Bayesian Network Search by Proxy

Benjamin Yackley, Blake Anderson, and Terran Lane

April 1, 2011

Abstract

Existing methods to search for an optimum Bayesian network suffer when the size of the data set grows to be too large. The number of possible networks grows superexponentially in the number of variables, and it becomes increasingly time-consuming to get reasonable results; in fact, finding an exact optimal network for a given data set is an NP-complete problem, so the question is often to find a network which is "good enough". However, as the numbers of instances and variables in the data set grow, the time to take even a single search step can get very costly. Searching by proxy can alleviate this problem; by selecting a random set of training samples and constructing an approximator around those, we can greatly reduce the time it takes to find a network with a score comparable to that obtainable by the same search algorithm using exact scoring. Moreover, with enough training samples, we can obtain networks with significantly better scores in a fraction of the time. However, with too many samples, overfitting occurs and the results do not improve as the number of samples increases. We conjecture that this is because the approximator smooths out the search landscape, making it less likely to get stuck in local minima, and give experimental evidence to support this.

1 Introduction

The problem of searching for an optimal Bayesian network given a set of data is one that has been studied for decades. However, modern machine learning problems often need to deal with much larger data sets than current methods can plausibly handle; the number of possible edges in a network grows quadratically with the number of nodes, and the number of possible networks grows exponentially in the number of edges. Furthermore, even the task of scoring a single network can grow linearly in the number of instances in the data. Methods such as the ADTree[12] exist to ameliorate this problem, but in networks with enough variables, creating an ADTree in the first place becomes infeasible. We propose here a new strategy for discovering high-scoring Bayesian networks for very large data sets, ones too large to realistically handle any other way.

This strategy, searching by proxy, is compatible with any existing search method that computes the scores of the networks as it searches; the idea here is not to change the search method itself, but the representation it uses to calculate scores. We create a proxy function — a Gaussian Process regressor — which we train on a selection of randomly drawn sample networks and their corresponding BDe scores, and then use our proxy function as a quick way to search. Given enough training samples (and "enough" can turn out to be very small), the resulting scores at the end are equivalent to or even better than those obtained by searching using a standard scoring function while taking less time to obtain.

There are two reasons this works as well as it does. First, and more obviously, the proxy function takes much less time to compute than a full exact BDe score. ADTrees are also a way of calculating these scores very quickly, but the up-front time and space required to build an ADTree in the first place can be prohibitively expensive. The second reason, detailed in the latter part of our experiments, is that creating an approximator smooths out the search space, reducing the chance of getting stuck in a local minimum.

There are some caveats to keep in mind. First, it should be noted that this work is not about the search process itself, but about representation. Searching by proxy is compatible with any search method that depends on calculation of a graph's score; it replaces the score calculation step, but leaves the rest of the algorithm unchanged. Second, this method searches over a smooth approximation to the true surface, which has two effects. One is that it is possible (and, as our results show, very probable) to get results that are significantly better than those an exact-scoring-based search would produce because of the reduced chance of getting stuck in a local optimum. However, adding more training samples will not always increase the effectiveness of the approximator; as our results show, there is a point past which the search performs no better given an increasing number of samples, and in fact can perform worse than it would with fewer training points. The data sets here are all assumed to be fully observed; in future work, we plan to address the possibility of using this same method to ease the search for optimal networks given missing data or even completely unobserved variables.

2 Bayesian Networks

A Bayesian network is a model of the independence relations of a set of variables. Suppose we have a data set with variables $x_{1 \ldots n}$, with m independent instances of values these variables take, assumed to be drawn from an unchanging underlying distribution. We could attempt to encode this joint distribution $p(x_1, x_2, \ldots, x_n)$ explicitly in an attempt to understand the underlying processes behind our data, but this suffers from two problems. One is that it hides possible dependence/independence relations between the variables – ones that might be useful for understanding the data – but the more important one is that it takes an exponentially large number of instances to learn the structure of a full joint distribution.

The alternative, then, is to assume independence relationships between the variables, encoding the dependencies in the form of a directed acyclic graph. The full joint distribution can then be factored into a product of marginals, one per variable:

$$p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i|\mathrm{Pa}(x_i))$$

The notation $\mathrm{Pa}(x_i)$ here means 'the parents of node x_i', or those variables with edges running from themselves to x_i. Now, instead of needing to learn the entire joint distribution at once, we only need to learn a set of conditional distributions, one per variable, given their parents in the graph. Finding these distributions is relatively easy; they can be readily estimated from the data set. The far more difficult (and interesting) problem is to find the structure of the graph.

3 Network Search

3.1 Notation and Terminology

Given a set of data, we wish to be able to find a graph $G \in \mathbb{G}_{DAG}$ with labeled nodes such that it represents the optimal Bayesian Network to explain a set of causal relationships between the variables of our data set. In order to make the word "optimal" make sense, of course, we need a numeric score to optimize; the computation of this will depend entirely on the structure of the graph and the set of data. The set $\mathbb{G}_{DAG}$ is the subset of the set of all graphs $\mathbb{G}$ consisting only of directed acyclic graphs (thus the acronym DAG). Additionally, we are only ever interested in graphs with the same number of nodes as the number of variables in our data set; our convention is to assume that we are searching over the correct subset of $\mathbb{G}_{DAG}$, since none of our search algorithms involve adding or deleting nodes.

We use the convention that our data set D is a matrix in $\mathbb{R}^{m \times n}$, where m is the number of data instances, and n is the number of variables. The data set D is fully observed, with no missing values or hidden variables. Each variable is assumed to be discrete, taking on values 1 to r_n (termed variable n's arity). Following Heckerman[9], we use the following notation to relate variables to their parents: $\mathrm{Pa}(x_i)$ denotes the set of parents of the variable x_i, and $q_i = \prod_{x_j \in \mathrm{Pa(x_i)}} r_j$ is the total number of possible configurations of the parents of variable x_i. The notation N_{ijk} indicates the number of instances (i.e. rows) in our data set D where $x_i = k$ and $\mathrm{Pa}(x_i) = j$, with k ranging over values 1 through r_n and j ranging over all possible configurations of values for x_i's parents. Related to this, $N_{ij} = \sum_k N_{ijk}$. The notations N'_{ij} and N'_{ijk} are used for pseudo-counts, or prior beliefs on the relative values of N_{ij} and N_{ijk}; these are hyperparameters on the Dirichlet prior on individual conditional probability tables. The BDe score of a graph, defined in full below, is the function $s_{BDe} : \mathbb{G}_{DAG} \to \mathbb{R}$ mapping from the set of directed acyclic graphs $\mathbb{G}_{DAG}$ to real numbers, while $s(G|D)$ indicates the score of graph G given data set D.

When we construct the approximator, we use the term "samples" to denote the testing graphs $g_1, g_2, \ldots g_{n_s}$ to be take exact scores of (as opposed to "instances", which are the individual points in our original data set D). The function $k(\cdot,\cdot)$ is a kernel function between graphs, with the matrix K and column vector $K*$ defined as $K_{ij} = k(g_i, g_j)$ and $K^*_i(g^*) = k(g_i, g^*)$ for some other graph g^*.

3.2 Scoring Functions

From the above, then, we want to find an edge set E for our graph G that optimizes some scoring measure given a corresponding data set over the variables. In other words, we want

$$\hat{E} = \arg\max_E s(G|D)$$

We use the BDe score as defined in Heckerman, which can be thought of as a Bayesian posterior estimation of the probability that the data set was generated from a probability model represented by graph G. Higher BDe scores indicate graphs which are more likely, and therefore preferable over those with lower scores, although as a logarithm of a fraction, BDe scores are necessarily always negative. The definition of the scoring function we use, the log-BDe, is:

$$s(G|D) = ln\left(p(G)\prod_{i=1}^{n}\prod_{j=1}^{q_i}\frac{\Gamma(N'_{ij})}{\Gamma(N'_{ij}+N_{ij})}\prod_{k=1}^{r_i}\frac{\Gamma(N'_{ijk}+N_{ijk})}{\Gamma(N'_{ijk})}\right)$$

This is the logarithm of the BDe function as given in Heckerman; we use this form for its ease of computation (it can be expanded as a sum of differences of log-gamma functions) and because computing the actual probability would rapidly lead to a numeric underflow. Working in the space of logarithms, scores are typically in the negative thousands; a higher score (i.e. closer to zero) is still better. However, even in log-space, calculation of this score can take quite a while, depending on it does as a triple-nested product, first over all nodes, then over all possible sets of values of each node's parents (which there are an exponential number of with larger parent sets), and finally over all possible values for a node given a single parent configuration; counts are required in the second (N_{ij}) and third (N_{ijk}) levels, so a naive approach of simply counting over the entire data set every time one of these is encountered can take a very long time indeed. However, since all of these counts are of the form "how many instances fit the query $x_1 = v_1 \wedge x_2 = v_2 \ldots x_t = v_t$", ADTrees are a natural choice to use for accelerating them, since they are designed to return counts of that exact form in very little time. As we shall see, though, even ADTrees bog down when used with sufficiently large data sets.

4 Prior Work

One of the earliest methods to find optimal Bayesian networks is the Chow-Liu Tree[5], which very quickly generates a network from a given set of data (and computed mutual information values between each pair of variables). The Chow-Liu Tree has been proven to be optimal within the space of networks constrained to tree topology[14]. However, most interesting networks are not trees; despite the existence of an efficient algorithm to find a tree, the general network search problem is, in fact, NP-complete as long as the actual optimum is required[4]. A brute-force search based on enumerating all possibilities quickly grows unfeasible, since the number of possible directed acyclic graphs grows superexponentially; there are 543 directed acyclic graphs on 4 nodes, but $29,281$ on 5. To alleviate this problem, there are methods to search through the set of DAGs in a smarter fashion. The simplest way to do this is incrementally, as with a greedy search that explores, at every step, the set of possible DAGs that can be generated through the addition or deletion (or, in some algorithms, reversal) of a single edge, and then accepts the change that gives the highest-scoring graph, continuing until no further improvements can be made[3]. As with nearly any greedy search, this does tend to get stuck in local optima; variations on this search technique adapt ideas from simulated annealing[10] or beam search[11]. Another possibility that is often used is a search based on a Markov Chain Monte Carlo algorithm; here, the algorithm wanders around through the space of graphs in such a way that it spends the most time in higher-scoring areas, and this can be left to run for as long as one desires, returning the highest-scoring graph seen so far as an answer upon being halted[8].

All of these methods share a common problem, however, in that large data sets quickly become cumbersome to find an optimal network on. To calculate a network's score the most straightforward way requires looking at each instance in the data set separately, which means that the time to score a network grows linearly in the number of instances; because of the i.i.d. assumption, each instance is independent of the others, and so the time to score a whole data set must be the sum of the times it takes to score each instance. With all of the data fully observed, each instance will take the same amount of time to score.

An even larger problem, however, is what happens as the number of variables increases. The size of the search space grows approximately at $O(2^{n^2})$ for n nodes[17], which means that many interesting and useful data sets are outside the realm of plausibility using conventional methods. Even smarter scoring methods fall into this exponential growth problem as the number of nodes increases; a cache of scores for individual nodes and their possible sets of parents will grow at $O(2^n)$.

4.1 ADTrees as Accelerator

ADTrees were introduced as a helper structure for a data set in order to make arbitrary queries of the form "how many instances in this set have $x_1 = v_1$, $x_2 = v_2$, ... and $x_q = v_q$" (where the v_i denote specific values that the variables may take) run in time independent of the actual number of instances in the set. This is accomplished through clever precaching. The root node of the ADTree is a "count node"; all children of count nodes are called "vary nodes", and all children of vary nodes are count nodes, giving the overall tree a stratified structure where the levels alternate between count and vary nodes. Count nodes are so named because they contain a count of how many instances match some specific query; the root node corresponds to a query that specifies no values for any variable, and therefore the number contained there is the number of data points in the entire set. Below each count node is a set of vary nodes, one for each variable that is not already contained in the count node's set of queries (with an additional condition to be explained later), and beneath each vary node is a set of count nodes, each one corresponding to a single value the vary node's variable could take. However, because many of the counts are redundant and calculable from other counts with some clever addition and subtraction, much of the apparent complexity can be pruned away.

Most obviously, if a query contains a set of values which no instances of the data set match at all, that entire count node can simply be replaced by a leaf labeled "0"; no further expansion needs to be done there, because every possible child of a tree with a zero count must also have a zero count. We can also assume that our queries pick out variables in a fixed order; for instance, if we want to query the tree for how many instances match $x_1 = 0 \wedge x_2 = 1$, we should get the same answer whether we query x_1 or x_2 first, making one of those paths redundant. Without loss of generality, then, we can assume all queries follow a strict ordering of the variables, and therefore prune all possible paths in the tree which would contradict this ordering. In other words, the count nodes beneath the top-level "vary x_1" node start from x_2, those beneath the "vary x_2" node start from x_3, and so forth.

The final simplification is the "most common value" speedup, which accounts for ADTree's excellent performance on data sets which are either entirely binary or have heavily lopsided distributions. Instead of explicitly encoding every possible value beneath each vary node, we first find out which value is the most common and then replace the corresponding count node with an "MCV" leaf, never expanding it any further. The key idea here is that if we have all but one count for the children of a single vary node (each corresponding to a possible value of a single variable), we can obtain the other through simple subtraction; take the count on that vary node's parent, subtract all of the counts on our unknown value's siblings, and the remainder is the count we want. This subtraction trick is sufficient to derive counts for any possible query that can be made while cutting down on the tree's size significantly.

However, ADTrees run into a significant problem for large data sets — not ones which are large in the number of instances, necessarily, but in the number of variables. Consider the overall size of an ADTree; from the root node at the top, we branch out into n separate vary nodes which then each branch out to r_i new ADTrees (which begin at x_2, x_3, and so on); the overall size of the tree, then, is $O(\prod_i r_i)$, which is clearly exponential in the number of variables. Even given the "most common value" speedup, each of these factors is reduced from r_i to $r_i - 1$; this is a clear win in the case when all variables are binary (so $r_i = 2$

Algorithm 1 Outline of a generic Bayesian Network search algorithm

```
Input: a data set $D$
  1. $G$ = an initial DAG
  2. $sc$ = the score of $G$ given scoring function $s(\cdot|D)$
  3. $G'$ = the next DAG to be examined
  4. $sc' = s(G'|D)$
  5. **if** $sc' > sc$ **then** $G = G'$ and $sc = sc'$
  6. If we haven't reached an optimum, go to step 3
```

for all i), since the tree has a branching factor of 1 out of each vary node[1]. However, many interesting data sets are not, and the size of the tree remains exponential in those cases.

Furthermore, the size of an ADTree also depends on the data's sparsity; a data set where one variable's values are equally proportionate will branch at that variable's Vary node into equal-sized ADTrees, while one with a heavily lopsided distribution of values will branch into a very large tree for its most common value (the basis for the MCV speedup, since this large tree is no longer needed) and very small ones for the least common ones. This is the trick used in Moore[12] to reduce a 97-variable data-set (the Birth data) to a reasonable size; as they state, in over 70 of these variables, over 95% of the values are False. We can make no such guarantees about data sets in general.

ADTrees, then, do offer significant speedups in calculating exact BDe scores, but run into significant problems with both size and speed on general data-sets with large numbers of variables. As the results show, they are useful for small to medium data sets, but in the realm of the very large, fail to keep up.

5 A Proxy-Based Search for Bayesian Net Structures

The term "proxy-based search" refers not to a specific search technique, but rather a way to accelerate any already-existing search method. When dealing with data sets of sufficient size, as proxy-based search is intended for, conventional methods simply take too long to produce reasonable answers. ADTrees do help in cases with a large number of instances, but suffer from the exponential amount of storage space needed as the number of variables grows. The answer is to construct a proxy function — one that can take an arbitrary graph and output a number which will be close enough to the real score that we can use the proxy to search instead of exact scores.

Algorithm 1 is an intentionally generic search for a Bayesian network given an input data set D; a typical algorithm for searching for an optimal network progresses along those lines, with suitable ways of defining steps 2 and 3 specific to the algorithm in question. Algorithm 2 is a modified form of Algorithm 1 demonstrating the basic concept of a search by proxy; we introduce a new parameter, the number of sampled random networks we score first, and then build an approximator. The advantage is not immediately obvious, but consider that the scoring step (step 4 in algorithm 1) is going to be repeated potentially many times; if the calculation of the score takes a long time, this time will be multiplied by the number of calculations we need to make. Proxy-based search can dramatically shorten the time it takes to perform a single calculation, thereby reducing the overall time to perform a search.

5.1 The Metagraph

In prior work[18], we introduced the concept of a "metagraph", a structure which represents a relationship between graphs as a graph itself. Each node in the metagraph as previously defined corresponds to a different edge configuration, while edges connect nodes whose corresponding graphs differ by exactly one edge. Scores,

[1] This is not to say that complicated queries are O(1) in a binary data set; the algorithm to derive counts from the ADTree will still require $O(v)$ look-ups for queries with v components.

Algorithm 2 Outline of a generic Bayesian Network search algorithm, modified to search by proxy

```
Input: a data set $D$ and a number of samples $n_s$
  1. Generate random DAGs $G_1 \dots G_{n_s}$
  2. Calculate training values $s_i = G_i$ for $i = 1 \dots n_s$
  3. Generate $\hat{s}(\cdot|D)$, a Gaussian Process regressor using the graphs $G_i$ and values $s_i$ as training data; see
     section 5.2
  4. $G$ = an initial DAG
  5. $sc = \hat{s}(G|D)$
  6. $G'$ = the next DAG to be examined
  7. $sc' = \hat{s}(G'|D)$
  8. **if** $sc' > sc$ **then** $G = G'$ and $sc = sc'$
  9. If we haven't reached an optimum, go to step 6
```

then, are a function from this metagraph to the real numbers. This construction is a useful one to keep in mind; although we do not explicitly use the hypercube structure to perform our approximation, it led to the simple form of the kernel function defined below, since we can imagine each of the learned weights to be one of the edge lengths of our metagraph hypercube.

5.2 Gaussian Process Regression

The Gaussian Process Regression method[16] (or GPR for short) is based on taking finite sets of known exact function values and constructing an approximator based on a Gaussian distribution in the infinite-dimensional space of functions. For simplicity, this can be equivalently thought of as a "mean function" which is a conventional regressor built out of smooth functions and a separate "covariance function" which says, for every point in the mean function, how much variance the approximator expects to see. Areas which are more densely populated with points from the original data set will have correspondingly low covariances, and, conversely, sparse areas will have a high covariance. However, in this work, we only treat the mean function as our approximator; the covariance function is irrelevant for now, although in future work it may become more useful as a way of determining which parts of a search landscape are more stable.

First, let our training data be the set (x_i, y_i) for $i = 1 \dots n$. Each x_i is drawn from a set $\mathbb{X}$ of arbitrary objects; the properties of the set $\mathbb{X}$ are largely irrelevant except as far as they are needed to define the kernel function $k : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$, a mapping from pairs of elements of $\mathbb{X}$ to the real numbers. The intuitive meaning of k is as a measure of similarity; the larger $k(x_1, x_2)$ is for some $x_1, x_2 \in \mathbb{X}$, the "closer" $x_1$and x_2 are. The specific kernel we use is

$$k(g_1, g_2) = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} I[e_{ij} \in g_1] I[e_{ij} \in g_2]$$

The summation is over the natural numbers up to n (the number of variables, equal to the number of nodes in both graphs); the weights w_{ij} are learned from the data (see below), and the two indicator functions contain abbreviated statements meaning "the edge from variable x_i to x_j is present in graph g". In other words, we assign every edge a weight, and then the value of the kernel is simply the sum of the weights of all edges the two graphs share in common.

Once this is defined, we can then use the conventional form of a GPR to create our approximator:

$$\hat{s}(g^*) = K^*(g^*) K^{-1} s$$

Here, s denotes the column vector of scores $s_{1 \dots n_s}$ where $s_i = s(g_i|D)$.

5.3 Learning the Weights

Following the method in Rasmussen[16], we train our weights (and therefore our kernel function's hyperparameters) by using the marginal likelihood gradient. Restating equation 5.9 in the book, we have:

$$\frac{\partial}{\partial \mathbf{w}} \log p(\mathbf{y}|X, \mathbf{w}) = \frac{1}{2}\mathrm{tr}\left(\left(\alpha\alpha^\top - K^{-1}\right) \frac{\partial K}{\partial \mathbf{w}} \right),\ \alpha = K^{-1}\mathbf{y}$$

Here, $\mathbf{w}$ represents our vector of weight values, one for each possible edge. Maximizing the probability involves setting this expression equal to zero, which can be done through a gradient descent method; note that the derivative still involves K, which depends on $\mathbf{w}$. However, gradient descent works well enough as a way to solve this equation (is this convex?). In our results, the time spent calculating the proper kernel weights is denoted "weight time".

The internal representation of graphs, in our implementation, is as a bit string of length $n(n-1)$, one bit for each possible edge (we omit self-loop edges because they can never occur in a Bayesian network); this makes the calculation of $K^*(g^*)$ particularly simple when the weights are known. If we use $B \in \{0,1\}^{n_s \times n(n-1)}$ to be a matrix where the rows are each given by the bit string equivalent of a single graph in our training set and W is the matrix whose diagonal is given by the elements of $\mathbf{w}$, then K^* is just a matrix product. In formal terms, let the function $b : \mathbb{G} \to \{0,1\}^{n(n-1)}$ be our translation function, each edge of the graph argument corresponding to 1 in the output with zeros everywhere else. The matrices B and W, then, are:

$$B = \begin{bmatrix} b(g_1) \\ b(g_2) \\ \vdots \\ b(g_{n_s}) \end{bmatrix} \quad W = \begin{bmatrix} w_1 & & 0 & \\ 0 & w_2 & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & w_{n(n-1)} \end{bmatrix}$$

Using these definitions, $K^*(g^*) = BWb(g^*)^\top$, and $K = BWB^\top$.

5.4 Search Methods

The simplest form of search we use here is the greedy search, which scores all possible one-edge variants of the network at a given step and uses the variant with the highest score as a basis for the next step, proceeding until we reach a network from which all variants have a lower score. This has the usual issue that all greedy searches have: a tendency to get stuck in local maxima. However, this is useful as a point of comparison; since our contribution is about scoring and not the actual search method, it should suffice to use greedy search for its simplicity and vary the way it gets its scores — either from direct calculation of a network's score (potentially through an ADTree for data sets small enough to support them), or from our Gaussian Process proxy.

5.5 The DAG constraint

One notable problem with the above is that our approximator is defined over the space of all graphs over a certain number of nodes, no matter their structure; since Bayesian networks are restricted to be directed and acyclic, we are generating scores which are potentially nonsensical. However, this is a problem we deal with not in the approximator itself, but in the search process; if, at any point, our search process attempts to move to a graph containing a loop, we immediately reject that graph and generate a new move, only continuing when we generate a true DAG. This checking does not add significantly to the time it takes to search, and as such, the time it takes to reject cyclic graphs is folded into the reported search times in the results.

It should be noted for completeness that, despite the holes that graphs containing cycles leave in the search landscape, the entire set is still connected, with all points reachable from all others; this is most easily seen by noting that any DAG can be transformed into any other by additions and deletions, first by deleting all of the initial graph's edges to leave an empty graph, and then by filling in all of the final graph's edges one by one. Since the final graph by definition has no cycles, there will never be a point at which the construction leaves a cycle in the graph being constructed. There are, of course, often faster routes to transform one graph into another, but this proof does at least show that the space of DAGs remains fully connected under the addition and deletion operations.

6 Results

The initial goal of the experiments was to show that a proxy-based search could beat an ADTree-accelerated one on time taken to get a network of equivalent score, and could beat the ADTree-based search on score when given the same amount of time to search. The results, however, are even more encouraging; not only can we beat an ADTree-based search on time to get to an equivalent score when given enough training samples, but we can also

6.1 Data sets

All data sets used in this research were taken from the UCI data repository[7] apart from the two synthetic networks, which were generated randomly with their given numbers of nodes (20 and 40, respectively) and then sampled to provide 20,000 instances for each. All variables in these two cases were binary; these data sets are called SYNTH20 and SYNTH40.

The ADULT1, ADULT2, and ADULT3 data sets are those used in the ADTree paper[1]; ADULT1 and ADULT2 contain 15 attributes related to census data. ADULT1 has 15,060 instances, ADULT2 30,162. ADULT3 is a concatenation of ADULT1 and ADULT2 into a single data set with 45,222 instances. These three data sets contain the fewest number of variables of any we tested.

The MUSK data set has the largest number of variables, making it a challenge to find any kind of meaningful network on. With 168 variables and 6,598 instances, this is typical of the size of the data sets we would like to consider, in columns if not in rows. The data is biochemical, the task being to classify indicated molecules as musk or non-musk given "distance features"[6], making all but one variable continuous (the last, its class, is binary). We quantize all of this data set's variables down to 5 quantiles to make the problem of scoring tractable, but even this simplification proves to be too much for an ADTree to handle (see section 6.4).

Because of space considerations, we only present results from the ADULT1 and MUSK data sets here, but other data sets of intermediate sizes were tested as well, and their results were consistent with those presented here.

6.2 Time and Score Comparisons

As an initial experiment, we searched for networks on the ADULT1 data set, giving the approximator varying numbers of training networks to use. The ADTree-based search, without the help of an approximator, took 86.21 seconds to create the tree and then 102.1 to perform the search, resulting in a final score of -1.7362×10^5. This should be compared to the results shown in Table 1. The upper half of the table contains the results of using the ADTree to score the training samples (the number of which is given along the top); because of this, the total time is reported including the time it took to create the tree (which is the same one used to do the ADTree-only search). Note that even with 50 samples, both the time and score are superior. As one would expect, the time it takes to generate and score the training graphs (reported as "gen. time") is linear in the number of graphs, while the time taken to tune the weights of the approximator grows more sharply. However, because it never takes that many training graphs to get reasonable results, the weighting time never becomes problematic. The lower half of the table contains results that were trained using the Bayes Net Toolkit[13]; the generation times are correspondingly longer, but without the overhead of creating the ADTree, the total times end up faster. A similar pattern emerges here, although the drop in scores between 250 and 500 samples is surprising. However, this is an example of the kind of overfitting that results from having too many samples.

6.3 Performance on Large Data Sets

First, a caveat about the meaning of the word "large". There are three independent ways which a data set could be called "large" — either it has many instances, or it has many variables, or the variables have a

	50	100	250	500
score (AD)	-1.592 ± 0.093	-1.539 ± 0.038	-1.557 ± 0.041	-1.553 ± 0.049
gen. time	1.71 ± 0.31	3.46 ± 0.29	8.64 ± 0.32	17.37 ± 0.55
weight time	0.07 ± 0.03	0.18 ± 0.12	1.26 ± 1.06	8.88 ± 8.57
search time	0.19 ± 0.03	0.37 ± 0.17	1.53 ± 1.08	9.82 ± 8.52
total + tree	88.19 ± 0.36	90.23 ± 0.39	97.65 ± 2.26	122.29 ± 16.78
score (BNT)	-1.565 ± 0.067	-1.578 ± 0.145	-1.510 ± 0.067	-1.618 ± 0.186
gen. time	3.89 ± 0.29	7.33 ± 0.52	17.67 ± 0.16	39.85 ± 2.02
weight time	0.09 ± 0.03	0.28 ± 0.26	1.31 ± 1.25	12.11 ± 6.69
search time	0.22 ± 0.03	0.47 ± 0.28	1.58 ± 1.25	13.35 ± 6.33
total	4.20 ± 0.27	8.07 ± 0.65	20.56 ± 2.58	65.30 ± 13.74

Table 1: ADULT1 numeric results

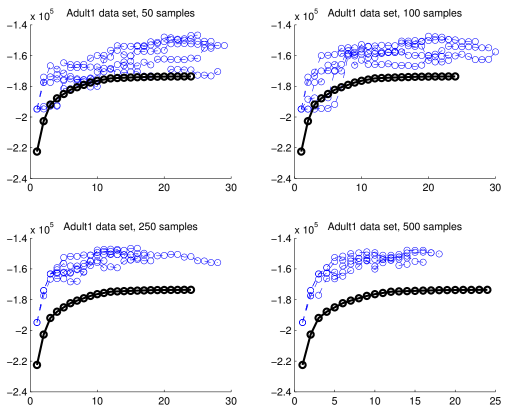

Figure 1: Trajectories for 50 to 500 samples, ADULT1 data set

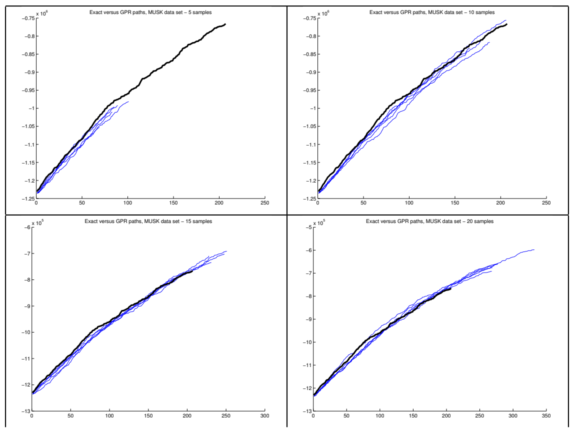

Figure 2: Trajectories for 5 to 20 samples, MUSK data set

samples	10	20	30	40	50	60
gen. time	29.4±1.3	60.3±3.0	93.2±2.3	120.7±5.4	146.7±5.1	175.4±0.8
weight time	0.18±0.11	0.21±0.06	0.33±0.19	0.32±0.17	0.39±0.10	1.57±0.46
search time	194.4±19.3	328.5±34.4	376.7±21.6	394.5±25.2	469.6±32.0	489.6±17.1
total	224.0±20.5	389.0±33.1	470.2±23.6	515.44±30.0	616.7±30.4	666.5±17.3
mean score($\times 10^5$)	−8.103	−6.573	−6.239	−6.082	−5.647	−5.579

Exact-scoring search: 3907 seconds, final score -7.659×10^5

Table 2: Timing results for 10 to 50 samples, Musk data set

samples	100	150	200	400	800
gen. time	304.8±7.9	448.5±11.3	604.8±15.8	1341.1±128.3	2346.6±8.9
weight time	2.3±1.7	7.9±3.6	13.4±6.8	81.9±38.4	401.5±226.9
search time	475.8±35.2	478.3±13.0	474.6±19.3	578.9±50.9	926.5±218.6
total	782.9±37.5	934.7±17.7	1092.8±25.5	2002.0±162.9	3674.7±447.2
mean score($\times 10^5$)	−5.819	−5.762	−5.765	−6.029	−6.149

Exact-scoring search: 3907 seconds, final score -7.659×10^5

Table 3: Timing results for 100 to 800 samples, Musk data set

high arity, thereby making the conditional probability tables large. The first sense of large is the one which ADTrees are good at handling; once a tree is built, it doesn't matter how many instances the original data had, since the tree only stores numeric counts. However, if there are many variables, or the variables have a high arity, the size of tree grows exponentially, as stated before.

Because of the sheer size of the Musk data set, creating an ADTree would be impossible. The size of an ADTree, even with most-common-value pruning, is still exponential. As an approximation, the size of such a tree is roughly $(r-1)^n$ for n nodes of arity r. If we have 168 nodes all of arity 5, as in the Musk data set, the size of tree would be 4^{168} or about 1.4×10^{101} internal nodes. For this reason, we have no ADTree results for this data set.

6.4 The Time/Score Trade-off Illustrated

Figure 2 shows the results of changing the number of samples while searching for graphs on the Musk data set. Because creating an ADTree on this data set would take far too much time and memory to be practical, the darker lines indicate a search using exact scoring. The lighter lines, meanwhile, show that even with a relatively small number of samples — 15 or 20 — we can achieve results that beat the exact search in score. As we increase the number of samples, one would expect the scores to improve. They do, but only up to a point; the best score is achieved with 60 samples, and then the scores start to decrease gradually while the times, as expected, continue to increase. This unexpected result is most likely due to overfitting.

Figure 3 is a more graphic demonstration of the overfitting, plotting the results from the Adult1 experiment with time on one axis and score on the other; each oval represents a different number of samples, with its center at the mean position of the five trials and the two axes corresponding to a single standard deviation in either direction. The pattern quickly becomes apparent; as we increase the time taken (by increasing the number of samples), the score increases as well, but only up to a point before it starts to drop again. Also note that, because of the nature of the axes, lower and to the right are both "better"; the result from the pure ADTree-based search, if plotted on these axes, would be off the chart to the top left.

6.5 Performance Boost as a Function of Smoothness

We hypothesize that the reason for the accelerator's performance is not only that it speeds up the calculation of scores, but also smooths out the search landscape, letting techniques such as greedy search get less confused by local optima and proceed more straightforwardly toward the desired results.

Note that the approximation does not appear to be a very good one at first; the search quickly proceeds

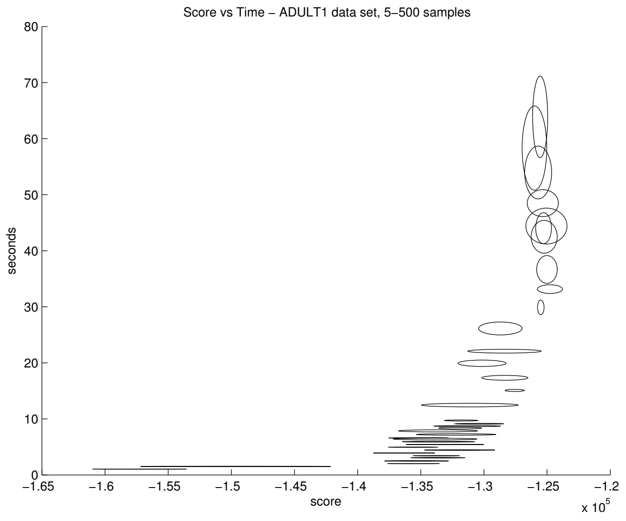

Figure 3: Time as a function of score, ADULT1 data set

into areas of very high positive score, which should be impossible by the definition of our scoring function. However, the actual scores are secondary — what we care about is ultimately the network that results from a search in this landscape, and this is where the approximator is helpful; even with very few data points, the gradients are captured well enough to push the search in the right direction. We exploit the overall smoothness offered by the regressor and let it drive us out of potential pitfalls. This also could explain the overfitting behavior; simply increasing the number of training samples will eventually lead to the final score leveling off and, in some cases, dropping.

7 Conclusion and Future Work

One obvious problem is the question of how to tell if we are in fact at a global optimum and not at either some crude approximation to one, given that our regressor function smooths out local features. This could be handled by adding a new subroutine to the search; if we've reached what looks like an optimum, sample more random points in its vicinity to retrain the approximator and let it explore that neighborhood more accurately. This can even be done recursively to get more and more detail in desired areas, down to the limit of our "vicinity" simply being all one-step changes from our graph, at which point we can simply select the best of those and call the process finished.

Another avenue of exploration is a setting where we wish to search over networks given incomplete data (where some entries in our data matrix D are simply missing or unknown). This is known to be a much harder problem[15], but if we can at least generate some kind of score for a group of sampled graphs, we should be able to proceed as before; once we have our approximator, the original data, complete or not, is irrelevant. Entire variables could be missing (the so-called latent variable problem[2]) and we would still be able to search efficiently.

Further work could also examine the structure of the kernel function itself; different kernel functions impose different structure on the space of graphs and make different assumptions on how functions will be smooth over that space; although the kernel we use is appealingly simple for its structure and ease of use with graphs represented as bit strings, there might be other more sophisticated methods of producing a kernel that would work better. It seems possible, for instance, that using a parameter-free kernel such as the thin plate spline might reduce the time needed by eliminating the need to train kernel parameters, although the weighting component of the search is what usually takes the least amount of time except in cases of large numbers of samples, and then the overfitting problem would occur. It would also be useful to perform an analysis of how many samples you need to converge to a reasonable estimate of the surface before overfitting sets in by using facts about how quickly various estimators converge.

References

[1] B. Anderson and A. Moore. Adtrees for fast counting and for fast learning of association rules. 310, 1998.

[2] C. M. Bishop. Latent variable models. *Learning in graphical models*, 1998.

[3] D. Chickering, D. Geiger, and D. Heckerman. Learning bayesian networks: Search methods and experimental results. pages 112–128, 1995.

[4] D. M. Chickering. Learning bayesian networks is np-complete. *Learning from data: Artificial intelligence and statistics v*, 112:121–130, 1996.

[5] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *Information Theory, IEEE Transactions on*, 14(3):462–467, 1968.

[6] Thomas G. Dietterich, Ajay N. Jain, Richard H. Lathrop, and Thomás Lozano-Pérez. A comparison of dynamic reposing and tangent distance for drug activity prediction. In *NIPS*, page 216, 1993.

[7] A. Frank and A. Asuncion. Uci machine learning repository, 2010.

[8] N. Friedman and D. Koller. Being bayesian about network structure. a bayesian approach to structure discovery in bayesian networks. *Machine Learning*, 50(1):95–125, 2003.

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

[10] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[11] B. T. Lowerre. The harpy speech recognition system. 1976.

[12] A. Moore and M. S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Arxiv preprint cs/9803102*, 1998.

[13] K. Murphy et al. The bayes net toolbox for matlab. *Computing science and statistics*, 33(2):1024–1034, 2001.

[14] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[15] M. Ramoni and P. Sebastiani. Robust learning with missing data. *Machine Learning*, 45(2):147–170, 2001.

[16] C. E. Rasmussen. Gaussian processes in machine learning. *Advanced Lectures on Machine Learning*, pages 63–71, 2004.

[17] R. Robinson. Counting unlabeled acyclic digraphs. *Combinatorial mathematics V*, pages 28–43, 1977.

[18] B. Yackley, E. Corona, and T. Lane. Bayesian network score approximation using a metagraph kernel. *In Advances in Neural Information Processing Systems*, 20, 2008.