Machine learning of Bayesian networks using constraint programming

Peter van Beek and Hella-Franziska Hoffmann

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
vanbeek@cs.uwaterloo.ca

Abstract. Bayesian networks are a widely used graphical model with diverse applications in knowledge discovery, classification, prediction, and control. Learning a Bayesian network from discrete data can be cast as a combinatorial optimization problem and there has been much previous work on applying optimization techniques including proposals based on ILP, A* search, depth-first branch-and-bound (BnB) search, and breadth-first BnB search. In this paper, we present a *constraint-based* depth-first BnB approach for solving the Bayesian network learning problem. We propose an improved constraint model that includes powerful dominance constraints, symmetry-breaking constraints, cost-based pruning rules, and an acyclicity constraint for effectively pruning the search for a minimum cost solution to the model. We experimentally evaluated our approach on a representative suite of benchmark data. Our empirical results compare favorably to the best previous approaches, both in terms of number of instances solved within specified resource bounds and in terms of solution time.

1 Introduction

Bayesian networks are a popular probabilistic graphical model with diverse applications including knowledge discovery, classification, prediction, and control (see, e.g., [1]). A Bayesian network (BN) can either be constructed by a domain expert or learned automatically from data. Our interest here is in the learning of a BN from discrete data, a major challenge in machine learning. Learning a BN from discrete data can be cast as a combinatorial optimization problem—the well-known *score-and-search* approach—where a scoring function is used to evaluate the quality of a proposed BN and the space of feasible solutions is systematically searched for a best-scoring BN. Unfortunately, learning a BN from data is NP-hard, even if the number of parents per vertex in the DAG is limited to two [2]. As well, the problem is unlikely to be efficiently approximatable with a good quality guarantee, thus motivating the use of global (exact) search algorithms over local (heuristic) search algorithms [3].

Global search algorithms for learning a BN from data have been studied extensively over the past several decades and there have been proposals based on dynamic programming [4–6], integer linear programming (ILP) [7, 8], A* search [9–11], depth-first branch-and-bound (BnB) search [12, 13], and breadth-first BnB search [14, 10, 15, 11]. In this paper, we present a *constraint-based* depth-first BnB approach for solving the

Bayesian network learning problem. We propose an improved constraint model that includes powerful dominance constraints, symmetry-breaking constraints, cost-based pruning rules, and an acyclicity constraint for effectively pruning the search for a minimum cost solution to the model. We experimentally evaluated our approach on a representative suite of benchmark data. Our empirical results compare favorably to the best previous approaches, both in terms of number of instances solved within specified resource bounds and in terms of solution time.

2 Background

In this section, we briefly review the necessary background in Bayesian networks before defining the Bayesian network structure learning problem (for more background on these topics see, for example, [16, 17]).

A Bayesian network (BN) is a probabilistic graphical model that consists of a labeled directed acyclic graph (DAG) in which the vertices $V = \{v_1, \ldots, v_n\}$ correspond to random variables, the edges represent direct influence of one random variable on another, and each vertex v_i is labeled with a conditional probability distribution $P(v_i \mid \mathit{parents}(v_i))$ that specifies the dependence of the variable v_i on its set of parents $\mathit{parents}(v_i)$ in the DAG. A BN can alternatively be viewed as a factorized representation of the joint probability distribution over the random variables and as an encoding of conditional independence assumptions.

The predominant method for BN structure learning from data is the *score-and-search* method[1]. Let G be a DAG over random variables V, and let $I = \{I_1, \ldots, I_N\}$ be a set of multivariate discrete data, where each instance I_i is an n-tuple that is a complete instantiation of the variables in V. A *scoring function* $\sigma(G \mid I)$ assigns a real value measuring the quality of G given the data I. Without loss of generality, we assume that a lower score represents a better quality network structure.

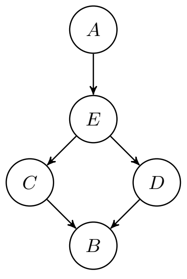
Definition 1. *Given a discrete data set $I = \{I_1, \ldots, I_N\}$ over random variables V and a scoring function σ, the* Bayesian network structure learning problem *is to find a directed acyclic graph G over V that minimizes the score $\sigma(G \mid I)$.*

Scoring functions commonly balance goodness of fit to the data with a penalty term for model complexity to avoid overfitting. Common scoring functions include BIC/MDL [18, 19] and BDeu [20, 21]. An important property of these (and all commonly used) scoring functions is decomposability, where the score of the entire network $\sigma(G \mid I)$ can be rewritten as the sum of local scores $\sum_{i=1}^{n} \sigma(v_i, \mathit{parents}(v_i) \mid I)$ that only depend on v_i and the parent set of v_i in G. Henceforth, we assume that the scoring function is decomposable and that, following previous work, the local score $\sigma(v_i, p \mid I)$ for each possible parent set $p \subseteq 2^{V - \{v_i\}}$ and each random variable v_i has been computed in a preprocessing step prior to the search for the best network structure. Pruning

[1] An alternative method, called *constraint-based* structure learning in the literature, is based on statistical hypothesis tests for conditional independence. We do not discuss it further here except to note that the method is known to scale to large instances but to have the drawback that it is sensitive to a single failure in a hypothesis test (see [17, p. 785]).

A :	$\{D\}$, 9.6	$\{C\}$, 9.9	$\{E\}$, 10.0	$\{\}$, 15.4	
B :	$\{C, D\}$, 12.1	$\{C\}$, 12.2	$\{E\}$, 12.3	$\{\}$, 14.1	
C :	$\{E\}$, 3.6	$\{D\}$, 5.2	$\{A, B\}$, 10.9	$\{A\}$, 11.4	$\{\}$, 17.0
D :	$\{E\}$, 3.6	$\{C\}$, 5.2	$\{A, B\}$, 10.9	$\{A\}$, 11.4	$\{\}$, 17.0
E :	$\{D\}$, 3.7	$\{A\}$, 4.2	$\{A, B\}$, 11.2	$\{C\}$, 11.6	$\{\}$, 17.0

(a)



(b)

Fig. 1. (a) Random variables and possible parent sets for Example 1; (b) minimum cost DAG structure with cost 38.9.

techniques can be used to reduce the number of possible parent sets that need to be considered, but in the worst-case the number of possible parent sets for each variable v_i is 2^{n-1}, where n is the number of vertices in the DAG.

Example 1. Let A, B, C, D, and E be random variables with the possible parent sets and associated scores shown in Figure 1(a). For example, if the parent set $\{C, D\}$ for random variable B is chosen there would be a directed edge from C to B and a directed edge from D to B and those would be the only incoming edges to B. The local score for this parent set is 12.1. If the parent set $\{\}$ for random variable A is chosen, there would be no incoming edges to A; i.e., A would be a source vertex. Figure 1(b) shows the minimum cost DAG with cost $15.4 + 4.2 + 3.6 + 3.6 + 12.1 = 38.9$.

3 Constraint Programming Approach

In this section, we present a constraint model and a depth-first branch-and-bound solver for the Bayesian network structure learning problem. Table 1 summarizes the notation.

Our constraint model consists of vertex variables, ordering variables, depth variables, and constraints over those variables. The ordering and depth variables, although redundant, improve the search for a solution.

Vertex (possible parent set) variables. There is a vertex variable v_i, $i \in V$, for each random variable in V and the domain of v_i, $dom(v_i)$, consists of the possible parent sets for v_i. The assignment $v_i = p$ denotes that vertex v_i has parents p in the DAG; i.e., the vertex variables represent the DAG over the set of random variables V. Associated with each domain value is a cost and the goal is to minimize the total cost, $cost(v_1) + \cdots + cost(v_n)$, subject to the constraint that the graph is acyclic. A global constraint is introduced to enforce that the vertex variables form a DAG,

$$\text{acyclic}(v_1, \ldots, v_n), \tag{1}$$

where the constraint is satisfied if and only if the graph designated by the parent sets is acyclic. The DAG is not necessarily connected. A satisfiability checker for the acyclic constraint is given in Section 3.7, which in turn can be used to propagate the constraint.

Table 1. Notation for specifying constraint model.

V	set of random variables
n	number of random variables in the data set
$cost(v)$	cost (score) of variable v
$dom(v)$	domain of v
$parents(v)$	set of parents of v in the DAG
$\min(dom(v))$	the minimum value in the domain of v
$v_1, \ldots, v_n$	vertex (possible parent set) variables
$o_1, \ldots, o_n$	ordering (permutation) variables
$d_1, \ldots, d_n$	depth variables
$depth(p \mid o_1, \ldots, o_{i-1})$	depth of $p \in dom(v_j)$, where v_j occurs at position i in the ordering

Ordering (permutation) variables. There is an ordering variable o_i for each random variable and $dom(o_i) = V$, the set of random variables. The assignment $o_i = j$ denotes that vertex v_j is in position i in the total ordering of the variables. The ordering variables represent a permutation of the random variables. A global constraint is introduced to enforce that the order variables form a permutation of the vertex variables,

$$\text{alldifferent}(o_1, \ldots, o_n). \tag{2}$$

The alldifferent constraint is propagated by, whenever a variable becomes instantiated, simply removing that value from the domains of the other variables.

Depth variables. There is a depth variable d_i for each random variable and $dom(d_i)$ = $\{0, ..., n-1\}$. The depth variables and the ordering variables are in one-to-one correspondence. The assignment $d_i = k$ denotes that the depth of the vertex variable v_j that occurs at position i in the ordering is k, where the depth is the length of the longest path from a source vertex to vertex v_j in the DAG.

Example 2. A constraint model for Example 1 would have variables v_A, …, v_E, o_1, …, o_5, and d_1, …, d_5, with domains $dom(v_A)$ = $\{\{C\}, \{D\}, \{E\}, \{\}\}$, …, $dom(v_E)$ = $\{\{D\}, \{A\}, \{A, B\}, \{C\}, \{\}\}$, $dom(o_i)$ = $\{A, \ldots, E\}$, and $dom(d_i)$ = $\{0, \ldots, 4\}$.

To more formally state additional constraints, we introduce the following notation for the depth of a domain value p for a vertex variable v_j.

Definition 2. *The* depth *of a domain value p for a vertex variable v_j that occurs at position i in the ordering, denoted $depth(p \mid o_1, \ldots, o_{i-1})$, is defined as: 0 if $p = \{\}$; one plus the maximum depth of the elements of p if each element of p occurs in a parent set of a vertex variable earlier in the ordering; and ∞ otherwise.*

Example 3. In Example 2, suppose that $o_1 = A$, $v_A = \{\}$, $d_1 = 0$, $o_2 = E$, $v_E = \{A\}$, and $d_2 = 1$. The value of $depth(p \mid o_1, o_2)$ for variable C is 0 if $p = \{\}$, 1 if $p = \{A\}$, 2 if $p = \{E\}$, and ∞ if $p = \{D\}$ or $p = \{A, B\}$.

Constraints 3 & 4 establish the correspondence between the three types of variables,

$$\forall j \bullet \forall p \bullet v_j = p \iff \exists! i \bullet o_i = j \land d_i = depth(p \mid o_1, \ldots, o_{i-1}), \tag{3}$$

$$\forall i \bullet \forall j \bullet o_i = j \iff \exists! p \bullet v_j = p \land d_i = depth(p \mid o_1, \ldots, o_{i-1}), \tag{4}$$

where the indices i and j range over $1 \leq i, j \leq n$ and the value p ranges over $dom(v_j)$. The constraints are propagated as follows. A value $p \in dom(v_j)$ can be pruned iff $\forall i \bullet j \in dom(o_i) \Rightarrow depth(p \mid o_1, \ldots, o_{i-1}) \notin dom(d_i)$. A value $j \in dom(o_i)$ can be pruned iff $\forall p \in dom(v_j) \bullet depth(p \mid o_1, \ldots, o_{i-1}) \notin dom(d_i)$. Only bounds are maintained on the depth variables. Hence, the notation $depth(p \mid o_1, \ldots, o_{i-1}) \notin dom(d_i)$ is to be interpreted as $depth(p \mid o_1, \ldots, o_{i-1}) < \min(dom(d_i)) \vee depth(p \mid o_1, \ldots, o_{i-1}) > \max(dom(d_i))$. When propagating Constraints 3 & 4, we must determine $depth(p \mid o_1, \ldots, o_{i-1})$. In general, this is a difficult problem. We restrict ourselves to two easy special cases: (i) all of $o_1, \ldots, o_{i-1}$ have been instantiated, or (ii) some of $o_1, \ldots, o_{i-1}$ have been instantiated and all of the $p \in dom(v_j)$ are subsets of these ordering variables. We leave to future work further ways of safely approximating the depth to allow further propagation.

Example 4. In Example 2, suppose that $o_1 = A$, $v_A = \{\}$, $d_1 = 0$, $o_2 = E$, $v_E = \{A\}$, $d_2 = 1$, and that, as a result of some propagation, $\min(d_i) = 1$, $i = 3, 4, 5$. The value $\{\}$ can be pruned from each of the domains of v_B, v_C, and v_D.

One can see that the vertex variables together with the acyclic constraint are sufficient alone to model the Bayesian network structure learning problem. Such a search space over DAGs forms the basis of Barlett and Cussens' integer programming approach [8]. One can also see that the ordering (permutation) variables together with the alldifferent constraint are sufficient alone, as the minimum cost DAG for a given ordering is easily determinable. Larranaga et al. [22] were the first to propose the search space of all permutations and Teyssier and Koller [23] successfully applied it within a local search algorithm. The permutation search space also forms the basis of the approaches based on dynamic programming [4–6] and on the approaches based on searching for the shortest path in an ordering graph using A* search [9–11], depth-first branch-and-bound DFBnB search [13], and BFBnB search [10, 15, 11].

The unique aspects of our model lie in combining the DAG and permutation search spaces and introducing the depth variables. As is shown in the next sections, the combination of variables allows us to identify and post many additional constraints that lead to a considerable reduction in the search space.

3.1 Symmetry-breaking constraints (I)

Many permutations and prefixes of permutations, as represented by the ordering variables, are symmetric in that they lead to the same minimum cost DAG, or are dominated in that they lead to a DAG of equal or higher cost. The intent behind introducing the auxiliary depth variables is to rule out all but the lexicographically least of these permutations. A lexicographic ordering is defined over the depth variables—and over the ordering variables, in the case of a tie on the values of the depth variables. The following constraints are introduced to enforce the lexicographic order.

$$d_1 = 0 \tag{5}$$

$$d_i = k \iff (d_{i+1} = k \vee d_{i+1} = k + 1), \qquad i = 1, \ldots, n-1 \tag{6}$$

$$d_i = d_{i+1} \implies o_i < o_{i+1}, \qquad i = 1, \ldots, n-1 \tag{7}$$

The constraints are readily propagated. Constraint 6 is a dominance constraint.

Example 5. In Example 2, consider the ordering prefix $(o_1, \ldots, o_4) = (E, C, A, D)$ with associated vertex variables $(v_E, v_C, v_A, v_D) = (\{\}, \{E\}, \{C\}, \{E\})$ and depths $(d_1, \ldots, d_4) = (0, 1, 2, 1)$. The cost of this ordering prefix is 33.8. The ordering prefix violates Constraint 6. However, the ordering prefix $(o_1, \ldots, o_4) = (E, C, D, A)$ with depths $(d_1, \ldots, d_4) = (0, 1, 1, 2)$ and vertex variables $(v_E, \ldots, v_D) = (\{\}, \{E\}, \{E\}, \{D\})$ satisfies the constraint and has a lower cost of 33.5.

Constraint 7 is a symmetry-breaking constraint.

Example 6. In Example 2, consider the ordering $(o_1, \ldots, o_5) = (A, E, D, C, B)$ with $(d_1, \ldots, d_5) = (0, 1, 2, 2, 3)$ and $(v_A, \ldots, v_B) = (\{\}, \ldots, \{C, D\})$. The ordering violates Constraint 7. However, the symmetric ordering $(o_1, \ldots, o_5) = (A, E, C, D, B)$, which represents the same DAG, satisfies the constraint and has equal cost.

Theorem 1. *Any ordering prefix $o_1, \ldots, o_i$ can be safely pruned if the associated depth variables $d_1, \ldots, d_i$ do not satisfy Constraints 5–7.*

3.2 Symmetry-breaking constraints (II)

In the previous section, we identified symmetries and dominance among the ordering variables. In this section, we identify symmetries among the vertex variables. Let $[x/y]\,dom(v)$ be the domain that results from replacing all occurrences of y in the parent sets by x. Two vertex variables v_1 and v_2 are symmetric if $[v_1/v_2]dom(v_1) = dom(v_2)$; i.e., the domains are equal once the given substitution is applied. The symmetry is broken by enforcing that v_1 must precede v_2 in the ordering,

$$\forall i \bullet \forall j \bullet o_i = 1 \land o_j = 2 \implies i < j. \tag{8}$$

Example 7. In Example 2, consider vertex variables v_C and v_D. The variables are symmetric as, $[v_C/v_D]dom(v_C) = \{\{E\}, \{C\}, \{A, B\}, \{A\}, \{\}\} = dom(v_D)$.

3.3 Symmetry-breaking constraints (III)

A BN can be viewed as an encoding of conditional independence assumptions. Two BN structures (DAGS) are said to be I-equivalent if they encode the same set of conditional independence assumptions (see, e.g., [16, 17]). The efficiency of the search for a minimal cost BN can be greatly improved by recognizing I-equivalent partial (non-)solutions. Chickering [24, 25] provides a local transformational characterization of equivalent BN structures based on covered edges that forms the theoretical basis of these symmetry-breaking constraints.

Definition 3 (Chickering [24]). *An edge $x \to y$ in a Bayesian network is a* covered *edge if $parents(y) = parents(x) \cup \{x\}$.*

Theorem 2 (Chickering [24]). *Let G be any DAG containing the edge $x \to y$, and let G' be the directed graph identical to G except that the edge between x and y in G' is oriented as $y \to x$. Then G' is a DAG that is equivalent to G if and only if $x \to y$ is a covered edge in G.*

Example 8. In Figure 1(b) the edge $A \rightarrow E$ is a covered edge and the Bayesian network with the edge reversed to be $E \rightarrow A$ is an I-equivalent Bayesian network.

In what follows, we identify three cases that consist of sequences of one or more covered edge reversals and break symmetries by identifying a lexicographic ordering. Experimental evidence suggests that these three cases capture much of the symmetry due to I-equivalence. Symmetries are only broken if the costs of the two I-equivalent DAGs would be equal; otherwise there is a negative interaction with pruning based on the cost function (discussed in Section 3.8).

Case 1. Consider vertex variables v_i and v_j. If there exists domain values $p \in dom(v_i)$ and $p \cup \{v_i\} \in dom(v_j)$, this pair of assignments includes a covered edge $v_i \rightarrow v_j$; i.e., v_i and v_j would have identical parents except that v_i would also be a parent of v_j. Thus, there exists an I-equivalent DAG with the edge reversed. We keep only the lexicographically least: the pair of assignments would be permitted iff $i < j$.

Case 2. Consider vertex variables v_i, v_j, and v_k. If there exists domain values $p \in dom(v_i)$, $p \cup \{v_i\} \in dom(v_j)$, $p \cup \{v_j\} \in dom(v_k)$, where $i < j$ and $k < j$, there is a covered edge $v_i \rightarrow v_j$ and, if this covered edge is reversed, the covered edge $v_j \rightarrow v_k$ is introduced, which in turn can be reversed. Thus, there exists an I-equivalent DAG with the edges $\{v_i \rightarrow v_j, v_j \rightarrow v_k\}$ and an I-equivalent DAG with the edges $\{v_k \rightarrow v_j, v_j \rightarrow v_i\}$. We keep only the lexicographically least: the triple of assignments would be permitted iff $i < k$.

Case 3. Consider vertex variables v_i, v_j, v_k, and v_l. If there exists domain values $p \in dom(v_i), p \cup \{v_i\} \in dom(v_j), p \cup \{v_i, v_j\} \in dom(v_k), p \cup \{v_j, v_k\} \in dom(v_l)$, where $i < j$, $l < j$, $j < k$, there exists an I-equivalent DAG with the edges $\{v_i \rightarrow v_j, v_i \rightarrow v_k, v_j \rightarrow v_k, v_j \rightarrow v_l, v_k \rightarrow v_l\}$ and an I-equivalent DAG with the edges $\{v_l \rightarrow v_j, v_l \rightarrow v_k, v_j \rightarrow v_k, v_j \rightarrow v_i, v_k \rightarrow v_i\}$. We keep only the lexicographically least: the triple of assignments would be permitted iff $i < l$.

In our empirical evaluation, these symmetry-breaking rules were used only as a satisfiability check at each node in the search tree, as we found that propagating the I-equivalence symmetry-breaking rules did not further improve the runtime.

3.4 Dominance constraints (I)

Given an ordering prefix $o_1, \ldots, o_{i-1}$, a domain value p for a vertex variable v_j is *consistent* with the ordering if each element of p occurs in a parent set of a vertex variable in the ordering. The domain value p assigned to vertex variable v_j that occurs at position i in an ordering should be the lowest cost p consistent with the ordering; assigning a domain value with a higher cost can be seen to be dominated as the values can be substituted with no effect on the other variables.

Theorem 3. *Given an ordering prefix $o_1, \ldots, o_{i-1}$, a vertex variable v_j, and domain elements $p, p' \in dom(v_j), p \neq p'$, if p is consistent with the ordering and $cost(p) \leq cost(p')$, p' can be safely pruned from the domain of v_j.*

Example 9. In Example 2, consider the prefix ordering $(o_1, o_2) = (C, D)$. The values $\{C\}$, $\{E\}$, and $\{\}$ can be pruned from each of the domains of v_A and v_B, and the values $\{A\}$, $\{A, B\}$, $\{C\}$, and $\{\}$ can be pruned from the domain of v_E.

3.5 Dominance constraints (II)

Teyssier and Koller [23] present a pruning rule that is now routinely used in score-and-search approaches as a preprocessing step before search begins.

Theorem 4 (Teyssier and Koller [23]). *Given a vertex variable v_j, and domain elements $p, p' \in dom(v_j)$, if $p \subset p'$ and $cost(p) \leq cost(p')$, p' can be safely pruned from the domain of v_j.*

Example 10. In Example 2, the value $\{A, B\}$ can be pruned from the domain of v_E.

We generalize the pruning rule so that it is now applicable during the search. Suppose that some of the vertex variables have been assigned values. These assignments induce ordering constraints on the variables.

Example 11. In Example 2, suppose $v_A = \{D\}$ and $v_C = \{A, B\}$. These assignments induce the ordering constraints $D < A$, $A < C$, and $B < C$.

Definition 4. *Given a set of ordering constraints induced by assignments to the vertex variables, let* $\mathrm{ip}(p)$ *denote the* induced parent set *where p has been augmented with any and all variables that come before in the ordering; i.e., if $y \in p$ and $x < y$ then x is added to p.*

The generalized pruning rule is as follows.

Theorem 5. *Given a vertex variable v_j, and domain elements $p, p' \in dom(v_j), p \neq p'$, if $p \subseteq \mathrm{ip}(p')$ and $cost(p) \leq cost(p')$, p' can be safely pruned from the domain of v_j.*

Example 12. Continuing with Example 11, consider v_E with $p = \{D\}$ and $p' = \{A\}$. The induced parent set $\mathrm{ip}(p')$ is given by $\{A, D\}$ and $cost(p) \leq cost(p')$. Thus, p' can be pruned. Similarly, $p' = \{C\}$ can be pruned.

3.6 Dominance constraints (III)

Consider an ordering prefix $o_1, \ldots, o_i$ with associated vertex and depth variables and let π be a permutation over $\{1, \ldots, i\}$. The cost of *completing* the partial solutions represented by the prefix ordering $o_1, \ldots, o_i$ and the permuted prefix ordering $o_{\pi(1)}, \ldots, o_{\pi(i)}$ are identical. This insight is used by Silander and Myllymäki [5] in their dynamic programming approach and by Fan et al. [9–11] in their best-first approaches based on searching for the shortest path in the ordering graph. However, all of these approaches are extremely memory intensive. Here, we use this insight to prune the search space.

Theorem 6. *Let $cost(o_1, \ldots, o_i)$ be the cost of a partial solution represented by the given ordering prefix. Any ordering prefix $o_1, \ldots, o_i$ can be safely pruned if there exists a permutation π such that $cost(o_{\pi(1)}, \ldots, o_{\pi(i)}) < cost(o_1, \ldots, o_i)$.*

Example 13. In Example 2, consider the ordering prefix $O = (o_1, o_2) = (E, A)$ with associated vertex variables $(v_E, v_A) = (\{\}, \{E\})$ and cost of 27.0. The ordering prefix O can be safely pruned as there exists a permutation $(o_{\pi(1)}, o_{\pi(2)}) = (A, E)$ with associated vertex variables $(v_A, v_E) = (\{\}, \{A\})$ that has a lower cost of 19.6.

Clearly, in its full generality, the condition of Theorem 6 is too expensive to determine exactly as it amounts to solving the original problem. However, we identify three strategies that are easy to determine and collectively were found to be very effective at pruning in our experiments while maintaining optimality.

Strategy 1. We consider permutations that differ from the original permutation only in the last l or fewer places ($l = 4$ in our experiments).

Strategy 2. We consider permutations that differ from the original permutation only in the swapping of the last variable o_i with a variable earlier in the ordering.

Strategy 3. We consider whether a permutation $o_{\pi(1)}, \ldots, o_{\pi(i)}$ of lower cost was explored earlier in the search. To be able to efficiently determine this, we use memoization for ordering prefixes and only continue with a prefix if it has a better cost than one already explored (see, e.g., [26, 27]). Our implementation of memoization uses hashing with quadratic probing and the replacement policy is to keep the most recent if the table becomes too full. It is well known that there is a strong relationship between backtracking search with memoization/caching and dynamic programming using a bottom-up approach, but memoization allows trading space for time and top-down backtracking search allows pruning the search space.

3.7 Acyclic constraint

In this section we describe a propagator for the acyclicity constraint that achieves generalized arc consistency in polynomial time. We first present and analyze an algorithm that checks satisfiability for given possible parent sets. We then explain how this algorithm can be used to achieve generalized arc consistency.

Algorithm 1 can check whether a collection of possible parent sets allows a feasible parent set assignment, i.e. an assignment that represents a DAG. Its correctness is based on the following well-known property of directed acyclic graphs that is also used in the ILP approaches [7, 8].

Theorem 7. *Let G be a directed graph over vertices V and let $parents(v)$ be the parents of vertex v in the graph. G is acyclic if and only if for every non-empty subset $S \subset V$ there is at least one vertex $v \in S$ with $parents(v) \cap S = \{\}$.*

The algorithm works as follows. First, it searches possible sources for the DAG, i.e. vertices for which $\{\}$ is a possible parent set. These vertices are stored in W^0. Note that if a directed graph does not have a source, it must contain a cycle by Theorem 7. Thus, if W^0 remains empty, there is no parent set assignment satisfying the acyclicity constraint. In the next iteration, the algorithm searches for vertices that have a possible parent set consisting of possible sources only. These vertices form set W^1. Again, if there are no such vertices, then no vertex in $V \setminus W^0$ has a possible parent set completely outside $V \setminus W^0$, which violates the acyclicity characterization of Theorem 7. Thus, there is no consistent parent set assignment. We continue this process until all vertices are included in one of the W^k sets or until we find a contradicting set $V \setminus (\bigcup_{i=0}^{k} W^i)$ for some k.

Theorem 8. *We can test satisfiability of the acyclic constraint in time $O(n^2 d)$, where n is the number of vertices and d is an upper bound on the number of possible parent sets per vertex.*

Algorithm 1: Checking satisfiability of acyclic constraint

```
Input: $V = \{v_1, \ldots, v_n\}$, set $dom(v_i)$ of possible parent sets for each vertex $v_i$ in $V$.
Output: True if there is a feasible parent set assignment and false otherwise. Additionally,
        the variables $\mathcal{S}_i$ represent a feasible assignment if one exists.
$k \leftarrow 0$;
$\mathcal{S}_i \leftarrow nil$  for all $v_i \in V$;
while $\bigcup_{j=0}^{k-1} W^j \neq V$ do
    $W^k \leftarrow \{\}$;
    for all vertices $v_i$ not in $\bigcup_{j=0}^{k-1} W^j$ do
        if $v_i$ has a possible parent set $p \in dom(v_i)$ with $p \subseteq \bigcup_{j=0}^{k-1} W^j$ then
            $\mathcal{S}_i \leftarrow p$;
            $W^k \leftarrow W^k \cup \{v_i\}$;
        end if
    end
    if $W^k = \{\}$ then return false;
    $k \leftarrow k + 1$;
end while
return true;
```

Example 14. Let v_A, v_B, v_C, and v_D be vertex variables with the possible parent sets,

$$\begin{aligned} &dom(v_A) = \{\{B\}, \{D\}\}, && dom(v_C) = \{\{B\}, \{D\}\}, \\ &dom(v_B) = \{\{A\}, \{C\}\}, && dom(v_D) = \{\{A\}, \{C\}\}. \end{aligned}$$

Algorithm 1 returns *false* as W^0 is found to be empty.

The algorithm for checking satisfiability can be used to achieve generalized arc consistency by iteratively testing, for each vertex v_i, whether each $p \in dom(v_i)$ has a support. We simply substitute the set of possible parent sets $dom(v_i)$ for v_i by the set $\{p\}$. A satisfiability check on the resulting instance successfully tests whether there is a consistent parent set assignment containing $v_i = p$. If we find a parent set p that cannot appear in any feasible solution, we remove p from the corresponding domain. Note that we only prune a value p from a domain $dom(v_i)$ if $v_i = p$ cannot be part of any feasible solution. This means that $v_i = p$ can also not be part of the support of any other variable value $q \in dom(v_j)$. Therefore, the removal of p cannot cause another supported value to become unsupported. Hence, we do not have to rerun any of the tests; we can simply run the test once for every value in every domain. These considerations show that we can enforce arc consistency for the acyclicity constraint in $O(n^3 d^2)$ steps.

In our empirical evaluation, we found that achieving generalized arc consistency did not pay off in terms of reduced runtime. Hence, in the current set of experiments Algorithm 1 was used only as a satisfiability check at each node in the search tree. Instead, a limited form of constraint propagation was performed based on necessary edges between vertex variables. An edge $v_i \to v_j$ is necessary if v_i occurs in every currently valid parent set for variable v_j; i.e., $\forall p \in dom(v_j) \bullet v_i \in p$. If a directed edge $v_i \to v_j$ is a necessary edge, the directed edge $v_j \to v_i$ cannot be an edge in a valid DAG, as a cycle would be introduced. Thus, any parent set that contains v_j can be removed from the domain of v_i. Removing domain elements may introduce additional necessary edges and pruning can be based on chaining necessary edges.

Example 15. Let v_A, v_B, v_C, and v_D be vertex variables with the possible parent sets,

$$\begin{array}{ll} dom(v_A) = \{\{\}, \{B\}, \{C\}\} & dom(v_C) = \{\{B\}\} \\ dom(v_B) = \{\{A\}, \{A, C\}\} & dom(v_D) = \{\{A\}, \{A, C\}\} \end{array}$$

Since the edge $B \to C$ is necessary, the value $\{A, C\}$ can be pruned from the domain of v_B. This causes the edge $A \to B$ to become necessary, and the values $\{B\}$ and $\{C\}$ can be pruned from the domain of v_A.

We conclude with the following observation. Let i_v be the index of the set W^i in which we include vertex v in the satisfiability algorithm. Then, i_v is a lower bound on the number that vertex v can have in any topological numbering. This lower bound can be used in propagating Constraint 4.

3.8 Solving the constraint model

A constraint-based depth-first branch-and-bound search is used to solve the constraint model; i.e., the nodes in the search tree are expanded in a depth-first manner and a node is expanded only if the propagation of the constraints succeeds and a lower bound estimate on completing the partial solution does not exceed the current upper bound.

The branching is over the ordering (permutation) variables and uses the static order $o_1, \ldots, o_n$. Once an ordering variable is instantiated as $o_i = j$, the associated vertex variable v_j and depth variable d_i are also instantiated.

The lower bound is based on the lower bound proposed by Fan and Yuan [11]. In brief, prior to search, the strongly connected components (SCCs) of the graph based on the top few lowest cost elements in the domains of the vertex variables are found and pattern databases are constructed based on the SCCs. The pattern databases allow a fast and often accurate lower bound estimate during the search (see [11] for details).

The initial upper bound, found before search begins, is based on the local search algorithm proposed by Teyssier and Koller [23]. The algorithm uses restarts and first-improvement moves, the search space consists of all permutations, and the neighborhood function consists of swapping the order of two variables. Of course, as better solutions are found during the search the upper bound is updated.

As a final detail, additional pruning on the vertex variables can be performed based on the (well-known) approximation of bounds consistency on a cost function that is a knapsack constraint: $z = cost(v_1) + \cdots + cost(v_n)$. Let the bounds on $cost(v_i)$ be $[l_i, u_i]$, and let lb and ub be the current lower bound and upper bound on the cost, respectively. At any point in the search we have the constraint $lb \leq z < ub$ and a value $p \in dom(v_i)$ can be pruned if $cost(p) + \sum_{j \neq i} u_j < lb$ or if $cost(p) + \sum_{j \neq i} l_j \geq ub$. Note that the expression $cost(p) + \sum_{j \neq i} l_j$ can be replaced with any lower bound on the cost of a solution that includes p and respects the current domains and instantiations, as long as the lower bound never over estimates. Fortunately, we have a fast and effective method of querying such lower bounds and we use it when performing this pruning.

4 Experimental Evaluation

In this section, we compare a bespoke C++ implementation of our constraint-based approach, called CPBayes [2], to the current state-of-the-art on benchmark instances and show that our approach compares favorably both in terms of number of instances solved within specified resource bounds and in terms of solution time.

The set of benchmark instances are derived from data sets obtained from the UCI Machine Learning Repository [3] and data generated from networks obtained from the Bayesian Network Repository [4]. Following previous work, the local score for each possible parent set and each random variable was computed in a preprocessing step (either by us or by others) prior to the search for the best network structure and we do not report the preprocessing time. Note that the computations of the possible parent sets for each variable are independent and can be determined in parallel. The BIC/MDL [18, 19] and BDeu [20, 21] scoring methods were used.

Table 2 shows the results of comparing CPBayes (v1.0) against Barlett and Cussens' [8] GOBNILP system (v1.4.1) based on integer linear programming, and Fan, Malone, and Yuan's [10, 15, 11] system (v2015) based on A* search. These two systems represent the current state-of-the-art for global (exact) approaches. Breadth-first BnB search [10, 15, 11] is also competitive but its effectiveness is known to be very similar to that of A*. Although for space reasons we do not report detailed results, we note that on these benchmarks CPBayes far outpaces the previous best depth-first branch-and-bound search approach [13]. GOBNILP (v1.4.1) [5] and A* (v2015) [6] are both primarily written in C/C++. A* (v2015) is the code developed by Fan et al. [10, 15], but in the experiments we included our implementation of the improved lower bounds recently proposed by Fan and Yuan [11]. Thus, both CPBayes (v1.0) and A* (v2015) use exactly the same lower bounding technique (see Section 3.8). The experiments were performed on a cluster, where each node of the cluster is equipped with four AMD Opteron CPUs at 2.4 GHz and 32.0 GB memory. Resource limits of 24 hours of CPU time and 16 GB of memory were imposed both for the preprocessing step common to all methods of obtaining the local scores and again to determine the minimal cost BN using a method. The systems were run with their default values.

5 Discussion and Future Work

The Bayesian Network Repository classifies networks as small (< 20 random variables), medium (20–60 random variables), large (60–100 random variables), very large (100–1000 random variables), and massive (> 1000 random variables). The benchmarks shown in Table 2 fall into the small and medium classes. We are not aware of any reports of results for exact solvers for instances beyond the medium class (Barlett and

[2] CPBayes code available at: `cs.uwaterloo.ca/~vanbeek/research`
[3] `archive.ics.uci.edu/ml/`
[4] `www.bnlearn.com/bnrepository/`
[5] `www.cs.york.ac.uk/aig/sw/gobnilp/`
[6] `bitbucket.org/bmmalone/`

Table 2. For each benchmark, time (seconds) to determine minimal cost BN using various systems (see text), where n is the number of random variables in the data set, N is the number of instances in the data set, and d is the total number of possible parents sets for the random variables. Resource limits of 24 hours of CPU time and 16 GB of memory were imposed: OM = out of memory; OT = out of time. A blank entry indicates that the preprocessing step of obtaining the local scores for each random variable could not be completed within the resource limits.

			BDeu				BIC			
				GOBN.	A*	CPBayes		GOBN.	A*	CPBayes
Benchmark	n	N	d	v1.4.1	v2015	v1.0	d	v1.4.1	v2015	v1.0
shuttle	10	58,000	812	58.5	0.0	0.0	264	2.8	0.1	0.0
adult	15	32,561	768	1.4	0.1	0.0	547	0.7	0.1	0.0
letter	17	20,000	18,841	5,060.8	1.3	1.4	4,443	72.5	0.6	0.2
voting	17	435	1,940	16.8	0.3	0.1	1,848	11.6	0.4	0.1
zoo	17	101	2,855	177.7	0.5	0.2	554	0.9	0.4	0.1
tumour	18	339	274	1.5	0.9	0.2	219	0.4	0.9	0.2
lympho	19	148	345	1.7	2.1	0.5	143	0.5	1.0	0.2
vehicle	19	846	3,121	90.4	2.4	0.7	763	4.4	2.1	0.5
hepatitis	20	155	501	2.1	4.9	1.1	266	1.7	4.8	1.0
segment	20	2,310	6,491	2,486.5	3.3	1.3	1,053	13.2	2.4	0.5
mushroom	23	8,124	438,185	OT	255.5	561.8	13,025	82,736.2	34.4	7.7
autos	26	159	25,238	OT	918.3	464.2	2,391	108.0	316.3	50.8
insurance	27	1,000	792	2.8	583.9	107.0	506	2.1	824.3	103.7
horse colic	28	300	490	2.7	15.0	3.4	490	3.2	6.8	1.2
steel	28	1,941	113,118	OT	902.9	21,547.0	93,026	OT	550.8	4,447.6
flag	29	194	1,324	28.0	49.4	39.9	741	7.7	12.1	2.6
wdbc	31	569	13,473	2,055.6	OM	11,031.6	14,613	1,773.7	1,330.8	1,460.5
water	32	1,000					159	0.3	1.6	0.6
mildew	35	1,000	166	0.3	7.6	1.5	126	0.2	3.6	0.6
soybean	36	266					5,926	789.5	1,114.1	147.8
alarm	37	1,000					672	1.8	43.2	8.4
bands	39	277					892	15.2	4.5	2.0
spectf	45	267					610	8.4	401.7	11.2
sponge	45	76					618	4.1	793.5	13.2
barley	48	1,000					244	0.4	1.5	3.4
hailfinder	56	100					167	0.1	9.9	1.5
hailfinder	56	500					418	0.5	OM	9.3
lung cancer	57	32					292	2.0	OM	10.5
carpo	60	100					423	1.6	OM	253.6
carpo	60	500					847	6.9	OM	OT

Cussens [8] report results for GOBNILP for $n > 60$, but they are solving a different problem, severely restricting the cardinality of the parent sets to ≤ 2).

Benchmarks from the small class are easy for the CPBayes and A* methods, but can be somewhat challenging for GOBNILP depending on the value of the parameter d, the total number of parent sets for the random variables. Along with the integer linear programming (ILP) solver GOBNILP, CPBayes scales fairly robustly to medium instances using a reasonable restriction on memory usage (both use only a few GB of memory, far under the 16 GB limit used in the experiments; in fairness, the scalability of the A* approach on a very large memory machine is still somewhat of an open question). CPBayes also has several other advantages, which it shares with the ILP approach, over A*, DP, and BFBnB approaches. Firstly, the constraint model is a purely declarative representation and the same model can be given to an exact solver or a solver based on local search, such as large neighborhood search. Secondly, the constraint model can be augmented with side structural constraints that can be important in real-world modeling (see [28]). Finally, the solver is an anytime algorithm since, as time progresses, the solver progressively finds better solutions.

Let us now turn to a comparison between GOBNILP and CPBayes. CPBayes scales better than GOBNILP along the dimension d which measures the size of the possible parent sets. A partial reason is that GOBNILP uses a constraint model that includes a (0,1)-variable for each possible parent set. GOBNILP scales better than CPBayes along the dimension n which measures the number of random variables. CPBayes has difficulty at the topmost range of n proving optimality. There is some evidence that $n = 60$ is near the top of the range for GOBNILP as well. Results reported by Barlett and Cussens [8] for the carpo benchmark using larger values of N and the BDeu scoring method—the scoring method which usually leads to harder optimization instances than BIC/MDL—showed that instances could only be solved by severely restricting the cardinality of the parent sets. A clear difficulty in scaling up all of these score-and-search methods is in obtaining the local scores within reasonable resource limits.

In future work on Bayesian network structure learning, we intend to focus on improving the robustness and scalability of our CPBayes approach. A direction that appears especially promising is to improve the branch-and-bound search by exploiting decomposition and lower bound caching during the search [29, 30]. As well, our approach, as with all current exact approaches, assumes complete data. An important next step is to extend our approach to handle missing values and latent variables (cf. [31]).

Acknowledgements

This research was partially funded through an NSERC Discovery Grant. We thank Claude-Guy Quimper, Alejandro López-Ortiz, Mats Carlsson, and Christian Schulte for helpful discussions, and Brandon Malone and James Cussens for providing test instances and their code.

References

1. Witten, I.H., Frank, E., Hall, M.A.: Data Mining. 3rd edn. Morgan Kaufmann (2011)
2. Chickering, D., Meek, C., Heckerman, D.: Large-sample learning of Bayesian networks is NP-hard. In: Proc. of UAI. (2003) 124–133
3. Koivisto, K.: Parent assignment is hard for the MDL, AIC, and NML costs. In: Proc. of COLT. (2006) 289–303
4. Koivisto, M., Sood, K.: Exact Bayesian structure discovery in Bayesian networks. J. Mach. Learn. Res. **5** (2004) 549–573
5. Silander, T., Myllymäki, P.: A simple approach for finding the globally optimal Bayesian network structure. In: Proc. of UAI. (2006) 445–452

6. Malone, B., Yuan, C., Hansen, E.A.: Memory-efficient dynamic programming for learning optimal Bayesian networks. In: Proc. of AAAI. (2011) 1057–1062
7. Jaakkola, T., Sontag, D., Globerson, A., Meila, M.: Learning Bayesian network structure using LP relaxations. In: Proc. of AISTATS. (2010) 358–365
8. Barlett, M., Cussens, J.: Advances in Bayesian network learning using integer programming. In: Proc. of UAI. (2013) 182–191
9. Yuan, C., Malone, B.: Learning optimal Bayesian networks: A shortest path perspective. J. of Artificial Intelligence Research **48** (2013) 23–65
10. Fan, X., Malone, B., Yuan, C.: Finding optimal Bayesian network structures with constraints learned from data. In: Proc. of UAI. (2014) 200–209
11. Fan, X., Yuan, C.: An improved lower bound for Bayesian network structure learning. In: Proc. of AAAI. (2015)
12. Tian, J.: A branch-and-bound algorithm for MDL learning Bayesian networks. In: Proc. of UAI. (2000) 580–588
13. Malone, B., Yuan, C.: A depth-first branch and bound algorithm for learning optimal Bayesian networks. In: Graph Structures for Knowledge Representation and Reasoning. Volume 8323 of Lecture Notes in Computer Science. Springer (2014) 111–122
14. de Campos, C.P., Ji, Q.: Efficient structure learning of Bayesian networks using constraints. Journal of Machine Learning Research **12** (2011) 663–689
15. Fan, X., Yuan, C., Malone, B.: Tightening bounds for Bayesian network structure learning. In: Proc. of AAAI. (2014) 2439–2445
16. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
17. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. The MIT Press (2009)
18. Schwarz, G.: Estimating the dimension of a model. Ann. Stat. **6** (1978) 461–464
19. Lam, W., Bacchus, F.: Using new data to refine a Bayesian network. In: Proc. of UAI. (1994) 383–390
20. Buntine, W.L.: Theory refinement of Bayesian networks. In: Proc. of UAI. (1991) 52–60
21. Heckerman, D., Geiger, D., Chickering, D.M.: Learning Bayesian networks: The combination of knowledge and statistical data. Machine Learning **20** (1995) 197–243
22. Larranaga, P., Kuijpers, C., Murga, R., Yurramendi, Y.: Learning Bayesian network structures by searching for the best ordering with genetic algorithms. IEEE Trans. Syst., Man, Cybern. **26** (1996) 487–493
23. Teyssier, M., Koller, D.: Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In: Proc. of UAI. (2005) 548–549
24. Chickering, D.M.: A transformational characterization of equivalent Bayesian network structures. In: Proc. of UAI. (1995) 87–98
25. Chickering, D.M.: Learning equivalence classes of Bayesian network structures. Journal of Machine Learning Research **2** (2002) 445–498
26. Michie, D.: "memo" functions and machine learning. Nature **218** (1968) 19–22
27. Smith, B.M.: Caching search states in permutation problems. In: Proc. of CP. (2005) 637–651
28. Cussens, J.: Integer programming for Bayesian network structure learning. Quality Technology & Quantitative Management **1** (2014) 99–110
29. Kitching, M., Bacchus, F.: Symmetric component caching. In: Proc. of IJCAI. (2007) 118–124
30. Kitching, M., Bacchus, F.: Exploiting decomposition in constraint optimization problems. In: Proc. of CP. (2008) 478–492
31. Friedman, N.: Learning belief networks in the presence of missing values and hidden variables. In: Proc. of ICML. (1997) 125–133