Tightening Bounds for Bayesian Network Structure Learning

Xiannian Fan, Changhe Yuan
Graduate Center and Queens College
City University of New York
365 Fifth Avenue, New York 10016
{xfan2@gc, change.yuan@qc}.cuny.edu

Brandon Malone
Helsinki Institute for Information Technology
Department of Computer Science
Fin-00014 University of Helsinki, Finland
brandon.malone@cs.helsinki.fi

Abstract

A recent breadth-first branch and bound algorithm (BF-BnB) for learning Bayesian network structures (Malone et al. 2011) uses two bounds to prune the search space for better efficiency; one is a lower bound calculated from pattern database heuristics, and the other is an upper bound obtained by a hill climbing search. Whenever the lower bound of a search path exceeds the upper bound, the path is guaranteed to lead to suboptimal solutions and is discarded immediately. This paper introduces methods for tightening the bounds. The lower bound is tightened by using more informed variable groupings when creating the pattern databases, and the upper bound is tightened using an anytime learning algorithm. Empirical results show that these bounds improve the efficiency of Bayesian network learning by two to three orders of magnitude.

Introduction

This paper considers the problem of learning an optimal Bayesian network structure for given data and scoring function. Exact algorithms have been developed for solving this problem based on dynamic programming (Koivisto and Sood 2004; Ott, Imoto, and Miyano 2004; Silander and Myllymaki 2006; Singh and Moore 2005; Malone, Yuan, and Hansen 2011), integer linear programming (Cussens 2011; Jaakkola et al. 2010) and heuristic search (Yuan, Malone, and Wu 2011; Malone et al. 2011; Malone and Yuan 2013). This paper focuses on the heuristic search approach, which formulates BN learning as a shortest path problem (Yuan, Malone, and Wu 2011; Yuan and Malone 2013) and applies various search methods to solve it, e.g., breadth-first branch and bound (BFBnB) (Malone et al. 2011). BFBnB uses two bounds, a lower bound and an upper bound, to prune the search space and scale up Bayesian network learning. Whenever the lower bound of a search path exceeds the upper bound, the path is guaranteed to lead to suboptimal solutions and is discarded immediately. With the help of disk space for storing search information, BFBnB was able to scale to larger data sets than many existing exact algorithms.

In this paper, we aim to tighten the lower and upper bounds of BFBnB. The lower bound is calculated from a pattern database heuristic called *k-cycle conflict heuristic* (Yuan and Malone 2012). Its main idea is to relax the acyclicity constraint between groups of variables; acyclicity is enforced among the variables within each group. A naive grouping based on the ordering of variables in a data set was used previously. Intuitively, a more informed grouping that minimizes correlation between the variables in different groups and maximizes the correlation within each group should provide a tighter lower bound. We investigate various approaches for achieving that, including constraint-based methods and topological ordering-based methods.

Any Bayesian network structure can serve as an upper bound because it is guaranteed to have an equal or worse score than the optimal structure (we consider the minimization problem in this paper, i.e., the lower the score, the better). The upper bound originally used in BFBnB was a suboptimal solution found by a *beam search*-based hill climbing search (Malone, Yuan, and Hansen 2011). Although efficient, the hill climbing method method may provide poor upper bounds. We also investigate approaches for finding better upper bounds.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

This section provides an overview of Bayesian network structure learning, the shortest-path formulation and the BF-BnB algorithm.

Bayesian Network Structure Learning

We consider the problem of learning a Bayesian network structure from a dataset $\mathbf{D} = \{D_1, ..., D_N\}$, where D_i is an instantiation of a set of random variables $\mathbf{V} = \{X_1, ..., X_n\}$. A scoring function is given to measure the goodness of fit of a network structure to $\mathbf{D}$. The problem is to find the structure which optimizes the score. We only require that the scoring function is *decomposable*. Many commonly used scoring functions, including MDL , BIC , AIC , BDe and fNML, are decomposable. For the rest of the paper, we assume the local scores, $score(X_i|PA_i)$, where PA_i is a parent set of X_i, are computed prior to the search.

Shortest Path Formulation

The above structure learning problem was recently formulated as a shortest-path problem (Yuan, Malone, and Wu

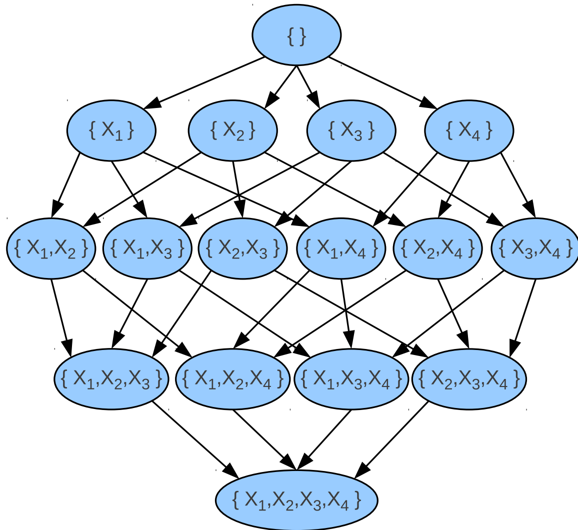


Figure 1: The order graph for four variables

2011; Yuan and Malone 2013). Figure 1 shows the *implicit* search graph for four variables. The top-most node with the empty variable set is the *start* node, and the bottom-most node with the complete set is the *goal* node.

An arc from $\mathbf{U}$ to $\mathbf{U} \cup \{X\}$ in the graph represents generating a successor node by adding a new variable X to an existing set of variables $\mathbf{U}$; the cost of the arc is equal to the score of the optimal parent set for X out of $\mathbf{U}$, and is computed by considering all subsets of $\mathbf{U}$, i.e.,

$$BestScore(X, \mathbf{U}) = \min_{PA_X \subseteq \mathbf{U}} score(X|PA_X).$$

In this search graph, each path from the start to the goal corresponds to an ordering of the variables in the order of their appearance, so the search graph is also called as *order graph*. Each variable selects optimal parents from the variables that precede it, so combining the optimal parent sets yields an optimal structure for that ordering. The shortest path gives the global optimal structure.

Breadth-First Branch and Bound

Malone *et al.* (2011) proposed to use breadth-first search to solve the shortest path problem. They observed that the order graph has a regular layered structure, and the successors of any node only appear in the very next layer. By searching the graph layer by layer, the algorithm only needs the layer being generated in RAM. All nodes from earlier layers are stored on disk, and nodes from the layer being expanded are read in from disk one by one. Furthermore, nodes from the layer being generated are written to disk as RAM fills; delayed duplicate detection methods are used to remove unnecessary copies of nodes (Korf 2008).

The basic breadth-first search can be enhanced with branch and bound, resulting in a breadth-first branch and bound algorithm (BFBnB). An upper bound ub is found in the beginning of the search using a hill climbing search. A lower bound called f-cost is calculated for each node $\mathbf{U}$ by summing two costs, g-cost and h-cost, where g-cost stands for the shortest distance from the start node to $\mathbf{U}$, and h-cost stands for an optimistic estimation on how far away $\mathbf{U}$ is from the goal node. The h-cost is calculated from a heuristic function. Whenever $f > ub$, all the paths extending $\mathbf{U}$ are guaranteed to lead to suboptimal solutions and are discarded immediately. The algorithm terminates when reaching the goal node. Clearly the tightness of the lower and upper bounds has a high impact on the performance of the algorithm. We study various approaches to tightening the bounds.

Tightening Lower Bounds

This section focuses on tightening the lower bounds of BFBnB. We first provide a brief review of the k-cycle conflict heuristic, and then discuss how it can be tightened.

K-Cycle Conflict Heuristic

The following simple heuristic function was introduced in (Yuan, Malone, and Wu 2011) for computing lower bounds for A* search.

Definition 1 *Let* $\mathbf{U}$ *be a node in the order graph. Its heuristic value is*

$$h(\mathbf{U}) = \sum_{X \in \mathbf{V} \setminus \mathbf{U}} BestScore(X, \mathbf{V} \setminus \{X\}). \quad (1)$$

The above heuristic function allows each remaining variable to choose optimal parents from all the other variables. Therefore it completely relaxes the acyclicity constraint of Bayesian networks to allow cycles between variables in the estimation. The heuristic was proven to be admissible, meaning it never overestimates the distance to the goal (Yuan, Malone, and Wu 2011). Admissible heuristics guarantee the optimality of BFBnB. However, because of the complete relaxation of the acyclicity constraint, the simple heuristic may generate loose lower bounds.

In (Yuan and Malone 2012), an improved heuristic function called *k-cycle conflict heuristic* was proposed by reducing the amount of relaxation. The idea is to divide the variables into multiple groups with a size up to k and enforce acyclicity within each group while still allowing cycles between the groups. There are two major approaches to dividing the groups. One is to enumerate all of the subsets with a size up to k (each subset is called a pattern); a set of mutually exclusive patterns covering $\mathbf{V} \setminus \mathbf{U}$ can be selected to produce a lower bound for node $\mathbf{U}$ as the heuristic is *additive* (Felner, Korf, and Hanan 2004). This approach is called *dynamic pattern database*. In (Yuan and Malone 2012), the dynamic pattern database was created by performing a reverse *breadth-first search* for the last k layers in the order graph. The search began with the goal node $\mathbf{V}$ whose reverse g cost is 0. A reverse arc from $\mathbf{U}' \cup \{X\}$ to $\mathbf{U}'$ corresponds to selecting the best parents for X from among $\mathbf{U}'$ and has a cost of $BestScore(X, \mathbf{U}')$. The optimal reverse g cost for node $\mathbf{U}'$ gives the cost of the pattern $\mathbf{V} \setminus \mathbf{U}'$. Furthermore, each pattern has an associated *differential score*, which is its improvement over the simple heuristic given in Equation 1.

Patterns which have the same differential score as their subsets cannot improve the dynamic pattern database and are pruned.

The other approach is to divide the variables $\mathbf{V}$ into several static groups $\mathbf{V_i}$ (typically two). We then enumerate all subsets of each group $\mathbf{V_i}$ as the patterns, which can be done by a reverse breadth-first search similar to dynamic pattern databases in an order graph containing only $\mathbf{V_i}$ (Yuan and Malone 2012). The patterns from different groups are guaranteed to be mutually exclusive, so we simply pick out the maximum-size pattern for each group that is a subset of $\mathbf{V} \setminus \mathbf{U}$ and add them together as the lower bound. This approach is called *static pattern database*. Both dynamic and static pattern databases remain admissible.

Tightening Dynamic Pattern Database

To use the dynamic pattern database to calculate the tightest-possible lower bound for a search node $\mathbf{U}$, we select the set of mutually exclusive patterns which covers all of the variables in $\mathbf{V} \setminus \mathbf{U}$ and has the maximum sum of costs. This can be shown to be equivalent to the *maximum weighted matching* problem (Felner, Korf, and Hanan 2004), which is NP-hard (Papadimitriou and Steiglitz 1982) for $k > 2$. Consequently, the previous work (Yuan and Malone 2012) used a *greedy approximation* to solve the matching problem; its idea is to greedily choose patterns with the maximum differential scores.

Solving the matching problem exactly improves the tightness of dynamic pattern database heuristic. To achieve that, we formulate the problem as an integer linear program (Bertsimas and Tsitsiklis 1997) and solve it exactly. A set of binary variables $A_{i,j}$ are created which indicate if X_i is in pattern j. A second set of variables p_j are created which indicate if pattern j is selected. The linear constraints that $A \cdot p = e$, where e is a vector with 1s for each variable in $\mathbf{V} \setminus \mathbf{U}$, are also added to the program. A standard integer linear program solver, such as SCIP (Achterberg 2007), is used to maximize the cost of the selected patterns subject to the constraints. The resulting value is the lower bound which is guaranteed to be at least as tight as that found by the greedy algorithm. Solving the integer linear program introduces overhead compared to the simple greedy algorithm, though.

Tightening Static Pattern Database

The tightness of the static pattern database heuristic depends highly on the static grouping used during its construction. A very simple grouping (SG) method was used in (Yuan and Malone 2012). Let $X_1, ..., X_n$ be the ordering of the variables in a dataset. SG divides the variables into two balanced groups, $\{X_1, ..., X_{\lceil \frac{n}{2} \rceil}\}$ and $\{X_{\lceil \frac{n}{2} \rceil+1}, ..., X_n\}$. Even though SG exhibited excellent performance on some datasets in (Yuan and Malone 2012), there is potential to develop more informed grouping strategies by taking into account of the correlation between the variables.

A good grouping method should reduce the number of directed cycles between the variables and enforce acyclicity as much as possible. Since no cycles are allowed within each group, we should *maximize* the correlation between the variables in each group. Also, because cycles are allowed between groups, we should *minimize* the correlation between the groups. Consider two variables X_1 and X_2. If they have no correlation, there will be no arc between these two variables in the optimal Bayesian network, so there is no need to put the two variables in the same group. On the other hand, if they have strong correlation, and if they are put into different groups G_1 and G_2, X_1 is likely to select X_2 from G_2 as a parent, and vice versa. Then a cycle will be introduced between the two groups, resulting in a loose bound. It is better to put these variables in one group and disallow cycles between them.

The above problem can be naturally formulated as a *graph partition* problem. Given a weighted undirected graph $G = (V, E)$ with V vertices and E edges, which we call the *partition graph*, the graph partition problem cuts the graph into 2 or more components while minimizing the weight of edges between different components. We are interested in *uniform* or *balanced* partitions as it has been shown that such partitions typically work better in static pattern databases (Felner, Korf, and Hanan 2004). Two issues remain to be addressed in the formulation: creating the partition graph and performing the partition.

Creating the Partition Graph We propose to use two methods to create the partition graph. One is to use constraint-based learning methods such as the Max-Min Parent Children (MMPC) algorithm (Tsamardinos, Brown, and Aliferis 2006) to create the graph. The MMPC algorithm uses independence tests to find a set called *candidate parent and children* (CPC) for each variable X_i. The CPC set contains all parent and child candidates of X_i. The CPC sets for all variables together create an undirected graph. Then MMPC assigns a weight to each edge of the undirected graph by independent tests, which indicate the strength of correlation with p-values. Small p-values indicate high correlation, so we use the *negative* p-values as the weights. We name this approach *family grouping* (FG).

The second method works as follows. The simple heuristic in Eqn. 1 considers the optimal parent set out of all of the other variables for each variable X, denoted as $PA(X)$. Those $PA(X)$ sets together create a directed *cyclic* graph; by simply ignoring the directions of the arcs, we again obtain an undirected graph. We then use the same independence tests as in MMPC to obtain the edge weights. We name this approach *parents grouping* (PG).

Performing the Partition Many existing partition algorithms can be used to perform the graph partition. Since we prefer balanced partitions, we select the METIS algorithm (Karypis and Kumar 1998). METIS is a multilevel graph partitioning method: it first coarsens the graph by collapsing adjacent vertices and edges, then partitions the reduced graph, and finally refines the partitions back into the original graph. Studies have shown that the METIS algorithm has very good performance at creating balanced partitions across many domains. Also, since the partition is done on the reduced graph, the algorithm is quite efficient. This is important for our purpose of using it only as a preprocessing

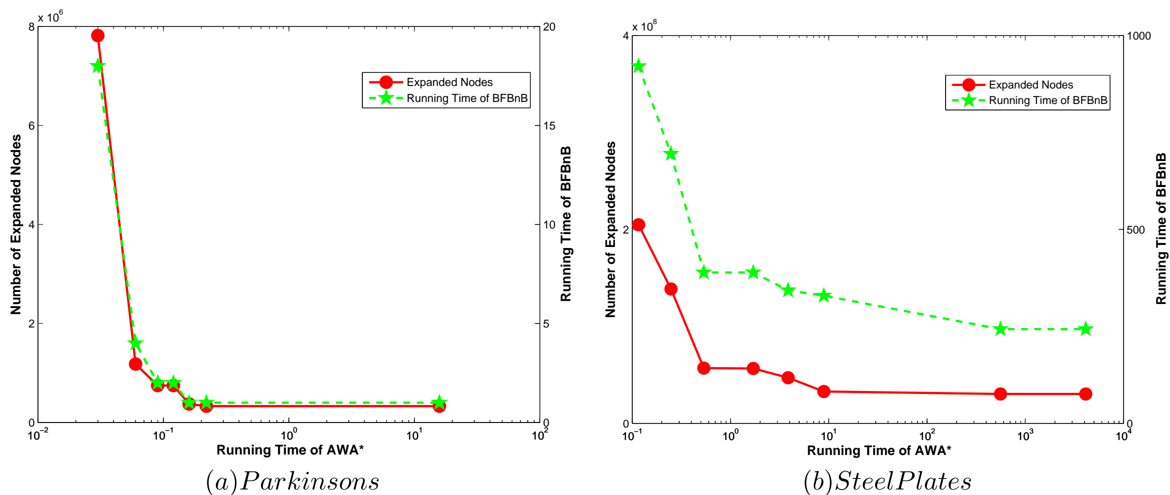


Figure 2: The effect of upper bounds generated by running AWA* for different amount of time (in seconds) on the performance of BFBnB on Parkinsons and Steel Plates.

step.

Topology Grouping Besides the above grouping methods based on partitioning undirected graphs, we also propose another method based on topologies of directed acyclic graphs found by approximate Bayesian network learning algorithms. Even though these algorithms cannot guarantee to find the optimal solutions, some of them can find Bayesian networks that are close to optimal. We assume that the suboptimal networks capture many of the dependence/independence relations of the optimal solution. So we simply divide the topological ordering of the suboptimal network into two groups, and those are the grouping for the static pattern database. Many approximate learning algorithms can be used to learn the suboptimal Bayesian networks. We select the recent anytime window A* algorithm (Aine, Chakrabarti, and Kumar 2007; Malone and Yuan 2013) (introduced for obtaining upper bounds in the next section) for this purpose. We name this approach as *topology grouping* (TG).

Tightening Upper Bounds

Breadth-first branch and bound improves search efficiency by pruning nodes which have an f-cost worse than some known upper bound, ub. In the best case, when ub is equal to the optimal cost, BFBnB provably performs the minimal amount of work required to find the shortest path (Zhou and Hansen 2006). As the quality of ub decreases, though, it may perform exponentially more work. Consequently, a tight upper bound is pivotal for good search behavior.

A beam search-based hill climbing algorithm was used in (Yuan and Malone 2012) to find an upper bound. Recently, anytime window A* (AWA*) (Aine, Chakrabarti, and Kumar 2007; Malone and Yuan 2013) was shown to find high quality, often optimal, solutions very quickly. Briefly, AWA* uses a sliding window search strategy to explore the order graph over a number of iterations. During each iteration, the algorithm uses a fixed window size, w, and tracks the layer l of the deepest node expanded. Nodes are expanded in best-first order as usual by A*; however, nodes selected for expansion in a layer less that $l - w$ are instead *frozen*. A path to the goal is found in each iteration, which gives an upper bound solution. After finding the path to the goal, the window size is increased by 1 and the frozen nodes become the open list. The iterative process continues until no nodes are frozen during an iteration, which means the upper bound solution is optimal, or a resource bound, such as running time, is exceeded. Due to its ability to often find tight upper bounds quickly, we use AWA* in this work.

Empirical Results

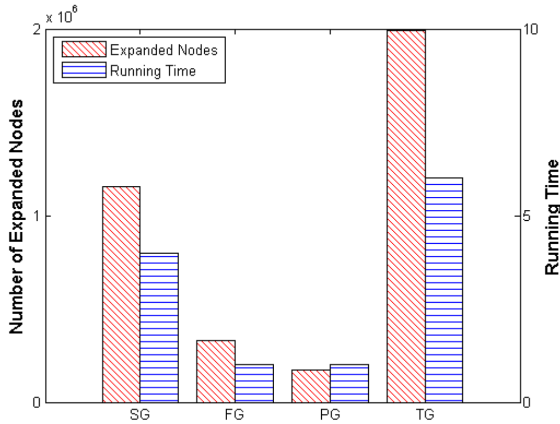
We empirically tested our new proposed tightened bounds using the BFBnB algorithm [1]. We use benchmark datasets from the UCI machine learning repository and Bayesian Network Repository. The experiments were performed on an IBM System x3850 X5 with 16 core 2.67GHz Intel Xeon Processors and 512G RAM; 1.7TB disk space was used.

Results on Upper Bounds

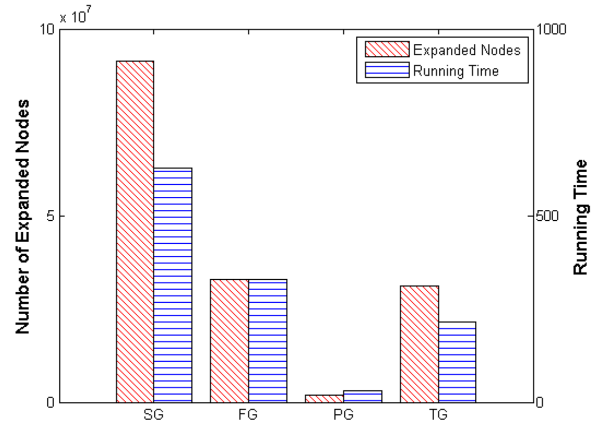
We first tested the effect of the upper bounds generated by AWA* on BFBnB on two datasets: Parkinsons and Steel Plates. Since AWA* is an anytime algorithm, it produces multiple upper bounds during its execution. We recorded each upper bound plus the corresponding running time of AWA*. Each upper bound is tested on BFBnB. In this experiment, we use static pattern database with family grouping (FG) as the lower bound. Figure 2 plots the running time of AWA* versus the running time and number of expanded nodes of BFBnB.

The experiments show that the upper bounds had a huge impact on the performance of BFBnB. Parkinsons is a small

[1]For comparisons between heuristic search-based methods and other exact learning algorithms, please refer to, e.g., (Malone and Yuan 2013; Yuan and Malone 2013).



$(a) Parkinsons$



$(b) SteelPlates$

Figure 4: The effect of different grouping strategies on the number of expanded nodes and running time of BFBnB on Parkinsons and Steel Plates. The four grouping methods are the simple grouping (SG), family grouping (FG), parents grouping (PG), and topology grouping (TG).

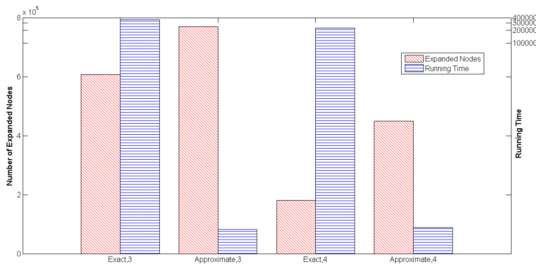


Figure 3: The number of expanded nodes and running time needed by BFBnB to solve Parkinsons based on dynamic pattern database strategies: matching ("Exact" or "Approximate") and k ("3" or "4").

dataset with only 23 variables. AWA* finds upper bounds within several hundredths of seconds and solves the dataset optimally around 10 seconds. The BFBnB was able to solve Parkinsons in 18 seconds using the first upper bound found by AWA* at 0.03s, and in 2 seconds using the upper bound found at 0.06s. Subsequent upper bounds bring additional but marginal improvements to BFBnB. This confirms the results in (Malone and Yuan 2013) that AWA* often finds excellent solutions early on and spends the rest of time finding marginally better solutions, or just proving the optimality of the early solution.

Steel Plates is a slightly larger dataset. AWA* needs more than 4,000 seconds to solve it. We tested all the upper bounds found by AWA*. The first upper bound found at 0.1s enabled BFBnB to solve the dataset within 1,000 seconds, and the third upper bound found at 0.5s solved the dataset within 400 seconds. Again, all the subsequent upper bounds only brought marginal improvements, even for the optimal bound found by AWA*. For much larger datasets, we believe that running AWA* for several hundreds of seconds should already generate sufficiently tight upper bounds; the time is minimal when compared to the amount of time needed to prove the optimality of Bayesian network structures. In all these experiments, the results on the number of expanded nodes show similar patterns as the running time.

Results on Dynamic Pattern Databases

We compared the dynamic pattern database with exact matching against with the previous greedy matching in (Yuan and Malone 2012) on the Parkinsons dataset. We used AWA* upper bounds in these experiments. The exact matching method is guaranteed to produce a tighter bound than the greedy method. However, an exact matching problem needs to be solved at each search step; the total amount of time needed may become too prohibitive. This concern is confirmed in the experiments. Figure 3 compares the performance of BFBnB with four dynamic pattern databases ($k = 3/4$, and matching=exact/approximate). Solving the matching problems exactly did reduce the number of expanded nodes by up to 2.5 times. However, solving the many exact matching problems necessary to calculate the h-costs of the generated nodes in the order graph far outweighed the benefit of marginally fewer expanded nodes. For example, BFBnB with the exact matching required up to 400k seconds to solve Parkinsons, compared to only 4 seconds using the approximate matching.

Results on Static Pattern Databases

We compared the new grouping strategies, including family grouping (FG), parents grouping (PG), and topology grouping (TG), against the simple grouping (SG) on Parkinsons and Steel Plates. We again used AWA* upper bounds. Figure 4 shows the results. On Parkinsons, FG and PG had the best performance; both FG and PG enabled BFBnB to solve the dataset three times faster. TG was even worse than

Dataset			Results						
Name	n	N		loose (SG)	tight (SG)	loose (FG)	tight (FG)	loose (PG)	tight (PG)
Parkinsons	23	195	Time (s)	13	4	12	1	8	1
			Nodes	5.3	1.1	5.0	0.3	4.2	0.2
Autos	26	159	Time (s)	19	5	16	3	20	4
			Nodes	9.6	2.4	7.8	0.9	8.7	1.6
Horse Colic	28	300	Time (s)	148	110	10	1	26	3
			Nodes	44.1	34.6	4.5	0.2	9.6	1.2
Steel Plates	28	1,941	Time (s)	1,541	628	1,535	329	1,329	30
			Nodes	246.5	91.4	246.3	32.8	222.5	2.0
Flag	29	194	Time (s)	150	79	115	50	240	88
			Nodes	39.0	21.4	28.3	14.4	57.6	25.6
WDBC	31	569	Time (s)	11,864	5,621	9,936	770	9,478	1,083
			Nodes	1,870.3	906.3	1,631.0	120.3	1,553.4	192.1
Alarm	37	1,000	Time (s)	OM	142,365	39,578	69	40,813	78
			Nodes	OM	16,989.2	5,154.1	13.0	5,154.1	13.0
Bands	39	277	Time (s)	OM	333,601	OM	47,598	OM	836
			Nodes	OM	31,523.6	OM	4,880.1	OM	128.9
Spectf	45	267	Time (s)	OM	506	OM	500	OM	11
			Nodes	OM	55.1	OM	27.0	OM	1.3

Table 1: The number of expanded nodes (in millions) and running time (in seconds) of the BFBnB algorithm on a set of benchmark datasets with different combinations of upper (loose: hill climbing; tight: AWA*) and lower bounds (SG: simple grouping; FG: family grouping; PG: parents grouping). "n" is the total number of variables, and "N" is the number of data points; "OM" means out of external memory.

the SG. On Steel Plates, however, TG was slightly better than FG. PG was again the best performing method on this dataset; it allowed BFBnB to solve the dataset 10 times faster than SG. These results suggest that family grouping (FG) and parents grouping (PG) are both reliably better than the simple grouping (SG). We therefore use FG and PG as the lower bounds in our subsequent experiments.

Batch Results on Benchmark Datasets

We tested BFBnB with different combinations of upper and lower bounds on a set of benchmark datasets. The upper bounds are found by hill climbing (loose) (Malone and Yuan 2013) and AWA* (tight); the lower bounds are static pattern databases based on simple grouping (SG), family grouping (FG), and parents grouping (PG). Dynamic pattern database with approximate matching was excluded because it is on average worse than static pattern databases even with simple grouping (Yuan and Malone 2012). We let AWA* run up to 10 seconds for the datasets with fewer than 30 variables, and up to 160 seconds for datasets with 30+ variables to generate the upper bounds (the upper bounds could be found much earlier though). The results are shown in Table 1.

The results further highlight the importance of tight upper bounds. BFBnB could not solve the largest datasets with the loose bounds due to running out of external memory. The tight bounds by AWA* enabled BFBnB to solve all of the datasets. The benefit of the new lower bounds is obvious, too. FG and PG consistently outperformed SG given any upper bound; the speedup ranged from several times faster (e.g., Autos and WDBC) to several orders of magnitude faster (e.g., Horse Colic and Alarm). When comparing FG and PG against each other, FG has advantages on some of the smaller datasets (e.g., Horse Colic and Flag) but PG was much faster on the two biggest datasets; it was 50 times faster on these datasets than FG with AWA* upper bounds.

Another important observation is that "cooperative" interactions between the tighter upper and lower bounds enabled BFBnB to achieve much better efficiency. For example, on WDBC, the AWA* upper bounds alone improved the speed of BFBnB by only 2 times; FG and PG only improved the speed slightly. However, AWA* upper bounds and FG together improved the speed by more than 10 times (from 11,864s to 770s), and AWA* upper bounds and PG together achieved similar speedup (to 1,083s). Alarm is another example for which the combination of tighter bounds produced dramatic speedup compared to either in isolation.

Conclusion

This paper investigated various methods for tightening the upper and lower bounds of breadth-first branch and bound algorithm (BFBnB) for Bayesian network structure learning (Yuan and Malone 2012). The results suggest that the upper bounds generated by anytime window A* and lower bounds generated by family grouping- and parents grouping-based static pattern databases can improve the efficiency and scalability of BFBnB by several orders of magnitude.

As future work, we plan to investigate integration of domain knowledge into the pattern database heuristics. Combining prior knowledge with information from suboptimal solutions could scale Bayesian network learning to big data.

Acknowledgments This work was supported by the NSF (grants IIS-0953723 and IIS-1219114) and the Academy of Finland (grant 251170).

References

Achterberg, T. 2007. *Constrained Integer Programming*. Ph.D. Dissertation, TU Berlin.

Aine, S.; Chakrabarti, P. P.; and Kumar, R. 2007. AWA*-a window constrained anytime heuristic search algorithm. In *Proceedings of the 20th International Joint Conference on Artifical intelligence*.

Bertsimas, D., and Tsitsiklis, J. 1997. *Introduction to Linear Optimization*. Athena Scientific.

Cussens, J. 2011. Bayesian network learning with cutting planes. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Jaakkola, T.; Sontag, D.; Globerson, A.; and Meila, M. 2010. Learning Bayesian network structure using LP relaxations. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*.

Karypis, G., and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20(1):359–392.

Koivisto, M., and Sood, K. 2004. Exact Bayesian structure discovery in Bayesian networks. *Journal of Machine Learning Research* 5:549–573.

Korf, R. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 35(6).

Malone, B., and Yuan, C. 2013. Evaluating anytime algorithms for learning optimal Bayesian networks. In *In Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence*.

Malone, B.; Yuan, C.; Hansen, E.; and Bridges, S. 2011. Improving the scalability of optimal Bayesian network learning with external-memory frontier breadth-first branch and bound search. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*.

Malone, B.; Yuan, C.; and Hansen, E. 2011. Memory-efficient dynamic programming for learning optimal Bayesian networks. In *Proceedings of the 25th National Conference on Artifical Intelligence*.

Ott, S.; Imoto, S.; and Miyano, S. 2004. Finding optimal models for small gene networks. In *Pacific Symposium on Biocomputing*.

Papadimitriou, C. H., and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Silander, T., and Myllymaki, P. 2006. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*.

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

Tsamardinos, I.; Brown, L.; and Aliferis, C. 2006. The max-min hill-climbing Bayesian network structure learning algorithm. *Machine Learning* 65:31–78.

Yuan, C., and Malone, B. 2012. An improved admissible heuristic for finding optimal bayesian networks. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*.

Yuan, C., and Malone, B. 2013. Learning optimal Bayesian networks: A shortest path perspective. *Journal of Artificial Intelligence Research* 48:23–65.

Yuan, C.; Malone, B.; and Wu, X. 2011. Learning optimal Bayesian networks using A* search. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*.

Zhou, R., and Hansen, E. A. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170:385–408.