Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence

Memory-Efficient Dynamic Programming for Learning Optimal Bayesian Networks

Brandon Malone and **Changhe Yuan** and **Eric A. Hansen**
Department of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
bm542@msstate.edu, {cyuan, hansen}@cse.msstate.edu

Abstract

We describe a memory-efficient implementation of a dynamic programming algorithm for learning the optimal structure of a Bayesian network from training data. The algorithm leverages the layered structure of the dynamic programming graphs representing the recursive decomposition of the problem to reduce the memory requirements of the algorithm from $O(n2^n)$ to $O(C(n, n/2))$, where $C(n, n/2)$ is the binomial coefficient. Experimental results show that the approach runs up to an order of magnitude faster and scales to datasets with more variables than previous approaches.

Introduction

We consider the combinatorial problem of finding the highest-scoring Bayesian network structure from data. Given n random variables and a set of N observations of each of the n variables, the problem is to infer a directed acyclic graph on the n variables such that the implied joint probability distribution best explains the set of observations. The problem is known to be NP-hard (Chickering 1996).

Several exact algorithms based on dynamic programming have been developed for learning the optimal structure of a Bayesian network (Koivisto and Sood 2004; Singh and Moore 2005; Silander and Myllymaki 2006). In the dynamic programming approach, the basic idea is to find small optimal subnetworks and use them to find larger optimal subnetworks until an optimal network for all of the variables has been found. Both the time and space complexity of the approach is $O(n2^n)$. But in practice, it is the space complexity that primarily limits scalability.

There are other exact approaches to the structure-learning problem. For example, de Campos *et al.* (2009) recently proposed a systematic search algorithm to identify optimal network structures. The algorithm begins by calculating optimal parent sets for all variables. These sets are represented as a directed graph that may have cycles. Cycles are then repeatedly broken by removing one edge at a time. The algorithm terminates with an optimal Bayesian network, but it is often less efficient than dynamic programming.

In this paper, we show how to improve the efficiency of the dynamic programming approach. In particular, we show how to leverage the layered structure of the problem to reduce the memory requirement of the dynamic programming algorithm from $O(n2^n)$ to $O(C(n, n/2))$. We also show that theoretical properties of the MDL scoring function can be used to reduce the average runtime necessary to find an optimal structure by up to an order of magnitude compared to the previous best approach.

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

A Bayesian network consists of a directed acyclic graph (DAG) structure and a set of parameters. Each vertex in the DAG corresponds to a random variable, $X_1, X_2, ..., X_n$. If X_i is a parent of X_j, then X_j depends directly on X_i. The parents of X_j are referred to as PA_j. A variable is conditionally independent of its non-descendants given its parents. The parameters of the network specify a conditional probability distribution, $P(X_j|PA_j)$ for each X_j.

Given a dataset $\mathbf{D} = \{D_1, ..., D_N\}$, where each D_i is an instantiation of all variables $\mathbf{V} = \{X_1, ..., X_n\}$, the structure learning problem consists of finding a network DAG structure over the variables that best fits $\mathbf{D}$ (Heckerman 1995). The fit of a structure to the dataset can be measured using a scoring function, such as the *minimum description length* (MDL) score (Rissanen 1978). The MDL scoring function captures the tradeoff between fit to data and network complexity with two terms. The first is an entropy-based term, while the second penalizes more complex structures. Let r_i be the number of states of the variable X_i, let N_{pa_i} be the number of data records consistent with $PA_i = pa_i$, and let N_{x_i,pa_i} be the number of data records consistent with $PA_i = pa_i$ and $X_i = x_i$. Then the MDL score for a structure G is defined as follows (Rissanen 1978),

$$MDL(G) = \sum_i MDL(X_i|PA_i), \tag{1}$$

where

$$
\begin{aligned}
MDL(X_i|PA_i) &= H(X_i|PA_i) + \frac{\log N}{2} K(X_i|PA_i), \\
H(X_i|PA_i) &= -\sum_{x_i, pa_i} N_{x_i,pa_i} \log \frac{N_{x_i,pa_i}}{N_{pa_i}} , \\
K(X_i|PA_i) &= (r_i - 1) \prod_{X_l \in PA_i} r_l .
\end{aligned}
$$

MDL is decomposable (Heckerman 1995); that is, the score for the entire structure is calculated by summing over each variable. Our algorithms can be adapted to use any decomposable scoring function. The following theorems due to Tian (2000) and de Campos *et al.* (2009) provide a basis for ignoring some sets of parents when searching for an optimal set of parents for a variable with the MDL scoring function.

Theorem 1. *In an optimal Bayesian network based on the MDL scoring function, each variable has at most* $\log(\frac{2N}{\log N})$ *parents, where* N *is the number of records.*

Theorem 2. *Let* $\mathbf{U} \subset \mathbf{V}$ *and* $X \notin \mathbf{U}$. *If* $BestMDL(X, \mathbf{U}) < BestMDL(X, \mathbf{V})$, $\mathbf{V}$ *cannot be the optimal parent set for* X.

Dynamic programming

Dynamic programming algorithms find an optimal Bayesian network structure in $O(n2^n)$ time and memory (Koivisto and Sood 2004; Singh and Moore 2005; Silander and Myllymaki 2006). The algorithms derive from the observation that the optimal network structure is a DAG. Therefore, it consists of an optimal leaf vertex with its parents plus an optimal subnetwork. This subnetwork is also a DAG. The algorithm recursively finds optimal leaves of subnetworks to find the optimal network structure. The following recurrence expresses this procedure for the MDL scoring function and variables $\mathbf{V}$ (Singh and Moore 2005).

$$
\begin{aligned}
MDL(\mathbf{V}) &= \min_{X \in \mathbf{V}} \{ MDL(\mathbf{V} \setminus \{X\}) + \\
&\qquad BestMDL(X, \mathbf{V} \setminus \{X\}) \},
\end{aligned}
$$

where

$$
BestMDL(X, \mathbf{V} \setminus \{X\}) = \min_{PA_X \subseteq \mathbf{V} \setminus \{X\}} MDL(X|PA_X).
$$

While this recurrence does suggest an algorithm, a more computationally amenable formulation (Silander and Myllymaki 2006) begins with a 0-variable subnetwork. Leaves are recursively added to the subnetworks until all variables have been added and the optimal network has been found.

These algorithms require solving two distinct tasks. First, for the set of candidate parent variables for each variable, the optimal subset of the parent variables must be calculated. Second, the sets of optimal parents are then used to identify optimal subnetwork structures until the optimal structure over all variables is identified.

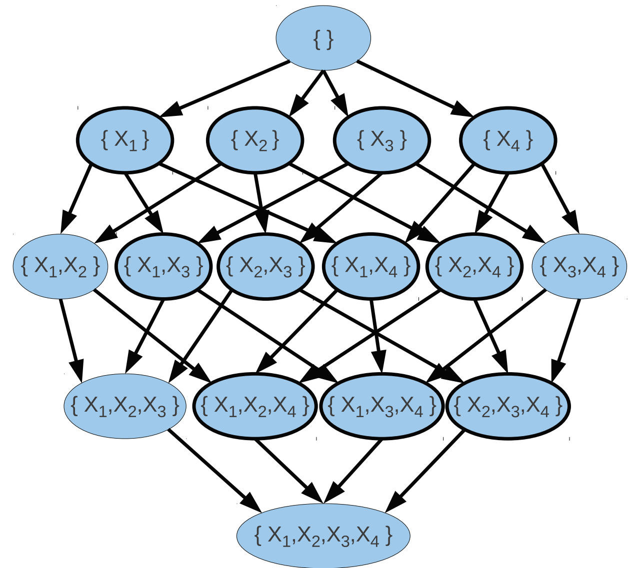

Figure 1: An order graph of four variables

Order graph and parent graphs

We can use two kinds of graphs to represent the recursive decomposition of the problem of finding the optimal network structure using dynamic programming. First consider what we call an *order graph*. For a problem with n variables, the order graph has 2^n nodes representing all subsets of the variables. Figure 1 shows an order graph for a problem with n variables. Each node of the graph represents the subproblem of finding an optimal network for the subset of variables corresponding to that node. Note that the order graph is a lattice that organizes the nodes into layers. All nodes in the same layer correspond to subnetworks of the same size. All nodes in layer l have l predecessors in the previous layer. Layer l has $C(n, l)$ nodes where $C(n, k)$ is the binomial coefficient. The second layer of the order graph ($l = 1$) consists of one single node subnetwork for each variable. The variable has no parents, and its score is equal to the score for the variable given no parents, $MDL(X_i|\{\})$.

Each node in the order graph chooses its predecessor which optimizes the MDL score for that subnetwork. Thus to evaluate the node for variables $\mathbf{O}$ in the order graph, we try each $X \in \mathbf{V} \setminus \mathbf{O}$ as a leaf and $\mathbf{O}$ as the subnetwork. The score for the new subnetwork, $MDL(\mathbf{O} \cup X)$, is $MDL(\mathbf{O}) + BestMDL(X, \mathbf{O})$. The optimal leaf, its parents and score are kept for each subnetwork. After considering all of node $\mathbf{O}$'s predecessors in the order graph, the node contains the optimal subnetwork for the corresponding set of variables.

We call it an *order graph* because a path from the root of the order graph to its single leaf node represents an order in which variables are added to an initially empty network to create a network over all variables. This ordering partially determines the structure of a network. The rest of the structure is determined by the set of parents for each node. To find the optimal set of parents for a node, we use what we

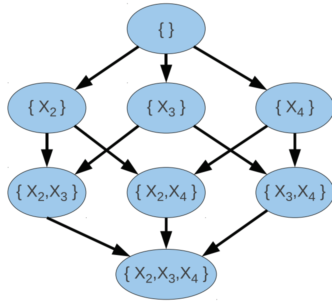

Figure 2: Parent graph for candidate parents $\{X_2, X_3, X_4\}$

call a *parent graph* to calculate $BestMDL(X, \mathbf{U})$. There is one parent graph for each variable. Figure 2 shows an example of a parent graph for variable X_1. The graph, which is also a lattice, contains all subsets of all $n-1$ other variables. Thus, it contains 2^{n-1} nodes. Each node in the parent graph represents a mapping from a candidate set of parents to the subset of those variables which optimizes the score of X_i, as well as that score. The lattice organizes the nodes naturally into layers. All nodes in the same layer consider the same number of possible parents. Additionally, all nodes in layer l have l predecessors in the previous layer. Layer l has $C(n-1, l)$ nodes. At the first layer of the construction ($l = 0$), the score of X_i given no parents is computed using the MDL scoring function. Trivially, this is the best score X_i can attain with no parents. Thus the graph contains a node which maps from the set with no variables to the score.

To evaluate the node for candidate parents $\mathbf{P}$ of X, we consider $BestMDL(X, \mathbf{P} \setminus \{Y\})$ for each $Y \in \mathbf{P}$, as well as $MDL(X|\mathbf{P})$. The best of these is the score for the new node, $BestMDL(X, \mathbf{P})$. $MDL(.|.)$ is stored in a score cache, either in a hash table in RAM or written to disk in an order which allows quick access depending upon available memory resources. The minimum score is kept for each candidate parent set. This follows from Theorem 2. After considering all of a node $\mathbf{P}$'s predecessors, it contains the subset of variables which minimize the score of X from among those candidate parents.

Memory-Efficient Dynamic Programming

We next make the observation that, because the parent and order graphs naturally partition into layers, only a limited amount of information is needed to evaluate each layer. Evaluating a layer in the parent graph requires the previous layer of the parent graph, while evaluating a layer in the order graph requires the previous layer of the order graph and the current layer of the parent graphs. Because the scores and optimal parent information are propagated from one layer to the next, a layer can be deleted once its successor layer has been evaluated. Thus we do not need to keep the entire graphs in memory. The layers of a graph can be generated as needed and then deleted from memory when no longer needed. Reconstructing the network structure simply requires we store the leaf, its optimal parent set and a pointer to its predecessor for each order graph node.

Algorithm 1 Memory-Efficient Dynamic Programming

```
1:  procedure GENERATELAYER(orderGraph)
2:      for each parentGraph pg do
3:          generateParentLayer(pg)
4:      end for
5:      for each node O ∈ prev do
6:          for each v ∈ V − O do
7:              pg ← parentGraphs[v].readNode()
8:              score ← pg.score + O.score
9:              if score < curr[O ∪ v].score then
10:                 curr[O ∪ v] ← {O.pars ∪ pg.pars, score}
11:             end if
12:             if v = Y_1 then write(curr[O ∪ v])
13:         end for[each v]
14:     end for[each node O]
15:     prev ← curr
16:     curr ← new HashTable
17: end procedure

18: procedure GENERATEPARENTLAYER(pg)
19:     for each node P ∈ prev do
20:         for each v ∈ V − P and v ≠ pg.v do
21:             if curr[P ∪ v] is null then
22:                 curr[P ∪ v] ← {P ∪ v, score(P ∪ v)}
23:             end if
24:             if P.score < curr[P ∪ v].score then
25:                 curr[P ∪ v] ← P
26:             end if
27:             if v = p_1 then write(curr[P ∪ v])
28:         end for[each v]
29:     end for[each node P]
30:     prev ← currentLayer
31:     curr ← new HashTable
32: end procedure

33: procedure MAIN
34:     for l = 1 → n do
35:         generateLayer(orderGraph)
36:     end for
37: end procedure
```

Algorithm 1 gives pseudocode for a memory-efficient layered dynamic programming algorithm. The main idea is to generate one layer of the parent and order graphs at a time and delete old layers when they are no longer needed. By storing only one layer at a time, we reduce the memory requirement for the algorithm from $O(n2^n)$ to $O(C(n, n/2))$ plus the portion of the order graph necessary for network reconstruction. However, the memory requirement can still grow quickly. We resort to external memory to further lighten the burden on RAM. Note that evaluating order graph node $\mathbf{O} \cup X$ requires $BestMDL(X, \mathbf{O})$. This score is stored in node $\mathbf{O}$ of the parent graph for X. Once $\mathbf{O} \cup X$ has been evaluated, $BestMDL(X, \mathbf{O})$ is not used in any other calculations, so it can be deleted.

Naively, this step would still require the current layer of all of the parent graphs as well as the previous and current

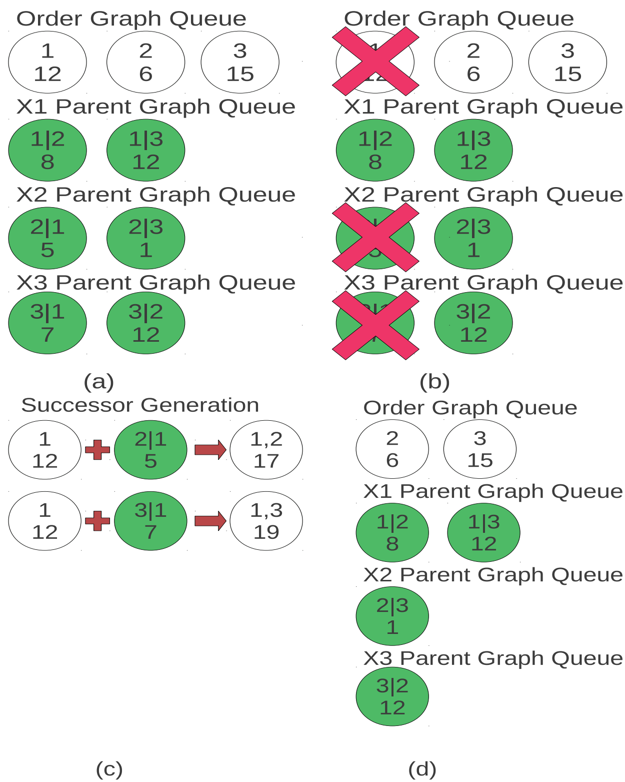

Figure 3: Generating successors of a node in the order graph. The top half of the order graph nodes (in white) is the subnetwork; the bottom half is the score of that subnetwork. The top half of the parent graph nodes (shaded) is the variable and the candidate set of parents; the bottom half is $BestMDL$ for that variable and candidate parent set. (a) The starting queues. (b) The popped off nodes. (c) The generated successors which are stored in the next layer. (d) The new queues.

layers of the order graph in RAM at once in order to quickly look up the scores for the predecessor nodes of $\mathbf{O}$. By carefully arranging the nodes in the order and parent graphs, only the current layer of the order graph must reside in RAM. Everything else can be stored on disk.

External-Memory Dynamic Programming

We arrange the nodes of each layer of the order and parent graphs in sorted files to ensure we never require random access to nodes during node expansions. We treat the files as queues in which the sequence of nodes is coordinated such that when a node $\mathbf{O}$ is removed from the order graph queue, the node at the head of each parent graph queue $X \in \mathbf{V} \setminus \mathbf{O}$ is the node for candidate parents $\mathbf{O}$. That is, the head of each of those parent graph queues is $BestMDL(X, \mathbf{O})$, while the head of the order graph queue is $MDL(\mathbf{O})$. Given these assumptions, we generate the successors of $\mathbf{O}$ in the next layer of the order graph by removing it and the head of each of the parent graphs in $\mathbf{V}\backslash\mathbf{O}$ from their respective queues. Each parent graph node is used to generate a successor of $\mathbf{O}$. Figure 3 shows an example of generating successors of an order graph node. The generated successors are stored in the next layer, while the remaining queues are used to generate successors for the other order graph nodes in the current layer. After all successors have been generated for a layer, we write the nodes to disk according to the ordering. An analogous approach can be used to generate the parent graphs. The RAM usage for layer l is $O(C(n,l))$ for storing the order graph layer.

Lexicographic Ordering of Sets of Combinations

The lexicographic ordering of subsets of $\mathbf{V}$ with size l is one sequence that ensures all of the necessary parent graph queues have the correct node at the head of their queue at layer l when $\mathbf{O}$ is removed from the order graph queue (file). For example, the ordering for 4 variables of size 2 is $\{\{X_1, X_2\}, \{X_1, X_3\}, \{X_2, X_3\}, \{X_1, X_4\}, \{X_2, X_4\}, \{X_3, X_4\}\}$. Thus, the order graph queue for layer 2 with 4 variables should be arranged in that sequence. Knuth (2009) describes an algorithm for efficient generation of this sequence. The parent graph queue for variable X_i should have the same sequence as the order graph, but without subsets containing X_i. The sequence of the parent graph queue for variable X_1 from the example is $\{\{X_2, X_3\}, \{X_2, X_4\}, \{X_3, X_4\}\}$ (the same as the order graph queue, but without subsets containing variable X_1).

The sequence has two other very helpful properties for further minimizing RAM usage. First, the optimal score and network for subnetwork $\{X_1 ... X_l\}$ have been calculated after computing the score using X_1 as the leaf of the subnetwork. This is because the last predecessor used to generate that subnetwork in the ordering contains variables $\{X_2 \dots X_l\}$. Second, the nodes in layer l of a graph are completed in the correct lexicographic order. A node is completed after expanding its last predecessor. As a result, a node can be written to disk and removed from RAM once it is completed, and the next layer will be sorted correctly.

Reconstructing the Optimal Network Structure

In order to reconstruct the optimal network structure, we permanently store a portion of each order graph node, including the subnetwork variables, the leaf variable and its parents, on disk. Solution reconstruction works as follows. The goal node contains the final leaf variable X and its optimal parent set. Its predecessor in the shortest path is $\mathbf{O} = \mathbf{V} \backslash \{X\}$. This predecessor is retrieved from the file for layer $|\mathbf{O}|$. Recursively, the optimal leaves and parent sets are retrieved until reconstructing the entire network structure.

Advantages of Memory-Efficient Dynamic Programming

Our layered version of dynamic programming is more memory-efficient and faster than other approaches, such as those presented by Singh and Moore (2005) or Silander and Myllymaki (2006), for two key reasons.

First, the layered structure we impose on the parent and order graphs ensures that we never need more than two layers of any of the graphs in memory, RAM or files on disk,

at once. None of the existing algorithms take advantage of the structure in the parent and order graphs when calculating $BestMDL(X, \mathbf{V})$ or $MDL(\mathbf{V})$. Singh uses a depth-first search ordering to generate the necessary scores and variable sets, while Silander uses the lexicographic ordering over all of the variables. (We use the lexicographic ordering only within each layer, not over all of the variables.) The depth-first approach does not generate nodes in one layer at a time. The lexicographic ordering also does not generate all nodes in one layer at a time. Consider the first four nodes in the lexicographic order: $\{X_1\}, \{X_2\}, \{X_1, X_2\}$ and $\{X_3\}$. Two nodes from layer 1 are generated, then a node in layer 2; however, the next node generated is again in layer 1. Similarly, the seventh node generated is $\{X_1, X_2, X_3\}$ while the eighth node is $\{X_4\}$. In contrast to the ordering we propose, this ordering expands $\{X_1, X_2\}$ before expanding $\{X_3\}$. Because generation of nodes from different layers is interleaved, these orderings require the entire graphs remain in memory (either in RAM or on disk). In contrast, our dynamic programming algorithm generates nodes one layer at a time and thus needs at most two layers of the graphs in memory, plus the extra information to reconstruct the path. Previous layers can safely be deleted.

Second, Theorem 1 coupled with an AD-tree search ensures that scores for large parent sets are never required. We calculate scores using a top-down, AD-tree-like method (Moore and Lee 1998). For example, we calculate the score of a variable given no parents before the score of a variable given one parent. Because of this approach and Theorem 1, we never need to calculate the scores, or even the counts, of large sets of variables. In fact, for a variable set with n variables and N records, we compute and store $n(2^{\log \frac{2N}{\log N}})$ scores. In contrast, methods that compute scores using a bottom-up approach must score all $n2^{n-1}$ possible parent sets. Even if these methods do employ Theorem 1, they must still consider all of the possible sets to calculate the counts required for the smaller parent sets.

Experiments

Memory-efficient dynamic programming (MEDP) implemented in Java was compared to an efficient implementation of dynamic programming which uses external memory written in C (Silander and Myllymaki 2006). We downloaded Silander's source code from http://b-course.hiit.fi/bene and refer to it as SM. Previous results (Silander and Myllymaki 2006) have shown SM is much more efficient than previous dynamic programming implementations. The algorithms were tested on benchmark datasets from the UCI repository (Frank and Asuncion 2010). The largest datasets have up to 30 variables and over 30,000 records. We discretized all continuous variables and discrete variables with more than four states into two states around the mean values; Records with missing values were removed. All algorithms were given a maximum running time of one day (86,400 seconds) and a maximum hard disk space of 150 gigabytes.

Although SM uses BIC instead of MDL, the calculations are equivalent and the learned networks were always equivalent. The experiments were performed on a 2.66 GHz Intel Xeon with 16 gigabytes of RAM running SUSE version 10.

We evaluated both the space and time requirements of the algorithms. We compared the size of the full order and parent graphs, which a typical dynamic programming implementation must store, to the maximum sizes of the graphs with MEDP. We also measured the running time of the algorithms. Table 1 gives the timing and space results.

The memory comparisons highlight the advantages of MEDP. Comparison between the sizes of the external files shows that the layered algorithm typically stores an order of magnitude less information on disk than SM. This also agrees with the theoretical behavior of working with only a layer at a time instead of the entire graph. The layered approach of MEDP permits deletion of files for old layers, so the external-memory version never uses as much disk space as SM. Because dynamic programming is systematic, the maximum size of the files stored by MEDP is the same for a given n. Thus, memory usage is unaffected by N.

We used the $wdbc$ dataset to test the scalability of the algorithms to larger sets of variables. The total dataset has 31 variables. We began by taking a subset of all of the variables in the dataset and learning optimal networks. We then added one variable at a time to the subset and again learned optimal networks. The first n variables of the full dataset were used.

As the scalability results show, MEDP always learns the optimal network with the time and space constraints; however, for 28 and 29 variables, SM ran for a day without finding it. Additionally, for 30 variables, SM consumed more than the available 150gb of hard disk space.

The timing results show that MEDP usually runs several times faster than SM. SM does run faster on two of the datasets with a large number of records and a modest number of variables. The larger number of records reduces the effectiveness of the pruning offered by Theorem 1. MEDP is faster when Theorem 1 can be applied more often. As the number of variables increases, the record count affects runtime less. For the segment (2,000 records) and mushroom (8,000 records) datasets, MEDP runs faster than SM.

Conclusion

The graphical structure of the parent and order graphs make them particularly well suited to our layered formulation of the optimal Bayesian network structure learning problem. We take advantage of the regular and clearly defined relationships between the layers of the graphs to reduce the memory complexity of the problem from $O(n2^n)$ to $O(C(n, n/2))$. In particular, the parent graphs require the previous layer of the parent graph, while the order graph needs both the previous layer of the order graph and the current layer of the parent graphs to generate the next layer. Hence, unnecessary layers can easily be deleted. Freeing the memory increases the size of learnable networks. We also apply theoretical properties of the MDL scoring function and calculate scores in a top-down manner to significantly improve the running time of the algorithm.

Our work can be used to compare approximate structure learning algorithms by using the learned optimal network as a "gold standard." Structures learned using approximate

Dataset			Timing Results (s)		Space Results (bytes)	
dataset	n	N	SM	MEDP	SM	MEDP
wine	14	178	1	0	1.16E+07	4.57E+05
adult	14	30,162	4	15	1.16E+07	4.57E+05
zoo	17	101	4	2	4.81E+07	4.01E+06
houseVotes	17	435	16	4	4.81E+07	4.01E+06
letter	17	20,000	66	131	4.81E+07	4.01E+06
statlog	19	752	73	12	1.82E+08	1.67E+07
hepatitis	20	126	63	15	3.79E+08	3.34E+07
segment	20	2,310	70	36	3.79E+08	3.34E+07
meta	22	528	227	78	1.67E+09	1.39E+08
imports	22	205	315	75	1.67E+09	1.39E+08
horseColic	23	300	1,043	170	3.48E+09	2.88E+08
heart	23	267	1,024	171	3.48E+09	2.88E+08
mushroom	23	8,124	1,473	537	3.48E+09	2.88E+08
parkinsons	23	195	714	169	3.48E+09	2.88E+08
wdbc23	23	569	1,516	196	3.48E+09	2.88E+08
wdbc24	24	569	3,332	413	7.26E+09	5.76E+08
wdbc25	25	569	5,638	931	1.51E+10	1.19E+09
wdbc26	26	569	16,127	1,985	3.15E+10	2.38E+09
wdbc27	27	569	36,563	4,644	6.29E+10	4.91E+09
wdbc28	28	569	OT	11,924	-	9.83E+09
wdbc29	29	569	OT	24,350	-	1.97E+10
wdbc30	30	569	OM	78,055	-	3.93E+10

Table 1: A comparison on the running time (in seconds) for Silander's dynamic programming implementation (SM) and our memory-efficient dynamic programming algorithm (MEDP). The sizes of files written to hard disk are given. The size for SM is calculated at the end of the search. The size for MEDP is calculated at the end of each layer; the maximum size is given. Each line of the $wdbc$ datasets shows the performance of the algorithms as the variable count is increased. The column headings mean: 'n' is the number of variables; 'N' is the number of records; 'OT' means failure to find optimal solutions due to out of time (more than 24 hours, or 86,400 seconds); 'OM' means out of memory (more than 150 gigabytes).

techniques can be compared to the optimal structure to assess the quality of the approximate algorithms on smaller datasets. We can then extrapolate which approximate algorithms will learn better networks on larger sets of variables.

As the memory complexity analysis demonstrated, MEDP requires the most memory and running time while processing the middle layers. Future work could investigate methods to prune the order and parent graphs using an admissible heuristic. Further reduction of the memory requirements will continue to improve the scalability of the method and allow the learning of even larger optimal networks.

Acknowledgements This work was supported by NSF CAREER grant IIS-0953723 and EPSCoR grant EPS-0903787.

References

Chickering, D. M. 1996. Learning bayesian networks is np-complete. In *Learning from Data: Artificial Intelligence and Statistics V*, 121–130. Springer-Verlag.

de Campos, C. P.; Zeng, Z.; and Ji, Q. 2009. Structure learning of bayesian networks using constraints. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, 113–120. New York, NY, USA: ACM.

Frank, A., and Asuncion, A. 2010. UCI machine learning repository.

Heckerman, D. 1995. A tutorial on learning bayesian networks. Technical report, Microsoft Research.

Knuth, D. E. 2009. *The Art of Computer Programming, Volume 4, Fascicles 0-4*. Addison-Wesley Professional, 1st edition.

Koivisto, M., and Sood, K. 2004. Exact bayesian structure discovery in bayesian networks. 549–573–.

Moore, A., and Lee, M. S. 1998. Cached sufficient statistics for efficient machine learning with large datasets. *J. Artif. Int. Res.* 8:67–91.

Rissanen, J. 1978. Modeling by shortest data description. *Automatica* 14:465–471.

Silander, T., and Myllymaki, P. 2006. A simple approach for finding the globally optimal bayesian network structure. In *Proceedings of the 22nd Annual Conference on Uncertainty in Artificial Intelligence (UAI-06)*, –. Arlington, Virginia: AUAI Press.

Singh, A., and Moore, A. 2005. Finding optimal bayesian networks by dynamic programming. Technical report, Carnegie Mellon University.

Tian, J. 2000. A branch-and-bound algorithm for mdl learning bayesian networks.