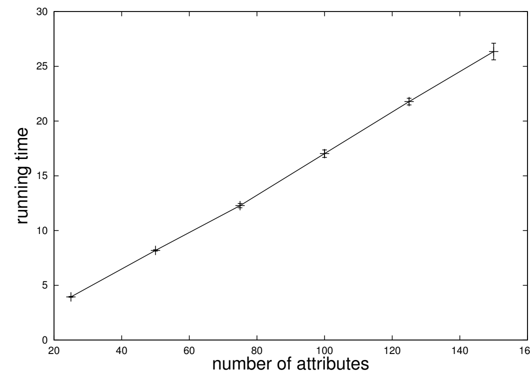


Fig. 7 Running time as a function of the number of attributes.

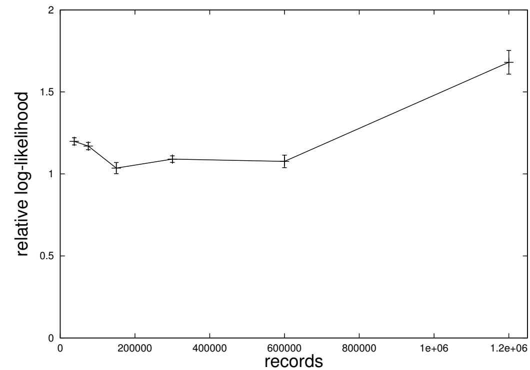


Fig. 8 Relative log-likelihood vs. the sample-based algorithm. The log-likelihood difference is divided by the number of records.

The running time is plotted against the number of data attributes in Figure 7. A linear relation is clearly seen, meaning that (at least for this particular data generation scheme) the algorithm is successful in doing work that is proportional to the number of tree edges.

Clearly speed has to be traded off. For our algorithm the risk is making the wrong decision about which edges to include in the resulting tree. For many applications this is an acceptable risk. However, there might be a simpler way to grow estimate-based dependency trees, one that does not involve complex red-edge rules. In particular, we can just run the original algorithm on a small sample of the data, and use the generated tree. It would certainly be fast, and the only question is how well it performs.

To examine this effect we have generated data as above, then ran a 30-fold cross-validation test for the trees our algorithm generated. We also ran a sample-based algorithm on each of the folds. This variant behaves just like the full-data algorithm, but instead examines just the fraction of it that adds up to the total amount of data used by our algorithm. Results for multiple data sets are in Figure 8. We see that our algorithm outperforms the sample-based algorithm, even though they are both using the same total amount of data. The reason is that using the same amount of data for all edges assumes all attribute-pairs have the same variance. This is in contrast to our algorithm, which determines the

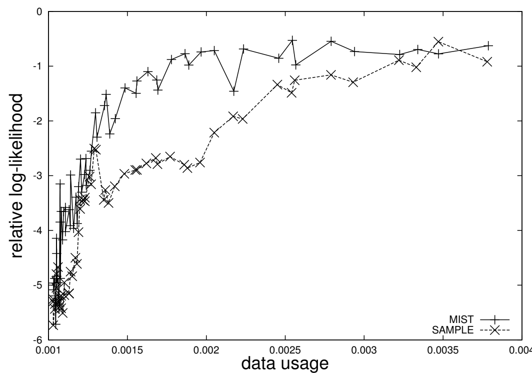


Fig. 9 Relative log-likelihood vs. the sample-based algorithm, drawn against the fraction of data scanned.

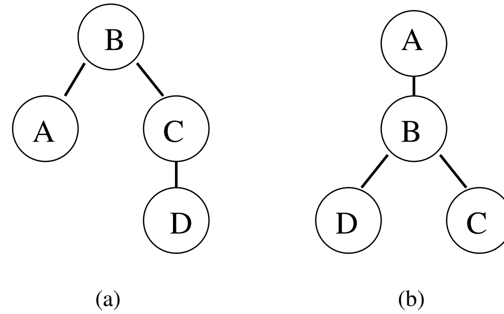


Fig. 10 Two dependency trees

amount of data for each edge independently. Apparently, for some edges this decision is very easy, requiring just a small sample. These "savings" can be used to look at more data for high-variance edges. The sample-based algorithm would not put more effort into those high-variance edges, eventually making the wrong decision. In Figure 9 we show the log-likelihood difference for a particular (randomly generated) set. Here, multiple runs with different δ and ε values were performed, and the result is plotted against the fraction of data used. The baseline (0) is the log-likelihood of the tree grown by the original algorithm using the full data. Again we see that MIST is better over a wide range of data utilization ratios.

With regard to log-likelihood, we are aware that it does not directly reflect differences in tree structures. However, it is a good indicator for the modeling power of a given tree. For example, consider the trees in Figure 10. The probability function of tree (a) is:

$$P(X) = P(D|C)P(C|B)P(A|B)P(B)$$

while tree (b) represents:

$$P(X) = P(C|B)P(D|B)P(B|A)P(A) \ .$$

In other words, even thought the structures are different, the log-likelihood differences is only affected by the differences between $P(D|B)$ and $P(C|B)$, and if these two conditional probabilities happen to be the same, so is the log-likelihood.

Keep in mind that the sample-based algorithm has been given an unfair advantage, compared with MIST: it knows exactly how much data it needs to look at. This parameter is implicitly passed to it from our algorithm, and represents an important piece of information about the data. Without it, there would need to be a preliminary stage to determine the sample size. The alternative is to use a fixed amount (specified either as a fraction or as an absolute count), which is likely to be too much or too little. Another option is to iterate over increasing data sizes (for example, double the sample size in each iteration). The problem with this approach is that it still leaves open the question of how to determine if the size is big enough.

Output: A data-record as a vector $\{X(0)\ldots X(n-1)\}$ of attributes.

- $X(0)$ is a value drawn from $N(0,1)$.
- $X(10i)$ is a value drawn from $N(X(10(i-1)),1)$.
- $X(10k+j)$ for $j \neq 0$ is drawn from $N(X(10k+j-1),j)$.

Fig. 11 The data-generation algorithm

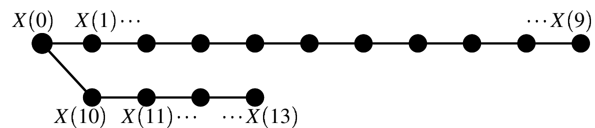


Fig. 12 Structure of the generated data for 14 attributes.

5.1 Sensitivity Analysis

We now examine the effect the user-supplied parameters ε and δ have on performance. Unless otherwise specified, here are the default values for the parameters. We set δ to 1% and ε to 0. The initial sample size is $5,000$ records. There are $100,000$ records and 100 attributes. The data is real-valued. The data-generation process for the synthetic sets is as in Figure 11. The correct dependency-tree for this process is shown in Figure 12. In the categorical case, the network is identical, but parent-child relationships are as follows. The root is true with probability 0.5. For the other nodes, the probability of them being true given that their parent is true is $0.5+c$ for some constant c, and the probability of them being true given that their parent is false is $0.5-c$. By setting the "coupling" parameter c to 0 we get a completely random data, while a value of 0.5 generates a highly-structured, noiseless data set.

Our next experiment examines the sensitivity of our algorithm to noisy data. Data was generated in the usual way, except that some fraction of the records had completely random values in all attributes. As shown in Figure 13, when

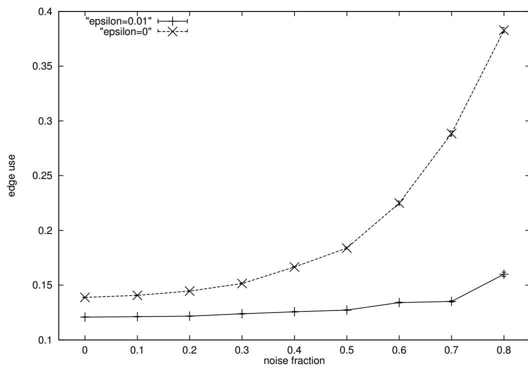


Fig. 13 Edge usage as a function of noise.

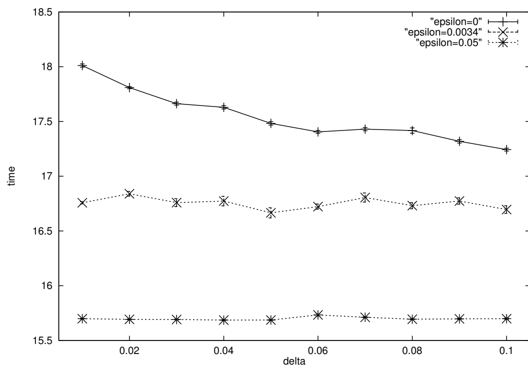


Fig. 14 Running time, in seconds, as a function of δ and ε.

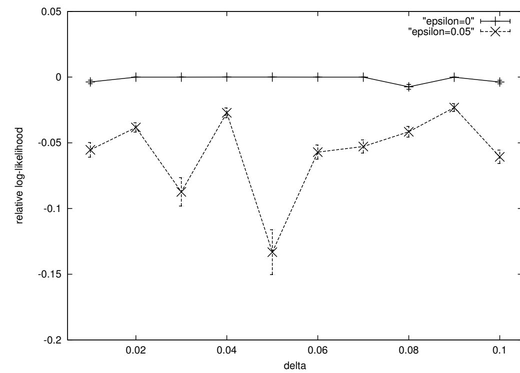


Fig. 15 Difference in log-likelihood (average per record) of the generated trees, as a function of δ and ε. Baseline log-likelihoods were in the order of 3.5×10^6.

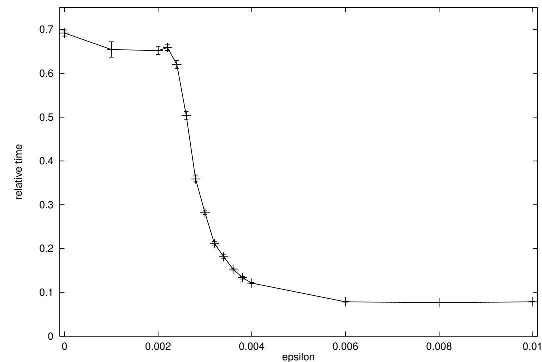


Fig. 16 Edge-usage as a function of ε. The data is categorical, with the "coupling" parameter c set to 0.04.

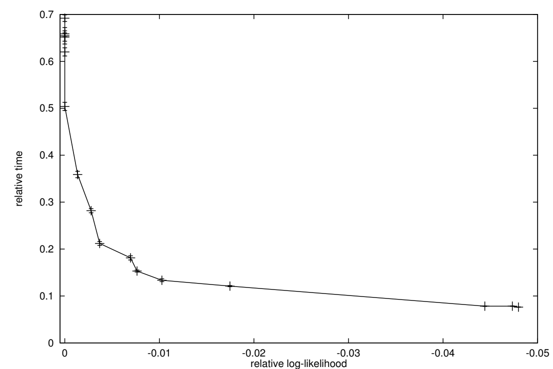


Fig. 17 Relative log-likelihood vs. relative time, as a function of ε. This is data from the same experiments as plotted in Figure 16.

ε is 0, data-usage is kept below 15% of maximum, similar to the performance with noiseless data, as long as the noise level is below 30%. With ε set to 0.01, this is true for all noise levels up to 80%.

Recall that the δ parameter controls how loose the confidence intervals are. The bigger it is, the higher the chance that a wrong decision about a tree-edge inclusion or exclusion will be made. Figure 14 shows the effect of δ on the running time. When ε is 0, it appears that higher values of δ do not improve the running time significantly, while increasing the chance of deviation from the output of the full algorithm. However, when ε was set to 0.05, an improvement in running-time can be traded for some decrease in the quality of the output (Figure 15). For this case none of the 30 runs in any of the 10 values for δ resulted in the same identical tree as with the full algorithm.

We continue to examine the effect the ε parameter has on performance. Recall that it controls a heuristic that may decrease the edge usage, but may also lead to the wrong edges being included in the tree. See Figure 16 for the effect on running time (or, equivalently, on the number of data cells scanned). We see that changes in ε can dramatically improve performance, down from 70% to about 10% on this data-set, with a sharp drop in the 0.002 — 0.004 range. The interesting question is, how badly is the output quality affected by this heuristic. To answer this we have plotted the data from the same experiments, but now with the X (not Y) axis being the relative log-likelihood of the output (Figure 17). The worst log-likelihood ratio is about 0.05, and it seems that with careful selection of ε it is possible to enjoy most of the time savings while sacrificing very little accuracy. For this particular data set this "sweet-spot" approximately corresponds to $\varepsilon = 0.0028$.

5.2 Buffer Cache Performance

As mentioned above, non-sequential data access can degrade performance. The buffer cache was designed to alleviate the problem, and below we quantify the extent to which it succeeds. In the tests below, the SQL back-end was a Postgresql 7.3.4 server on the same machine, using local disk for storage. The data tables were verticalized, and an index created on the "row" and "column" values. The Postgresql "CLUSTER" command was used to arrange the table on disk in index order.

Remember that the total size of the buffer pool is specified by the user. An issue we left open is the number of buffers in the pool. Many buffers offer more flexibility in allocation. But as the size of each buffer decreases, the overhead associated with buffer management increases. Figure 18 shows results on synthetic data with $40,000$ records and a one-megabyte buffer pool. For this experiment, the pool was deliberately chosen to be too small. Within its operating range it performs well enough to mask any measurable effect, as shown below. For this experiment, we varied the number of buffers, and measured the accesses to data that was previously fetched. In other words, we counted the number times in which the disk had to "go back" in the data order. We see that if the number of buffers is very low, performance plummets. This is because buffers are evicted very quickly, and are very rarely re-used. But if the number of buffers is at least approximately as large as M, then the non-sequential access cost is no worse than four times the amount of work required to read the full data. Considering this experiment, and taking into account variations most likely to occur in real data, we recommend using $10 \cdot M$ buffers.

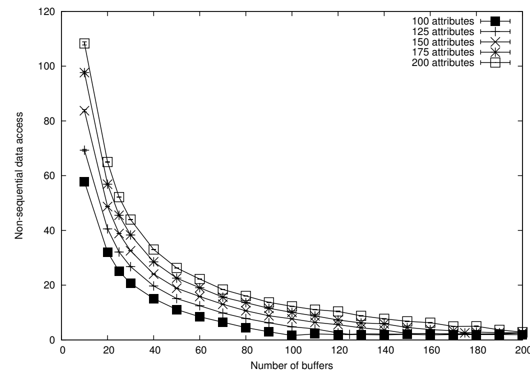


Fig. 18 Relative amount of data accesses out-of-order (in multiples of the original data size). The data has 40000 attributes, and the buffer pool size is 1MB.

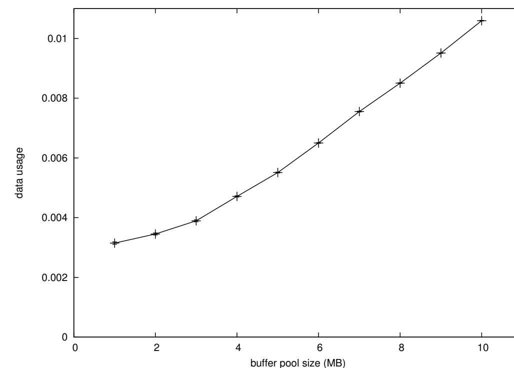


Fig. 19 Amount of data read, in data cells. The data has 100 attributes, and the buffer pool size consists of 2 megabytes divided among 1000 buffers.

Under less extreme conditions, the database-backed variant behaves similarly to the original memory-backed implementation. In Figure 19 we measure performance versus the total pool size. Here, we used synthetic data sets with 0.5 million records and 100 attributes, we varied the buffer size between 1 and 10 megabytes. This compares to the total data size of 400 megabytes. The pool was divided into 1000 buffers. We see that about 1% of the data was accessed through the cache. This agrees with the results shown earlier for the memory-based implementation. In all of the experiments, we measured the amount of data accessed in non-sequential manner, and found it to be zero. We can also see that data usage slowly grows with the cache size. This is because data is always read and used if it is already in a buffer. In theory, the algorithm could stop reading data records as soon as it eliminates an edge. In practice, the processing of records following this point until the end of the buffer is "wasted". The waste is potentially bigger in larger buffers.

Lastly, we wanted to verify that the database-backed variant does not change the fundamental characteristics of the original version. In particular, that the run time, holding all other factors constants, is near-constant in the number of records. Our experiments in this regard show mixed results.

First, when inspecting the total number of data cells accessed by the program, (Figure 20), we see near-constant behaviour regardless of the number of records. The constant itself is higher than it is for the memory-based version (Figure 6), for the reasons explained above. This supports our scalability claim.

However, if we plot the actual run-time for the program (Figure 21), we see a clear linear dependence. This is surprising also because in the memory-based implementation, the run time was very tightly coupled with the data cell count. But here, we see divergence, as one is near-constant, and the other linear. We suspect an implementation or instrumentation problem with our code, but so far were unable to pin it down. Another possible factor might be related to Postgresql. Unfortunately, the total time it takes to run such experiments, including popluating the SQL tables, is so long that we are unable to conduct more thorough exeperiments.

5.3 Real Data

To test our algorithm on real-life data, we used data sets from various public repositories (Blake & Merz, 1998; Hettich & Bay, 1999), as well as analyzed data derived from astronomical observations taken in the Sloan Digital Sky

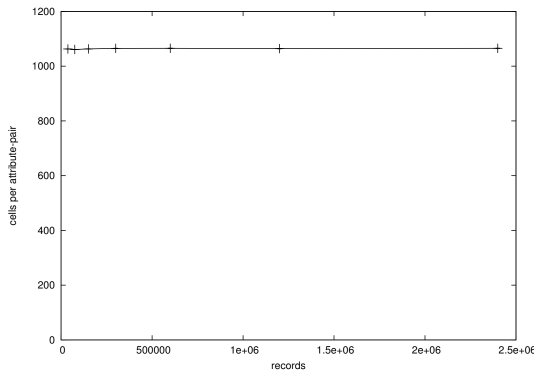


Fig. 20 Amount of data read, in attribute-pair units per attribute. The data has 100 attributes, and the buffer pool size consists of 2 megabytes divided among 1000 buffers.

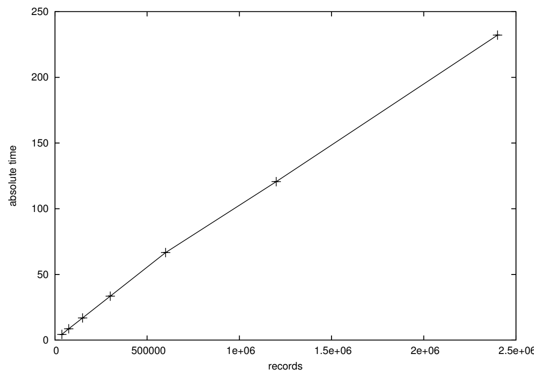


Fig. 21 Running time, in seconds, using the PGSQL-backed implementation . The data has 100 attributes, and the buffer pool size consists of 2 megabytes divided among 1000 buffers.

Survey (SDSS, 1998). On each data set we ran a 30-fold cross-validation test as described above. For each training fold, we ran our algorithm, followed by a sample-based algorithm that uses as much data as our algorithm did. Then the log-likelihoods of both trees were computed for the test fold. Table 1 shows whether the 99% confidence interval for the log-likelihood difference indicates that either of the algorithms outperforms the other. In seven cases the MIST-based algorithm was better, while the sample-based version won in four, and there was one tie. Remember that the sample-based algorithm takes advantage of the "data usage" quantity computed by our algorithm. Without it, it would be weaker or slower, depending on how conservative the sample size was.

We then turned the SDSS data into a *second-order* data set to provide an example of data with many attributes and many records. We first discretized all of the attributes. Then we added all pairwise conjunctions of these attributes. There were 23 original attributes $X_1 \dots X_{23}$ to which were added $\binom{23}{2}$ additional attributes $A_{i,j}$ where $A_{i,j} = X_i \wedge X_j$.

After doing that for all attributes and removing attributes which take on constant values we were left with 148 attributes and the original 2.4 million records. The naive algorithm constructs a tree for this set in 6.6 hours, while the fast algorithm (with default settings) takes about 21 minutes, meaning a speedup of 19. The tree generated by the fast algorithm weights 99.89% of the naive tree, and the difference in log-likelihoods is 1.26×10^5, or about 0.05 per record.

As far as run time is concerned, we measured it to be closely related to the number of value pairs visited. In other words, the edge usage value for an accelerated run is almost the same as the fraction of its run-time, to the run over the full data. As can be seen Table 1, many of these fractions are very small, making exact measurements impossible. Where it could be measured, this observation holds both for our synthetic and real data runs.

6 Red vs. Blue Rule

Our algorithm is based on the "red edge" rule. As mentioned above, it is also possible to base MST algorithms on the "blue edge" rule. A natural question to ask is: will the blue edge rule be beneficial in a framework that utilizes probabilistic intervals on edge weights?

Before attempting to answer, we review the difference between the rules. The red rule operates on a *cycle* in the graph; the heaviest edge on the cycle can be eliminated.[5] See Figure 22. In contrast, the blue rule operates on a *cut* in the graph. A cut is defined by a subset of the nodes, and consists of the edges that have exactly one endpoint in the set. The lightest edge in a cut can be proven to be in the MST.[6] Therefore it is an inclusion rule. See Figure 23. The blue rule is used exclusively in the popular Prim and Kruskal algorithms. There are also algorithms that combine both rules.

Returning to the question of using the blue rule in a Chow-Liu framework, we believe it will not be beneficial, for the following reason. The number of edges in a cut can be much larger than in a cycle: up to $O(M^2)$. To calculate the expected number of edges in a random cut, denote the number of nodes on one side of the cut by i. We assume the probability of a cut is uniform over i. All cuts of this size have the same number of edges $i \cdot (M-i)$, therefore the expected value is:

$$
\begin{aligned}
&\frac{1}{M-1}\sum_{i=1}^{M-1} i \cdot (M-i) \\
&= \frac{1}{M-1}\left[M\sum_{i=1}^{M-1} i - \sum_{i=1}^{M-1} i^2\right] \\
&= \frac{1}{M-1}\left[M\frac{M(M-1)}{2} - \frac{M(M-1)(2M-2+1)}{6}\right] \\
&= \left[\frac{M^2}{2} - \frac{M(2M-1)}{6}\right] \\
&= \frac{M^2+M}{6}
\end{aligned}
$$

[5] The cycle must not contain any red edges.

[6] The cut can not contain any blue edges.

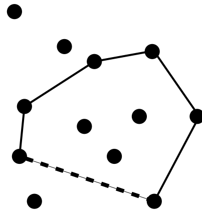


Fig. 22 The Red Edge rule.

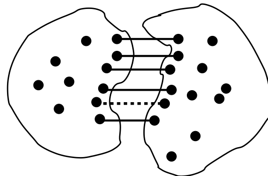


Fig. 23 The Blue Edge rule.

That is, $O(M^2)$. Consider that for a cycle, the *maximal* number of edges is $M-1$. Therefore cuts require knowledge of more edge weights, which in our case usually translates to reading of more data. This is exactly what we are trying to avoid.

7 Error Analysis

The use of probabilistic bounds means that there is a risk of making a wrong decision. We now quantify this risk. For the purpose of the analysis, we treat the tree built by the Chow-Liu algorithm, when given exact edge weights, as optimal. We consider every deviation from this tree as an error.[7] For simplicity we assume that the edge weights are all unique and so is the optimal weight. We call the edges that form the optimal tree "optimal edges". We also ignore the δ optimization (i.e., assume it is set to zero). The event that we are interested in is that the tree output by the modified MIST algorithm is identical to the optimal tree.

Consider a MIST run which ends in the optimal tree. It starts with an arbitrary tree, and eliminates and swaps edges as in Section 4. We make two observations. One, it is sufficient to consider just steps in which an edge is eliminated. This is true because there is no risk of making a mistake by deferring the decision due to insufficient data. Two, in all the elimination steps, edges that are not in the optimal tree are eliminated.

Now, follow the sequence of execution; each elimination step corresponds to one of the following scenarios:

1. Eliminate a non-optimal and non-tree edge because it is heavier than a tree edge.
2. Swap an optimal and a non-optimal edge because the optimal edge is lighter. The non-optimal edge swapped out from the current tree is then eliminated.

[7] A less strict error analysis would consider the expected weight difference between the generated and optimal trees. Conceivably, one could make some assumptions on edge weight distribution to perform this kind of analysis.

In both cases, a non-optimal edge is eliminated because a weight comparison determines it is heavier than an optimal edge. We calculate the probability of arriving at the opposite decision erroneously. Let A be the true weight of the optimal edge (meaning the value used by the original Chow-Liu algorithm), and let a be the corresponding confidence interval. Similarly, let B be the true value for the non-optimal edge and b its interval. See Figure 24.

In our case, $B > A$. A mistake happens by having a and b such that the edge associated with a is eliminated. Since the algorithm defers all decisions based on overlapping intervals, the only configuration allowing this is where $\min(a) > \max(b)$ (see Figure 24(d)). Given that $B > A$, this cannot hold if both a and b contain their respective true values. The probability of each interval not containing the true value is at most ε, and the probability of not making either mistake is at most $(1-\varepsilon)^2$. Therefore the probability of making the wrong decision is at most $1-(1-\varepsilon)^2 = \varepsilon(2-\varepsilon)$. Note that this bound is loose, since not every failure to include the exact value in the intervals results in full inversion of the intervals.

As explained above, this kind of decision is made exactly once per eliminated edge. Their number is just the total number of edges which are not tree edges, or $\binom{M}{2} - (M-1) = (M-1)^2$. We now have a bound on the probability of failure in a single test, and we know the number of tests. We can hence derive a lower bound on the probability of making no mistakes: $\left[\varepsilon(2-\varepsilon)^2\right]^{(M-1)^2} = \left[\varepsilon(2-\varepsilon)\right]^{2\cdot(M-1)^2}$.

8 Conclusion

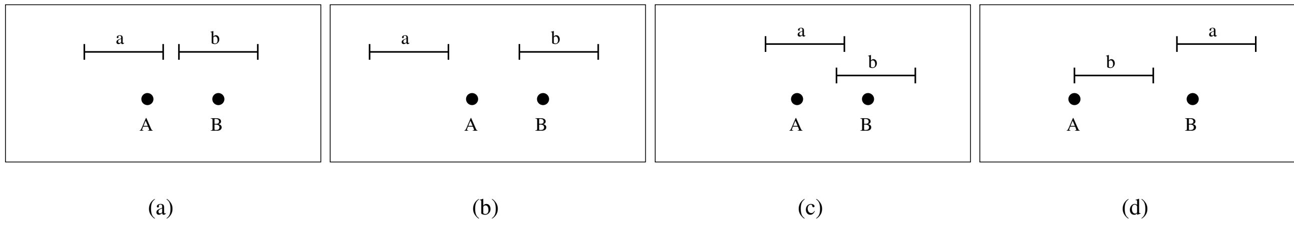
We have presented an algorithm that applies a "probably approximately correct" approach to dependency-tree construction for real-valued and categorical data. Experiments in data sets with up to millions of records and hundreds of attributes show it is capable of processing massive data sets in time that is constant in the number of records, with just a minor loss in output quality.

While we derived formulas for both numeric and categorical data, we currently do not allow both types of attributes to be present in a single network. We address this issue in a forthcoming paper.

References

Blake, C., & Merz, C. (1998). UCI repository of machine learning databases. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Chow, C. K., & Liu, C. N. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, *14*, 462–467.



(a) (b) (c) (d)

Fig. 24 Several possibilities for two estimated intervals *a* and *b* and their respective exact values *A* and *B*.

Table 1 Results, relative to the sample-based algorithm, on real data. "Type" means numerical or categorical data.

NAME	ATTR.	RECORDS	TYPE	DATA USAGE	MIST BETTER?	SAMPLE BETTER?
CENSUS-HOUSE	129	22784	N	1.0%	×	√
COLORHISTOGRAM	32	68040	N	0.5%	√	×
COOCTEXTURE	16	68040	N	4.6%	×	√
ABALONE	8	4177	N	21.0%	×	×
COLORMOMENTS	10	68040	N	0.6%	×	√
CENSUS-INCOME	678	99762	C	0.05%	√	×
COIL2000	624	5822	C	0.9%	√	×
IPUMS	439	88443	C	0.06%	√	×
KDDCUP99	214	303039	C	0.02%	√	×
LETTER	16	20000	N	1.5%	√	×
COVTYPE	151	581012	C	0.009%	×	√
PHOTOZ	23	2381112	N	0.008%	√	×

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1989). *Introduction to algorithms*. McGraw-Hill.

Davies, S. (2002). *Scalable and practical probability density estimators for scientific anomaly detection*. Doctoral dissertation, Carnegie-Mellon University.

Domingos, P., & Hulten, G. (2000). Mining high-speed data streams. *Proceedings of 6th International Conference on Knowledge Discovery and Data Mining* (pp. 71–80). N. Y.: ACM Press.

Domingos, P., & Hulten, G. (2001a). A general method for scaling up machine learning algorithms and its application to clustering. *Proceedings of the Eighteenth International Conference on Machine Learning*. Morgan Kaufmann.

Domingos, P., & Hulten, G. (2001b). Learning from infinite data in finite time. *Advances in Neural Information Processing Systems 14*. Vancouver, British Columbia, Canada.

Friedman, N., Goldszmidt, M., & Lee, T. J. (1998). Bayesian Network Classification with Continuous Attributes: Getting the Best of Both Discretization and Parametric Fitting. *Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufmann, San Francisco, CA.

Friedman, N., Nachman, I., & Peér, D. (1999). Learning bayesian network structure from massive datasets: The "sparse candidate" algorithm. *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-99)* (pp. 206–215). Stockholm, Sweden.

Goldenberg, A., & Moore, A. (2004). Tractable learning of large bayes net structures from sparse data. *Proc. 21st International Conf. on Machine Learning*.

Hettich, S., & Bay, S. D. (1999). The UCI KDD archive. `http://kdd.ics.uci.edu`.

Maron, O., & Moore, A. W. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. *Advances in Neural Information Processing Systems* (pp. 59–66). Denver, Colorado: Morgan Kaufmann.

Meila, M. (1999a). An accelerated chow and liu algorithm: fitting tree distributions to high dimensional sparse data. *Proceedings of the Sixteenth International Conference on Machine Learning*.

Meila, M. (1999b). *Learning with mixtures of trees*. Doctoral dissertation, Massachusetts Institute of Technology.

Moore, A. W., & Lee, M. S. (1994). Efficient algorithms for minimizing cross validation error. *Proceedings of the Eleventh International Conference on Machine Learning* (pp. 190–198). New Brunswick, US: Morgan Kaufmann.

Pelleg, D. (2004). *Scalable and practical probability density estimators for scientific anomaly detection*. Doctoral dissertation, Carnegie-Mellon University.

Reza, F. (1994). *An introduction to information theory*, 282–283. New York: Dover Publications.

SDSS (1998). *The sloan digital sky survey*. SDSS. `www.sdss.org`.

Tarjan, R. E. (1983). *Data structures and network algorithms*, vol. 44 of *CBMS-NSF Reg. Conf. Ser. Appl. Math.* SIAM.