Learning Optimal Bayesian Networks Using A* Search

Changhe Yuan[†], **Brandon Malone**[†], **and Xiaojian Wu**[‡]
[†]Department of Computer Science and Engineering, Mississippi State University
[‡]Department of Computer Science, University of Massachusetts, Amherst
cyuan@cse.msstate.edu, bm542@msstate.edu, xiaojian@cs.umass.edu

Abstract

This paper formulates learning optimal Bayesian network as a shortest path finding problem. An A* search algorithm is introduced to solve the problem. With the guidance of a consistent heuristic, the algorithm learns an optimal Bayesian network by only searching the most promising parts of the solution space. Empirical results show that the A* search algorithm significantly improves the time and space efficiency of existing methods on a set of benchmark datasets.

1 Introduction

Applying Bayesian networks to real-world problems typically requires building graphical representations of the problems. One popular approach is to use score-based methods to find high-scoring structures for given data [Cooper and Herskovits, 1992; Heckerman, 1998]. Since finding an optimal solution is rather difficult, early approaches are mostly approximation methods [Cooper and Herskovits, 1992; Friedman *et al.*, 1999; Heckerman, 1998]. Unfortunately, the models found by these methods are not guaranteed to be optimal and, even worse, may be of unknown quality.

Several exact algorithms based on dynamic programming have recently been developed to learn optimal Bayesian networks [Koivisto and Sood, 2004; Silander and Myllymaki, 2006; Singh and Moore, 2005]. The main idea is to solve small subproblems first and use the results to find solutions to larger problems until a global learning problem is solved. However, these algorithms may be inefficient due to their need to fully evaluate an exponential solution space.

A recent exhaustive search algorithm [de Campos *et al.*, 2009] first finds optimal parent sets for the individual variables by ignoring the acyclic constraint and then detects and breaks all the cycles to find a valid Bayesian network. This algorithm was shown to be often less efficient than dynamic programming. We believe a major reason for its inefficiency is the search space of this algorithm consists of directed *cyclic* graphs except the final solution; such a formulation makes the size of the search space unnecessarily large.

This paper presents a new approach that formulates learning optimal Bayesian networks as a shortest path finding problem. An A* search algorithm is introduced to solve the problem. With the guidance of a consistent heuristic, the algorithm is able to focus on searching the most promising parts of a solution space in finding its solution. Other parts of the solution space are safely pruned. The algorithm is shown to be able to find optimal Bayesian networks not only more efficiently but also in less space.

2 Learning Optimal Bayesian Networks

A Bayesian network is a directed acyclic graph (DAG) G that represents a joint probability distribution over a set of random variables $\mathbf{V} = \{X_1, X_2, ..., X_n\}$. A directed arc from X_i to X_j represents the dependence between the two variables; we say X_i is a parent of X_j. We use PA_j to stand for the parent set of X_j. The dependence relation between X_j and PA_j are quantified using a conditional probability distribution, $P(X_j|PA_j)$.

Given a data set $\mathbf{D} = \{D_1, ..., D_N\}$, where each D_i is a vector of values over variables $\mathbf{V}$, learning an *optimal* Bayesian network is the task of finding an optimal network structure that best fits $\mathbf{D}$. We use "an optimal" instead of "the optimal" because there may be multiple structures that have the same optimal score and belong to one equivalence class [Chickering, 2002]. The degree of the fitness of a network can be measured using scoring functions such as Minimum Description Length (MDL) [Rissanen, 1978]. Let r_i be the number of states of X_i, N_{pa_i} be the number of data points consistent with $PA_i = pa_i$, and N_{x_i,pa_i} be the number of data points further constrained with $X_i = x_i$. MDL is defined as follows [Tian, 2000].

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

where

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

The goal is then to find a Bayesian network that has the *minimum* MDL score. Although we only use MDL in our

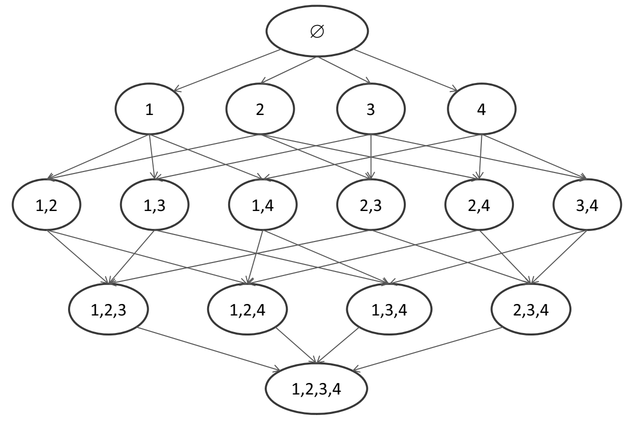


Figure 1: An order graph of four variables

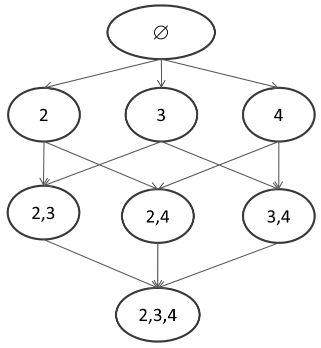


Figure 2: Parent graph for variable 1 with candidate parents $\{2, 3, 4\}$

algorithm, other scoring functions can potentially be used as well, including BIC, K2, BDeu, and BDe [Heckerman, 1998]. A common characteristic of all these scores is they are *modular* or *decomposable*, i.e., the score of a structure can be decomposed into a sum of node scores.

3 Dynamic Programming

There are many existing methods for learning optimal Bayesian networks, including both approximate and exact algorithms. This paper focuses on exact algorithms. In this section, we review in more detail the dynamic programming algorithm presented in [Singh and Moore, 2005].

The dynamic programming algorithm is based on the observation that a Bayesian network has at least one *leaf* [Singh and Moore, 2005]. In order to find an optimal Bayesian network for a set of variables $\mathbf{V}$, we can find the best leaf choice. For any leaf choice X, the best possible Bayesian network is constructed by letting X to choose an optimal parent set PA_X from $\mathbf{V}\backslash\{X\}$ and letting $\mathbf{V}\backslash\{X\}$ to form an optimal subnetwork. The best leaf choice X is then the one that maximize the sum of $MDL(X|PA_X)$ and $MDL(\mathbf{V}\backslash\{X\})$. More formally, we have:

$$MDL(\mathbf{V}) = \min_{X\in\mathbf{V}}\{MDL(\mathbf{V}\setminus\{X\}) + BestMDL(X, \mathbf{V}\setminus\{X\})\}, \tag{2}$$

where

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

Given the recurrence relation, the dynamic programming algorithm works as follows. It first finds optimal structures for single variables, which is trivial. Starting with these base cases, the algorithm builds optimal subnetworks for increasingly larger variable sets until an optimal network is found for $\mathbf{V}$. The dynamic programming algorithm can find an optimal Bayesian network in $O(n2^n)$ time [Silander and Myllymaki, 2006; Singh and Moore, 2005].

Figure 1 shows an *order graph* that visualizes how the dynamic programming algorithm solves a learning problem. It is called an order graph because any path from top to bottom in the graph is an ordering of the variables. For example, the path traversing nodes $\emptyset, \{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}$ stands for the variable ordering $1, 2, 3, 4$. Evaluating a variable ordering amounts to finding an optimal network that is consistent with the ordering. We can find a globally optimal Bayesian network by evaluating all the possible orderings. Dynamic programming evaluates all the orderings using a top down sweep of the order graph. Layer by layer, dynamic programming finds an optimal subnetwork for the variable set contained in each node of the order graph based on results from the previous layers. For example, there are three ways to construct a Bayesian network for node $\{1, 2, 3\}$: using $\{2, 3\}$ as subnetwork and 1 as leaf, using $\{1, 3\}$ as subnetwork and 2 as leaf, or using $\{1, 2\}$ as subnetwork and 3 as leaf. The top-down sweep makes sure that optimal substructures are already found for $\{2, 3\}$, $\{1, 3\}$, and $\{1, 2\}$. We only need to find optimal parents for the leafs in the three cases, from which we can find an optimal network for $\{1, 2, 3\}$. Once the evaluation reaches the node in the last layer, an optimal Bayesian network is found for the global variable set.

We use *parent graph* to find optimal parents for a variable X out of a candidate set. A parent graph for a variable X is similar to an order graph except that it only contains variables $\mathbf{V}\backslash\{X\}$. For example, the parent graph of variable 1 with potential parents $\{2, 3, 4\}$ is shown in Figure 2. If we need to find optimal parents for 1 out of $\{2, 3\}$, we find a subset of $\{2, 3\}$ that has the best parent score. There is a parent graph for each variable.

4 An A* Search Algorithm

A critical drawback of the dynamic programming approach is its need to find optimal subnetworks for all subsets of the variables, which in turn requires computing all the parent scores for each variable. For n variables, there are 2^n nodes in the order graph, and n parent graphs with 2^{n-1} parent nodes each. Totally, $n2^{n-1}$ parent scores need to be computed. As the number of variables increases, computing and storing the order and parent graphs quickly becomes infeasible.

We present an A* search algorithm for learning optimal Bayesian networks. We first introduce the formulation of the algorithm. We then discuss the pruning techniques we used to improve the search efficiency. We end this section with a brief summary of the advantages of our algorithm.

4.1 The formulation

The basic idea of our algorithm is to formulate learning optimal Bayesian networks as a *shortest path finding* problem. We use the order graph in Figure 1 as the search graph. We let the top-most node that contains no variables be the *start state* and the bottom-most node with all variables be the *goal state*. For any two neighboring nodes $\mathbf{S_1}$ and $\mathbf{S_2}$ with $\mathbf{S_1}$ being parent of $\mathbf{S_2}$, we define the *edge cost* $c(\mathbf{S_1}, \mathbf{S_2})$ to be $BestMDL(X, \mathbf{S1})$, where X is the only variable in $\mathbf{S_2} \backslash \mathbf{S_1}$. The goal is then to find a *shortest path* from the start state to the goal state that has the lowest cost. By definition, the shortest path corresponds to an optimal Bayesian network.

Once we have the formulation, we can apply any graph search technique to solve the shortest-path finding problem. In this paper, we introduce an A* search algorithm. The algorithm uses a priority queue, called OPEN list, to organize the search frontier and initialize it with the start state. At each search step, the search node with the smallest cost from the OPEN list is selected to expand its successor nodes. For each successor node, we compute its estimated total cost (f cost) as the sum of the exact cost so far (g cost) and the estimated future cost to the goal state (h cost). Once a node is expanded, it is placed in a CLOSED list. Duplicate detection is performed for each newly generated node on both OPEN and CLOSED lists. If a duplicate is detected in the CLOSED list, we discard the new node immediately due to a consistent heuristic we used, as we will discuss later. If a duplicate is detected in the OPEN list, and the new node has a lower g-cost, we update the existing node with the new g-cost and parent pointer. Once the goal state is selected for expansion, we find a complete path from the start state to the goal state, from which we can extract a Bayesian network as each edge on the path records an optimal parent set for a variable.

The g cost of a node is computed as the sum of edge costs on the best path from the start state to the current node. Each edge cost is computed when a successor is generated during the search by retrieving information from a corresponding node in a parent graph. Since the A* search explores just part of the order graph, we only need to compute some of the edge costs. This pruning is inherent in the search algorithm and is not reliant on any property of the scoring function.

The h cost is computed using a heuristic function. If the heuristic function is *consistent* (a consistent heuristic is guaranteed to be admissible), the A* search algorithm guarantees to find a *shortest* path to any node once that node is selected for expansion. A shortest path that corresponds to an optimal Bayesian network is found once the goal state is selected for expansion. Let $\mathbf{U}$ be a node in the order graph. We consider the following heuristic function h.

Definition 1

$$h(\mathbf{U}) = \sum_{X \in \mathbf{V} \backslash \mathbf{U}} BestMDL(X, \mathbf{V} \backslash \{X\}). \tag{3}$$

Heuristic function h is clearly admissible, because it allows each remaining variable to choose parents from all the other variables in $\mathbf{V}$, which effectively relaxes the acyclic constraint and results in a lower bound cost. The following theorem proves that the heuristic is also consistent.

Theorem 1 *h is consistent.*

Proof: For any successor node $\mathbf{R}$ of $\mathbf{U}$, let $Y \in \mathbf{R} \setminus \mathbf{U}$. We have

$$\begin{aligned} h(\mathbf{U}) &= \sum_{X \in \mathbf{V} \backslash \mathbf{U}} BestMDL(X, \mathbf{V} \backslash \{X\}) \\ &\leq \sum_{X \in \mathbf{V} \backslash \mathbf{U}, X \neq Y} BestMDL(X, \mathbf{V} \backslash \{X\}) \\ &\qquad + BestMDL(Y, \mathbf{U}) \\ &= h(\mathbf{R}) + c(\mathbf{U}, \mathbf{R}). \end{aligned}$$

The inequality holds because fewer variables are used to select optimal parents for Y. Hence, h is consistent. □

The consistent heuristic allows us to discard any duplicate found in the CLOSED list. The heuristic may seem expensive to compute as it requires computing $BestMDL(X, \mathbf{V} \backslash \{X\})$ for each variable X. As we discuss in the next section, we use hash tables to organize the parent graphs. It only takes linear time to find the best score in each parent graph and sum them together. Any subsequent computation of h, however, only takes constant time by subtracting the best score of a newly added variable from the heuristic of a parent node.

4.2 Optimizing parent graphs

The parent graphs are created for looking up the edge costs for the order graph during the search. A typical look-up takes a variable and a candidate parent set as inputs and returns the best parent set out of the candidate set plus its score. The size of a full parent graph is exponential in the number of variables of a domain. It is desirable if we can not only make querying the parent graphs efficient but also minimize the size of the graphs by pruning.

We use *hash tables* to organize the parent graphs to make them efficient to access. A hash function that maps a candidate parent set to an integer index is designed to locate a hash-table entry in constant time. Each entry in a hash table corresponds to a node in a parent graph and points to a data structure that contains information of an optimal parent set. The hash tables are created by a top-down search of the parent graphs, during which we do two things: propagate information of optimal parent sets, and prune parent nodes that are not optimal parent set out of any candidate set of variables.

To propagate optimal parent sets, we use the following theorem presented in [de Campos *et al.*, 2009].

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.

Therefore, when we generate a successor node $\mathbf{V}$ of $\mathbf{U}$, we check whether $MDL(X|\mathbf{V})$ is greater than $BestMDL(X, \mathbf{U})$. If so, we let $\mathbf{V}$ point to a data structure that records $\mathbf{V}$ as the optimal parent set. Otherwise, we propagate the optimal parent set in $\mathbf{U}$ to $\mathbf{V}$. Better yet, we only need to create one data structure containing information about the optimal parent set; both $\mathbf{U}$ and $\mathbf{V}$ point to the same data structure to save space. The propagation makes sure that an optimal parent set can be immediately returned once a hash-table entry is located for a candidate parent set.

Furthermore, we can avoid computing some parent scores using the following theorem presented in [Tian, 2000].

Theorem 3 *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 data points.*

Therefore, there is no need to compute scores for any parent set with a size larger than $\log(\frac{2N}{\log N})$, because these parent sets are guaranteed to be suboptimal. We still need to create hash-table entries for these parent sets for efficient look-up, although it is only necessary to propagate optimal parent sets to them from their ancestor nodes.

Another technique prunes all the supersets of a parent node if all these supersets are guaranteed to be worse than the parent node based on the following theorem presented in [Singh and Moore, 2005].

Theorem 4 *Let* $\mathbf{U} \subset \mathbf{V}$ *and* $X \notin \mathbf{U}$. *Let* $hMDL(X, \mathbf{U}, \mathbf{V})$ *be a lower bound that bounds* $BestMDL(X, \mathbf{R})$ *for any* $\mathbf{R}$ *such that* $\mathbf{U} \subset \mathbf{R} \subseteq \mathbf{V}$. *If* $hMDL(X, \mathbf{U}, \mathbf{V}) < BestMDL(X, \mathbf{U})$, *no proper superset of* $\mathbf{U}$ *can be the optimal parent set for* X.

This theorem presents a nice pruning technique because we do not even need to compute the exact scores of these supersets. However, we need to have the lower bound $hMDL(X, \mathbf{U}, \mathbf{V})$ in order to use the theorem. One such lower bound is defined in [Suzuki, 1996].

Theorem 5 *Let* $\mathbf{U} \subset \mathbf{V}$ *and* $X \notin \mathbf{U}$. *For any* $\mathbf{R}$ *such that* $\mathbf{U} \subset \mathbf{R} \subseteq \mathbf{V}$, *we have*

$$MDL(X|\mathbf{R}) \geq \frac{\log N}{2} K(X|\mathbf{U}). \quad (4)$$

Another one is defined in [Tian, 2000].

Theorem 6 *Let* $\mathbf{U} \subset \mathbf{V}$ *and* $X \notin \mathbf{U}$. *For any* $\mathbf{R}$ *such that* $\mathbf{U} \subset \mathbf{R} \subseteq \mathbf{V}$, *we have*

$$MDL(X|\mathbf{R}) \geq H(X|\mathbf{V}) + \frac{\log N}{2} K(X|\mathbf{U}). \quad (5)$$

Theorem 6 defines a tighter lower bound than Theorem 5, but $H(X|\mathbf{V})$ requires that we collect count statistics for full instantiations of variables $\mathbf{V}$. As we discuss next, Theorem 3 is also used to prune the counts for large variable configurations. We therefore only apply the lower bound in Theorem 5 to the pruning method presented in Theorem 4.

We use the AD-tree method [Moore and Lee, 1998] to collect all the counts from a dataset. An *AD-tree* is an unbalanced tree structure that contains two types of nodes, AD-tree nodes and varying nodes. An AD-tree node stores the number of data points consistent with the variable instantiation of this node; a varying node is used to instantiate the state of a variable. A *full* AD-tree stores counts of data points that are consistent with all *partial instantiations* of the variables.

For n variables with d states each, the number of ADtree nodes in an AD-tree is $(d+1)^n$. It grows even faster than the size of an order or parent graph. It is impractical to compute and store all the count statistics for a large dataset. Therefore, we use a depth-first search to traverse the AD-tree and use the counts to update the parent graphs. The counts can be discarded once they are used. This not only saves space but also improves efficiency. Furthermore, Theorem 3 requires that we only compute scores for small parent sets. Therefore, we only need to collect count statistics for small variable instantiations as well.

The above pruning techniques make it only necessary for the A* search algorithm to collect some of the count statistics and compute parts of the parent and order graphs. They make the search algorithm much more efficient in both computation and space. These pruning techniques are not applicable to the dynamic programming algorithm by Silander and Myllymaki because they compute the counts and parent scores by starting with complete variable instantiations.

4.3 Advantages of A* search

The major advantage of our A* search algorithm over dynamic programming is that A* only explores part of the order graph and computes some of the parent scores; dynamic programming has to fully evaluate these graphs. The sizes of the order and parent graphs are all exponential in the number of variables. The pruning by the A* search algorithm has the potential to significantly improve the scalability of learning optimal Bayesian networks.

However, each step of the A* search algorithm has some overhead cost for computing the heuristic function, maintaining a priority queue, etc. So an A* step is slightly more expensive than a similar dynamic programming step. If the pruning of A* does not outweigh its overhead, A* can be slower than dynamic programming. Since the number of data records is typically small relative to the number of variables in large real datasets, the gain brought by the pruning is likely to outweigh the overhead.

A major difference between A* and the exhaustive search method [de Campos *et al.*, 2009] is that our algorithm always maintains an directed *acyclic* graph during the search. There is no need to detect and break cycles. This difference turns out to be a huge advantage of our algorithm.

5 Experiments

We evaluated our A* search algorithm on a set of benchmark datasets from the UCI repository listed in Table 1 [Asuncion and Newman, 2007]. The datasets have up to 24 variables and $30,162$ data points. We discretized all continuous variables into two states using the mean values, and all discrete variables with five states or more into two states as well (The discrete variables with four states or fewer were left intact). We deleted all the data points with missing values.

Our A* search algorithm is implemented in Java. We compared our algorithm against the exhaustive search (ES) [de Campos *et al.*, 2009] and dynamic programming (DP) algorithms [Silander and Myllymaki, 2006]. The binary code of the ES algorithm was provided by de Campos *et al.* on their website[1]. The C source code of the DP algorithm was provided by Silander and Myllymaki on their website as well[2];

[1]http://www.ecse.rpi.edu/∼cvrl/structlearning.html

[2]http://b-course.hiit.fi/bene

Dataset			Timing results (s)			Space results			
dataset	n	N	ES	DP	A*	A-parent	f-parent	A-order	f-order
wine	14	178	95	1	1	2,427	114,674	5,662	16,384
adult	14	30,162	-	4	11	24,515	114,674	15,103	16,384
zoo	17	101	4,584	4	1	3,831	1.11E+06	28,405	131,072
house	17	435	5,792	16	2	1,418	1.11E+06	30,741	131,072
letter	17	20,000	-	66	112	396,199	1.11E+06	121,673	131,072
statlog	19	752	-	73	12	19,997	4.98E+06	325,403	524,288
hepatitis	20	126	276	63	11	2,105	1.05E+07	321,369	1.05E+06
segment	20	2310	-	70	41	70,304	1.05E+07	866,938	1.05E+06
meta	22	528	-	227	65	182,184	4.61E+07	713,783	4.19E+06
imports	22	205	-	315	100	16,283	4.61E+07	2.76E+06	4.19E+06
hosre	23	300	-	1,043	228	9,736	9.65E+07	5.21E+06	8.39E+06
heart	23	267	-	1,024	262	19,614	9.65E+07	7.20E+06	8.39E+06
mushroom	23	8,124	-	846	503	85,597	9.65E+07	7.33E+06	8.39E+06
parkinsons	24	195	-	714	239	25,743	9.65E+07	5.75E+06	8.39E+06

Table 1: A comparison on the running time (in seconds) for the following algorithms: exhaustive search (ES), Dynamic programming (DP), and the A* search algorithm, and the sizes of AD-tree, parent, and order graphs. The column headings have the following meanings: ‘n’ is the total number of variables; ‘N’ is the number of data points; ‘A-parent’ and ‘A-order’ are the sizes of the parent and order graphs by A* respectively; ‘f-parent’ and ‘f-order’ are the same statistics when no pruning is done. ‘-’ shows failure to find optimal solutions due to out of memory or out of time (time limit is 30 minutes).

their DP algorithm was shown to be much more efficient than the implementation in [Singh and Moore, 2005]. ES and DP do not calculate MDL, but they use the BIC score, which uses an equivalent calculation as MDL. Our results confirmed that the algorithms found Bayesian networks that either are the same or belong to the same equivalence class. The experiments were performed on a 2.66 GHz Intel Xeon with 16GB of RAM and running SUSE Linux Enterprise Server version 10.

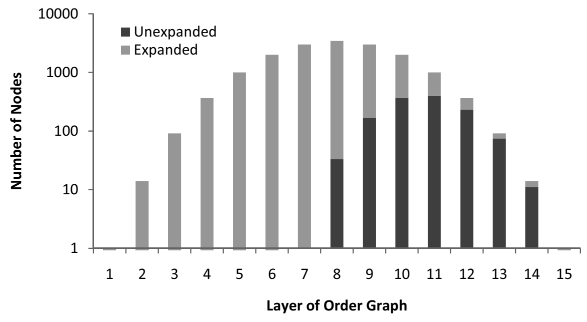
Table 1 reports the running time of the three algorithms in solving the benchmark datasets. We terminate an algorithm early if it runs for more than 30 minutes on a dataset. We also report the sizes of parent graphs and order graphs created with or without the pruning by A* to compare the amount of memory consumption by DP and A*. The memory needed by DP is equivalent to the complete parent graphs and order graphs. We did not track the size of the search space by ES because only binary code is provided.

The timing results show that our A* algorithm is typically several times faster than DP and orders of magnitude faster than ES on most of the datasets we tested. A* is only slower than DP on adult and letter. The reason is both these datasets have a large number of data points, which makes the pruning technique in Theorem 3 less effective. Although the DP algorithm does not perform any pruning, due to its simplicity, the algorithm can be highly streamlined and optimized in performing all its calculations. That is why the DP algorithm was faster than A* search on adult and letter. However, our A* algorithm was much more efficient when the number of data points is relatively small in comparison to the number of variables, which is often the case for real-world machine learning datasets.

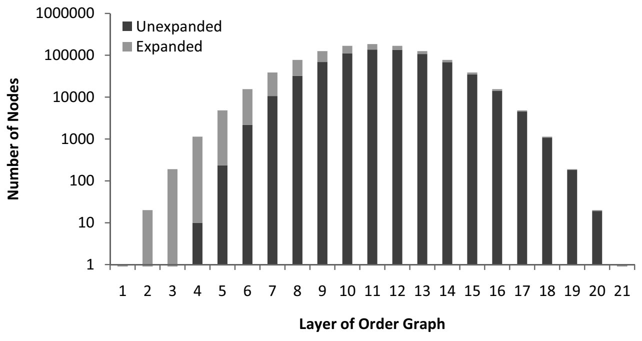
The sizes of full and pruned parent and order graphs show that the size of the parent graphs is always significantly reduced by the pruning of A* search. Many parent nodes are pruned because they are not optimal parent sets of any candidate set, which also means many candidate sets share the same optimal parent sets. The amount of pruning done in the order graphs, however, varies a lot across the datasets. For example on adult, the size reduction is about 80% for the parent graphs, but only 10% for the order graph. On hepatitis, however, the size of the order graph is reduced 70% after pruning. The total size of the parent graphs also reduced from being in the order of 10^7 to merely $2,105$. A* kept all its parent and order graphs in RAM. It would not have been possible to store the full order and parent graphs of some of the largest datasets in RAM though. The DP algorithm was able to solve all these datasets because it stores intermediate results as computer files on hard disks.

To gain more insight on the pruning of order graph, we plot in Figure 3 the detailed number of nodes expanded by A* versus the full size at each layer for adult and hepatitis. In the beginning layers of the search, the heuristic function used by A* provides lower bounds that are still loose for the actual scores, so A* may have to expand most nodes in the beginning layers. As the depth increases, the heuristic function becomes much tighter, which enables A* to prune many more orders. For the adult dataset, A* expanded almost all the order nodes in the beginning 7 layers of the order graph before it started to prune order nodes in the final layers. In contrast, only a small percentage of the nodes was expanded in the order graph of hepatitis. The pruning became quite effective as early as at layer 4 and 5. Only a few nodes were expanded in the last 10 layers.

Finally, the exhaustive search algorithm [de Campos *et al.*, 2009] is much slower than the A* search algorithm. The major difference between these two search algorithms is the formulation of the search space. The results indicate that it is better to search in the space of directed acyclic graphs directly in finding an optimal Bayesian network.



(a) *Adult*



(b) *Hepatitis*

Figure 3: The number of nodes expanded by A* at each layer of the order graph versus that in the full order graph.

6 Concluding Remarks

We have presented an A* search algorithm for learning optimal Bayesian networks. It uses a consistent heuristic to guide the search so that only the most promising parts of the solution space are explored. The pruning by A* allows an optimal solution to be found not only much more efficiently, but also in less space. The A* search algorithm is especially effective in domains where there is a large number of variables relative to the number of data instances. Such datasets are rather common in real-world machine learning problems.

We currently use MDL in the A* search algorithm. However, any decomposable scoring function can be used in the A* search algorithm, as the same heuristic function defined in Equation 3 can be used. However, the amount of pruning may potentially be affected. For example, Theorem 3 is a property of the MDL score. Theorem 4 requires a lower/upper bound for the scoring function. We may only be able to use some of these pruning techniques depending on the scoring function being used.

Finally, exact algorithms for learning optimal Bayesian networks are still limited to relatively small problems. This makes approximation methods still useful in domains with a large number of variables. Our algorithm, however, can serve as the basis to evaluate different approximation methods on larger datasets than existing exact methods.

Acknowledgement: This research was supported by the NSF grants IIS-0953723 and EPS-0903787.

References

[Asuncion and Newman, 2007] A. Asuncion and D.J. Newman. UCI machine learning repository. 2007.

[Chickering, 2002] David Maxwell Chickering. Learning equivalence classes of Bayesian-network structures. *Journal of Machine Learning Research*, 2:445–498, February 2002.

[Cooper and Herskovits, 1992] G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[de Campos *et al.*, 2009] Cassio de Campos, Zhi Zeng, and Qiang Ji. Structure learning of Bayesian networks using constraints. In *Proceedings of the International Conference on Machine Learning*, Montreal, Canada, 2009.

[Friedman *et al.*, 1999] N. Friedman, I. Nachman, and D. Peer. Learning Bayesian network structure from massive datasets: The sparse candidate algorithm. In *Proceedings of UAI-13*, pages 206–215, 1999.

[Heckerman, 1998] David Heckerman. A tutorial on learning with Bayesian networks. In Dawn Holmes and Lakhmi Jain, editors, *Innovations in Bayesian Networks*, volume 156 of *Studies in Computational Intelligence*, pages 33–82. Springer Berlin / Heidelberg, 1998.

[Koivisto and Sood, 2004] M. Koivisto and K. Sood. Exact Bayesian structure discovery in Bayesian networks. *J. Mach. Learn. Res.*, 5:549–573, 2004.

[Moore and Lee, 1998] Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, March 1998.

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

[Silander and Myllymaki, 2006] T. Silander and P. Myllymaki. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of UAI-06*, 2006.

[Singh and Moore, 2005] Ajit Singh and Andrew W. Moore. Finding optimal Bayesian networks by dynamic programming. Technical Report CMU-CALD-05-106, Carnegie Mellon University, 2005.

[Suzuki, 1996] Joe Suzuki. Learning bayesian belief networks based on the minimum description length principle: An efficient algorithm using the B&B technique. In *International Conference on Machine Learning*, pages 462–470, 1996.

[Tian, 2000] Jin Tian. A branch-and-bound algorithm for MDL learning Bayesian networks. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 580–588, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.