A Depth-first Branch and Bound Algorithm for Learning Optimal Bayesian Networks

Brandon Malone[1] and Changhe Yuan[2]

[1] Helsinki Institute for Information Technology,
Department of Computer Science, Fin-00014 University of Helsinki, Finland
[2] Queens College/City University of New York
brandon.malone@cs.helsinki.fi, changhe.yuan@qc.cuny.edu

Abstract. Early methods for learning a Bayesian network that optimizes a scoring function for a given dataset are mostly approximation algorithms such as greedy hill climbing approaches. These methods are anytime algorithms as they can be stopped anytime to produce the best solution so far. However, they cannot guarantee the quality of their solution, not even mentioning optimality. In recent years, several exact algorithms have been developed for learning optimal Bayesian network structures. Most of these algorithms only find a solution at the end of the search, so they fail to find any solution if stopped early for some reason, e.g., out of time or memory. We present a new anytime algorithm that finds increasingly better solutions and eventually converges to an optimal Bayesian network upon completion. The algorithm is shown to not only improve the runtime to *find* optimal network structures up to 100 times compared to some existing methods, but also prove the optimality of these solutions about 10 times faster in some cases.

Introduction

Early methods for learning a Bayesian network that optimizes a scoring function for a given dataset are mostly approximation algorithms such as greedy hill climbing approaches [1–3]. Even though these approximation algorithms do not guarantee optimality, they do have other nice properties. Because they usually only store the current best network plus local score information, their memory requirements are modest. They also exhibit good anytime behavior, as they can be stopped anytime to output the best solution found so far.

In recent years, several exact algorithms have been developed for learning optimal Bayesian networks, including dynamic programming [4–9], branch and bound (BB) [10], integer linear programming (ILP) [11, 12], and admissible heuristic search [13–15]. These algorithms typically do not possess inherent anytime behavior, that is, they do not find any solution before finding the optimal solution. However, some of these methods can be easily augmented with anytime behavior. For example, BB uses an approximation algorithm to find an upper bound solution in the very beginning of the search for pruning. Also, it regularly strays away from its main best-first search strategy and expands the worst nodes in the priority queue in hope to find better solutions sooner.

This paper develops an *intrinsically* anytime algorithm based on depth-first search. Several key theoretical contributions were needed to make the algorithm effective for the learning problem, including a newly formulated search graph, a new heuristic function, and a duplicate detection and repairing strategy. The algorithm combines the best characteristics of exact (guaranteed optimality when finishes) and local search (quickly identifying good solutions) algorithms. Experimentally, we show that the new algorithm has an excellent anytime behavior. The algorithm continually finds better solutions during the search; it not only *finds* the optimal solutions up to two orders of magnitude faster than some existing algorithms but also *proves* their optimality typically one order of magnitude faster when it finishes.

Background

This section reviews the basics of score-based Bayesian network structure learning and the shortest-path perspective of the learning problem [13], which is the basis of our new algorithm.

Learning Bayesian Network Structures

A Bayesian network is a directed acyclic graph (DAG) in which the vertices correspond to a set of random variables $\mathbf{V} = \{X_1, ..., X_n\}$, and the arcs represent dependence relations between the variables. The set of all parents of X_i are referred to as PA_i. The parameters of the network define a conditional probability distribution, $P(X_i|\text{PA}_i)$, for each X_i.

We consider the problem of learning a network structure from a dataset $\mathbf{D} = \{D_1, ..., D_N\}$, where D_i is an instantiation of all the variables in $\mathbf{V}$. A scoring function measures the goodness of fit of a network structure to $\mathbf{D}$ [2]. The problem is to find the structure which optimizes the score. In this work we used the minimum description length (MDL) [16], so it is minimization problem. Let r_i be the number of states of X_i, N_{pa_i} be the number of records in $\mathbf{D}$ consistent with $PA_i = pa_i$, and N_{x_i,pa_i} be the number of records further constraint with $X_i = x_i$. MDL is defined as follows [17].

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

where

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

Other decomposable scores [2] can also be used, such as BIC [18], AIC [19], BDe [20, 21] and fNML [22]. For many of those scoring functions, the best network

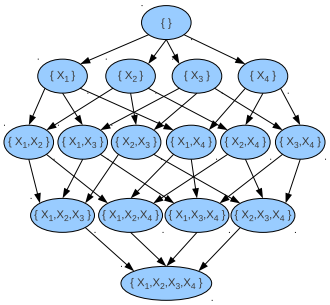

Fig. 1. A forward order graph of four variables.

maximizes the score. We can multiply the scores by -1 to ensure the best network minimizes the score. For the rest of the paper, we assume the local scores, $score(X_i|PA_i)$, are computed prior to the search.

Shortest-path Perspective

Yuan *et al.* ([13]) viewed the learning problem as a shortest-path problem in an *implicit* search graph. Figure 1 shows the 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 $\mathbf{U}$, which is computed by considering all subsets of the variables in $\mathbf{U}$, 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 we also call the graph a *forward 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 cost of a path is equal to the sum of the parent selections entailed by the ordering. The shortest path gives the global optimal structure.

Finding Optimal Parents

While searching the forward order graph, the cost of each visited arc is calculated using *parent graphs*. The parent graph for X consists of all subsets of $\mathbf{V} \setminus \{X\}$. Figure 2(a) shows a parent graph for X_1. Some subsets cannot possibly be the optimal parent set according to the following theorem [23].

Theorem 1. *Let* $\mathbf{U} \subset \mathbf{T}$ *and* $X \notin \mathbf{T}$. *If* $score(X|\mathbf{U}) < score(X|\mathbf{T})$, $\mathbf{T}$ *is not the optimal parent set for* X.

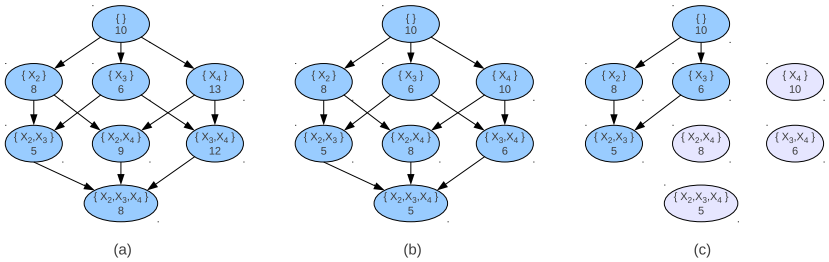

Fig. 2. A sample parent graph for variable X_1. (a) The local scores, $score(X_1, \cdot)$ for all the parent sets. (b) The optimal scores, $BestScore(X_1, \cdot)$, for each candidate parent set. (c) The unique optimal parent sets and their scores. The pruned parent sets are shown in gray. A parent set is pruned if any of its predecessors has a better score.

$parents_{X_1}$	$\{X_2, X_3\}$	$\{X_3\}$	$\{X_2\}$	$\{\}$
$scores_{X_1}$	5	6	8	10

Table 1. Sorted scores and parent sets for X_1 after pruning sets which are not possibly optimal.

While we focus on the MDL score in this paper, the theorem holds for all decomposable scoring functions.

Figure 2(b) shows the optimal scores after propagating $BestScore(\cdot)$ from subsets to supersets. The score of the arc from $\mathbf{U}$ to $\mathbf{U} \cup \{X\}$ in the order graph is given by the node $\mathbf{U}$ in the parent graph for X.

The parent graph for a variable X exhaustively enumerates and stores $BestScore(X, \mathbf{U})$ for all subsets of $\mathbf{V} \setminus \{X\}$. Naively, all the parent graphs require storing $n2^{n-1}$ subsets. Due to Theorem 1, the number of *unique* optimal parent sets is often far smaller than the total number, as the same score can be optimal for many nodes. We can remove the repetitive information by only storing the optimal parent sets and their scores as sorted lists for each parent graph. The results are a set of *sparse parent graphs* [15]. Figure 2(c) shows the sparse parent graph of X_1, and Table 1 shows the corresponding sorted lists. We call these lists $scores_X$ and $parents_X$. When given a candidate parent set $\mathbf{U}$ for X, we find the optimal parents by scanning through $parents_X$ and find the first subset of $\mathbf{U}$ plus its score.

Finding the Shortest Path

Various methods can be applied to solve the shortest path problem. Dynamic programming can be considered to evaluate the order graph using a top down sweep of the graph [6, 9]. The A* algorithm in [13] evaluates the order graph with a best first search guided by an admissible heuristic function. The breadth-first branch and bound (BF-BnB) algorithm in [14] searches the order graph one layer at a time and uses an initial upper bound solution for pruning.

None of these algorithms finds a solution until the very end of the search. If they run out of resources, they yield no solution. We address these limitations by introducing a simple but very effective anytime learning algorithm.

A Depth-first Branch and Bound Algorithm

Our new algorithm applies *depth-first search* (DFS) to solve the shortest path problem. Depth-first search has an intrinsic anytime behavior; each time the depth-first search reaches the goal node, a new solution is found and can potentially be used to update the incumbent solution.

However, applying depth-first search to the learning problem is not as straightforward as expected. Several improvements are needed to make the algorithm effective for the particular problem, including a newly formulated search graph, a new heuristic function, and a duplicate detection and repairing strategy.

Reverse order graph

A key improvement comes from addressing the inefficiency in finding optimal parents in the sparse parent graphs. For a candidate parent set $\mathbf{U}$ for X, we need to find the optimal parents by scanning through $parents_X$ and find the first subset of $\mathbf{U}$ as the optimal parents. The scan is achieved by removing all the non-candidate variables $\mathbf{V} \setminus \mathbf{U}$ and is best implemented using bit vectors and bitwise operations. Readers are referred to [15] for more details because of the lack of space. It suffices to say here that bit vectors are used to keep track of optimal parent sets that contain allowable candidate parents, and bitwise operations are applied to the vectors to remove non-candidate variables one at a time.

For example, the first score in Table 1 is optimal for the candidate parent set $\{X_2, X_3, X_4\}$. To remove X_2 from the candidate set and find $BestScore(X_1, \{X_3, X_4\})$, we scan $parents_X$ from the beginning. The first parent set which does not include X_2 is $\{X_3\}$. To remove both X_2 and X_3 and find $BestScore(X_1, \{X_4\})$, we scan until finding a parent set which includes neither X_2 nor X_3; that is $\{\}$. In the worst case, a complete scan of the list is needed, which makes each search step expensive.

The scan can be made more efficient if only one variable needs to be removed at each search step by reusing results from previous steps. Depth-first search enables us to achieve that. We can store the bit vectors used in previous scans in the *search stack*. Since each search step in the order graph only processes one more variable, the search can start from the bit vectors on top of the stack and remove only one more variable to find the optimal parents. Also, depth-first search makes it necessary to store only at most $O(n^2)$ bit vectors at any time. Other search methods may require an exponential number of bit vectors to use the same incremental strategy.

However, the forward order graph is not very amenable to the above incremental scan, because variables are added as candidate parents as the search goes deeper. We therefore propose a newly formulated search graph. We notice that there is symmetry between the start and goal nodes in the forward order graph; the shortest path from the start node to the goal node is the same as the shortest path from the latter to the former. The reversed direction of search, however, is better because variables are removed incrementally. Therefore, we propose to use the *reverse order graph* shown in Figure 3 as our search graph. The top-most node with the complete set is the new start node, and the bottom-most empty set is the new goal node. An arc from $\mathbf{U}$ to $\mathbf{U} \setminus \{X\}$ represents making X a leaf node for the subnetwork over $\mathbf{U}$. The cost of the arc is

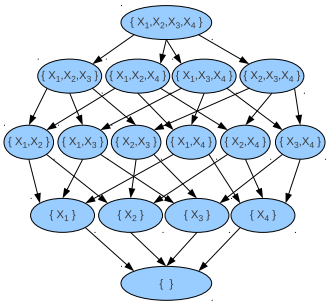

Fig. 3. A reverse order graph of four variables.

$BestScore(X, \mathbf{U} \setminus \{X\})$. The task is to find the shortest path between the new start and goal nodes.

Branch and Bound

The efficiency of the depth-first search can be significantly improved by pruning. The best solution found so far is an upper bound for the optimal solution. If we can also estimate a lower bound h on the cost from the current search node to the goal, then we can calculate a lower bound on the cost of a solution using the current path to this node. If that lower bound is worse than the upper bound solution, then the current path will not lead to any better solution.

Since the reverse order graph has a different goal node from the forward order graph, we cannot use the same heuristic function in [14, 13]. We therefore design a new admissible heuristic function. At any point in the search, a set of variables remain for which we must select parents. Addtionally, we know that, at a node $\mathbf{U}$, the parent selections for the remaining variables are a subset of $\mathbf{U}$. Therefore, $BestScore(X, \mathbf{U})$ is a lower bound on the scores for each remaining variable, X. We calculate $BestScore(X, \mathbf{U})$ for the remaining X at node $\mathbf{U}$ with the bit vectors on top of the search stack. Summing over these scores gives a lower bound on the optimal subnetwork over $\mathbf{U}$. More formally, we use the following new heuristic function h^*.

Definition 1.

$$h^*(\mathbf{U}) = \sum_{X \in \mathbf{U}} BestScore(X, \mathbf{U} \setminus \{X\}) \tag{2}$$

The heuristic is admissible because it allows the remaining variables to select optimal parents from among all of the other remaining variables. This relaxes the acyclic constraint on those variables. The following theorem shows the heuristic is also consistent (proof is in the appendix).

Theorem 2. *h^* is consistent.*

Duplicate detection and repair

A potential problem with DFS is generating duplicate nodes. A traditional DFS algorithm does not perform duplicate detection; much work can be wasted in re-expanding duplicate nodes. Our search graph contains many duplicates; a node in layer l can be generated by all its l predecessor nodes. In order to combat this problem, our algorithm stores nodes that have been expanded in a hash table and detects duplicate nodes during the search. If a better path to a generated node has already been found, we can immediately discard the duplicate.

On the other hand, we need to re-expand a node if a better path is found to it. We efficiently handle re-expansions by adopting an *iterative* DFS strategy. During each iteration, we expand each node only once. We keep a list which stores all nodes to which we find a better path. We expand those nodes in the next iteration and iterate this process until no nodes are added to the list.

We further improve the anytime behavior of the algorithm by storing the cost of the best discovered path from $\mathbf{U}$ to $goal$, $h_{exact}(\mathbf{U})$. When backtracking, we compute $h_{exact}(\mathbf{U})$ by calculating, for each successor $\mathbf{R}$, the total distance between $\mathbf{U}$ and $\mathbf{R}$ and between $\mathbf{R}$ and the goal. The minimum distance among them is $h_{exact}(\mathbf{U})$. Trivially, h_{exact} of an immediate predecessor of $goal$ is just the distance from it to $goal$. We pass this information up the call stack to calculate h_{exact} for predecessor nodes. The next time $\mathbf{U}$ is generated, we sum the distance on the current path and $h_{exact}(\mathbf{U})$. If it is better than the existing best path, $optimal$, then we update the best path found so far. We store h_{exact} in the same hash table used for duplicate detection.

Depth-first Search

Algorithm 1 (at the end of the paper) provides the pseudocode for the DFS algorithm. After initializing the data structures (lines 25 - 27), the EXPAND function is called with the start node as input. At each node $\mathbf{U}$, we make one variable X as a leaf (line 5) and select its optimal parents from among $\mathbf{U}$ (lines 6, 7). We then check if that is the best path to the subnetwork $\mathbf{U} \setminus \{X\}$ (lines 10 - 12). The bitwise operators are used to remove X as a possible parent for the remaining variables (lines 13 - 15). Parents are recursively selected for the remaining variables (line 16). Because we did not modify $valid$, the call stack maintains the valid parents before removing X, so we perform the bitwise operators for the next leaf. After trying every remaining variable as a leaf, we check if a better path to the goal has been found (line 22). During each iteration of the search, we keep a list that tracks nodes to which we find a better path (line 10) and repair those nodes in the next iteration by iteratively performing the DFS (lines 29 - 37). The search continues until no nodes are added to the list.

The depth-first nature of the algorithm means we typically find a path to the goal after n node expansions; it can be stopped at any time to return the best path found so far. For conciseness, Algorithm 1 only includes the main logic in computing the optimal score; we use the network reconstruction technique described in [14].

Algorithm 1 A DFBnB Search Algorithm

```
1: procedure EXPAND(U, valid, toRepair)
2:     for each X ∈ U do
3:         BestScore(X, U \ {X}) ← scores_X[firstSetBit(valid_X)]
4:         g ← g(U) + BestScore(X, U \ {X})
5:         duplicate ← exists(g(U \ {X}))
6:         if g < g(U \ {X}) then g(U \ {X}) ← g
7:         if duplicate and g < g(U \ {X}) then toRepair ← toRepair ∪ {U, g}
8:         f ← h(U \ {X}) + g(U \ {X})
9:         if !duplicate and f < optimal then
10:            for each Y ∈ U do
11:                valid'_Y ← valid_Y & ∼ parents_Y(X)
12:            end for
13:            expand(U \ {X}, valid')
14:        end if
15:        if h_exact(U) > BestScore(X, U \ {X}) + h_exact(U \ {X}) then
16:            h_exact(U) ← BestScore(X, U \ {X}) + h_exact(U \ {X})
17:        end if
18:    end for
19:    if optimal > h_exact(U) + g(U) then optimal ← h_exact(U) + g(U)
20: end procedure

21: procedure MAIN(D)
22:    for each X ∈ V do
23:        scores_X, parents_X ← initDataStructures(X, D)
24:    end for
25:    h_exact(U) ← 0
26:    toRepair_l ← {V, 0}
27:    while |toRepair_l| > 0 do
28:        for each {V, g} ∈ toRepair_l do
29:            if g(V) > g then
30:                g(V) ← g
31:                expand(V, valid, toRepair_{l+1})
32:            end if
33:        end for
34:        toRepair_l ← toRepair_{l+1}
35:    end while
36: end procedure
```

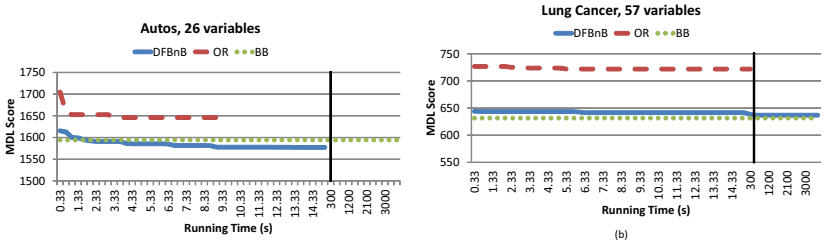

Fig. 4. The scores of the best networks learned by DFBnB, OR and BB on two datasets over time. The algorithms are given up to one hour (3600 s) of runtime. A black bar indicates an increase in the time scale. OR reached a local optimum and terminated within 300s. DFBnB found and proved the optimal solution for the Autos dataset within the time limit. BB did not improve its initial solution or find the optimal solution within the time limit for either dataset.

Experiments

We tested the DFBnB algorithm against several existing structure learning algorithms: dynamic programming (DP) [6], breadth-first branch and bound (BFBnB) [14], branch and bound (BB) [10], and optimal reinsertion (OR) [3]. The source code for DFBnB is available online (http://url.cs.qc.cuny.edu/software/URLearning.html). Experiments were performed on a PC with 3.07 GHz Intel i7, 16 GB RAM and 500 GB hard disk. We used benchmark datasets from UCI [24]. Records with missing values were removed. Variables were discretized into two states.

Comparison of Anytime Behavior

We first compared the anytime behavior of DFBnB to those of BB and OR on two datasets of up to 57 variables. Other datasets that we tested show similar results. We chose to compare to BB because, like DFBnB, it has anytime behavior and optimality guarantees. We compare to OR as a representative for local search techniques because several studies [23, 25] suggest that it performs well. We ran each algorithm for one hour and plotted the scores of the networks learned by each algorithm as a function of time in Figure 4.

As the convergence curves of these algorithms show, OR was always the first algorithm to terminate; it reached local optima far from optimal and was unable to escape. BB did not finish within the time limit on either dataset. The convergence curves of BB stayed flat for both. BB used a sophisticated approximation algorithm to find its initial solution. That means BB was not able to improve its initial solutions in an hour of runtime, so the true anytime behavior of BB is unclear from the results. In comparison, DFBnB intrinsically finds all solutions, so its curves provide a reliable indication of its anytime behavior. On both datasets, DFBnB continually found better solutions during the search, and was able to find and prove the optimality of its solution on the Autos dataset within 5 minutes. BB learned a network whose score was 0.7% lower than DFBnB on the Lung Cancer dataset.

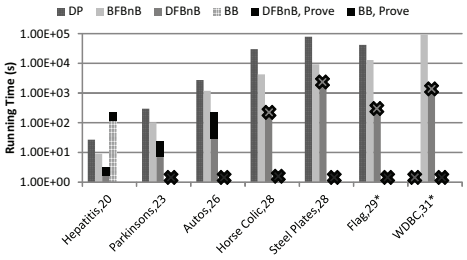

Fig. 5. A comparison on the running time for BFBnB, DP, DFBnB and BB. DP and BFBnB were given an arbitrary amount of time to complete because of their efficient use of external memory. DFBnB and BB were given a maximum running time of one hour. For DP, an "X" for a dataset means the external memory usage exceeded 500 GB. For DFBnB and BB, an "X" means it did not prove the optimality of the solution found. An "X" at time 0 for BB means that BB did not improve its initial greedy solution and did not find the optimal solution. DFBnB found the optimal solution, though may not have proved it, for all datasets which do not have an "*" beside the name.

Comparison of Running Time

We compared the running time of DFBnB to those of DP, BFBnB and BB. DFBnB and BB were again given a one hour limit. We let BFBnB and DP run longer in order to obtain the optimal solutions for evaluation. For this comparison, we considered both the time to *find* the best structure and to *prove* its optimality by DFBnB and BB. The results in Figure 5 demonstrate that DFBnB finds the optimal solution nearly two orders of magnitude faster than the other algorithms, even though it was not always able to prove optimality due to lack of RAM. When it had enough RAM, though, DFBnB proved optimality nearly an order of magnitude faster that other algorithms.

The latter result is somewhat surprising given that BFBnB is based a similar shortest-path formulation and was shown to be as efficient as A* [14]. The impressive performance of DFBnB comes from the incremental scan technique used to find optimal parents in the sparse parent graphs. In comparison, each step of BFBnB and A* starts from scratch and removes all non-candidate variables to find optimal parents.

However, DFBnB does not take advantage of disk. The program stops when its hash table fills RAM, so it is unable to prove the optimality for some of the searches. Nevertheless, the search finds optimal solutions for all but the largest two datasets (verified with BFBnB). BB proved the optimality of its solution on only the smallest dataset. On all other cases, it did not improve its initial solutions. DFBnB found better solutions than BB on all these datasets. Note Lung Cancer is not included here because none of the algorithms proved the optimality of their solutions within the resource limits.

Conclusion

In this paper we have proposed an anytime algorithm for learning exact Bayesian network structures. The algorithm introduces a new shortest-path formulation of the problem to use depth-first search. We introduced the reverse order graph and a new heuristic

function to improve the efficiency of the search algorithm. A duplicate detection and repair strategy is also used to avoid unnecessary re-expansions while maintaining excellent anytime behavior.

Experimental results demonstrated improved anytime behavior of our algorithm compared to some other structure learning algorithms. Our algorithm often finds good solutions more quickly than the other anytime algorithms, even though the local search techniques forgo optimality guarantees. Optimal reinsertion is one of the best performing approximation methods, but our results show that it often finds solutions that are far from optimal.

Our algorithm is also guaranteed to converge to an optimal solution when it has enough RAM. In those cases, it proves the optimality of its solution almost an order of magnitude faster than other search algorithms. Consequently, our algorithm combines the best qualities of both globally optimal algorithms and local search algorithms.

Appendix

Proof of Theorem 2.

Proof. For any successor node $\mathbf{R}$ of $\mathbf{U}$, let $Y \in \mathbf{U} \setminus \mathbf{R}$, let $c(\mathbf{U}, \mathbf{R})$ be the cost of using Y as the leaf of $\mathbf{U}$. We have

$$\begin{aligned} h^*(\mathbf{U}) &= \sum_{X \in \mathbf{U}} BestScore(X, \mathbf{U} \setminus \{X\}) \\ &\leq \sum_{X \in \mathbf{U}, X \neq Y} BestScore(X, \mathbf{U}) + BestScore(Y, \mathbf{U}) \\ &\leq \sum_{X \in \mathbf{R}} BestScore(X, \mathbf{R} \setminus \{X\}) + BestScore(Y, \mathbf{U} \setminus \{Y\}) \\ &= h^*(\mathbf{R}) + c(\mathbf{U}, \mathbf{R}). \end{aligned}$$

The inequality holds because the variables in $\mathbf{R}$ have fewer parents to choose from after making Y a leaf. Hence, h^* is consistent.

Acknowledgements This work was supported by the Academy of Finland (Finnish Centre of Excellence in Computational Inference Research COIN, 251170) and NSF grants IIS-0953723 and IIS-1219114.

.

References

1. Cooper, G.F., Herskovits, E.: A bayesian method for the induction of probabilistic networks from data. Mach. Learn. **9** (October 1992) 309–347
2. Heckerman, D.: A tutorial on learning with Bayesian networks. Technical report, Learning in Graphical Models (1996)
3. Moore, A., Wong, W.K.: Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning. In: Intl. Conf. on Machine Learning. (2003) 552–559

4. Koivisto, M., Sood, K.: Exact Bayesian structure discovery in Bayesian networks. Journal of Machine Learning Research (2004) 549–573
5. Ott, S., Imoto, S., Miyano, S.: Finding optimal models for small gene networks. In: Pac. Symp. Biocomput. (2004) 557–567
6. Silander, T., Myllymaki, P.: 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 (2006)
7. Singh, A., Moore, A.: Finding optimal Bayesian networks by dynamic programming. Technical report, Carnegie Mellon University (June 2005)
8. Parviainen, P., Koivisto, M.: Exact structure discovery in Bayesian networks with less space. In: Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, Montreal, Quebec, Canada, AUAI Press (2009)
9. Malone, B., Yuan, C., Hansen, E.: Memory-efficient dynamic programming for learning optimal Bayesian networks. In: Proceedings of the 25th national conference on Artifical intelligence. (2011)
10. de Campos, C.P., Ji, Q.: Efficient learning of bayesian networks using constraints. Journal of Machine Learning Research **12** (2011) 663–689
11. Cussens, J.: Bayesian network learning with cutting planes. In: Proceedings of the Proceedings of the Twenty-Seventh Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-11), Corvallis, Oregon, AUAI Press (2011) 153–160
12. Jaakkola, T., Sontag, D., Globerson, A., Meila, M.: Learning Bayesian network structure using LP relaxations. In: Proc. of the 13th International Conference on Artificial Intelligence and Statistics. (2010)
13. Yuan, C., Malone, B., Wu, X.: Learning optimal Bayesian networks using A* search. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence. (2011)
14. Malone, B., Yuan, C., Hansen, E., Bridges, S.: Improving the scalability of optimal Bayesian network learning with external-memory frontier BFBnB search. In: Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Corvallis, Oregon, AUAI Press (2011) 479–488
15. Yuan, C., Malone, B.: An improved admissible heuristic for learning optimal Bayesian networks. In: Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI-12), Catalina Island, CA (2012)
16. Lam, W., Bacchus, F.: Learning Bayesian belief networks: An approach based on the MDL principle. Computational Intelligence **10** (1994) 269–293
17. Tian, J.: A branch-and-bound algorithm for MDL learning Bayesian networks. In: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann Publishers Inc. (2000) 580–588
18. Schwarz, G.: Estimating the dimension of a model. **6** (1978) 461–464
19. Akaike, H.: Information theory and an extension of the maximum likelihood principle. In: Proceedings of the Second International Symposium on Information Theory. (1973) 267–281
20. Buntine, W.: Theory refinement on Bayesian networks. In: Proc. of the Seventh conference (1991) on Uncertainty in artificial intelligence, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1991) 52–60
21. Heckerman, D., Geiger, D., Chickering, D.M.: Learning Bayesian networks: The combination of knowledge and statistical data. **20** (1995) 197–243
22. Silander, T., Roos, T., Kontkanen, P., Myllymaki, P.: Factorized normalized maximum likelihood criterion for learning Bayesian network structures. In: Proceedings of the 4th European Workshop on Probabilistic Graphical Models (PGM-08). (2008) 257–272
23. Teyssier, M., Koller, D.: Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In: Proceedings of the Twenty-First Conference Annual Conference

on Uncertainty in Artificial Intelligence (UAI-05), Arlington, Virginia, AUAI Press (2005) 584–590
24. Frank, A., Asuncion, A.: UCI machine learning repository (2010)
25. Tsamardinos, I., Brown, L., Aliferis, C.: The max-min hill-climbing Bayesian network structure learning algorithm. Machine Learning **65** (2006) 31–78 10.1007/s10994-006-6889-7.