Finding Optimal Bayesian Network Structures with Constraints Learned from Data

Xiannian Fan[1], **Brandon Malone**[2] **and Changhe Yuan**[1]
[1]Queens College/City University of New York
[2]Helsinki Institute for Information Technology,
xfan2@gc.cuny.edu, brandon.malone@cs.helsinki.fi, changhe.yuan@qc.cuny.edu

Abstract

Several recent algorithms for learning Bayesian network structures first calculate potentially optimal parent sets (POPS) for all variables and then use various optimization techniques to find a set of POPS, one for each variable, that constitutes an optimal network structure. This paper makes the observation that there is useful information implicit in the POPS. Specifically, the POPS of a variable constrain its parent candidates. Moreover, the parent candidates of all variables together give a directed cyclic graph, which often decomposes into a set of strongly connected components (SCCs). Each SCC corresponds to a smaller subproblem which can be solved independently of the others. Our results show that solving the constrained subproblems significantly improves the efficiency and scalability of heuristic search-based structure learning algorithms. Further, we show that by considering only the top p POPS of each variable, we quickly find provably very high quality networks for large datasets.

1 INTRODUCTION

Bayesian networks (BNs) are graphical models that represent uncertain relationships between random variables. While BNs have become one of the most popular and well-studied probabilistic model classes, a common bottleneck lies in deciding upon their structure. Often, experts are unable to completely specify the structure; in these cases, a good structure must be learned from expert knowledge and available data. In this work, we consider the problem of exact, score-based Bayesian network structure learning (BNSL), which is known to be NP-hard (Chickering 1996).

Despite the difficulty of BNSL, though, a variety of algorithms have been proposed which can solve modest-sized learning problems. The first exact algorithms were based on dynamic programming (Koivisto and Sood 2004; Ott, Imoto, and Miyano 2004; Singh and Moore 2005; Silander and Myllymäki 2006). Later algorithms have used strategies such as integer linear programming (Jaakkola et al. 2010; Cussens 2011; Bartlett and Cussens 2013) and heuristic search (Yuan and Malone 2013; Malone et al. 2011; Malone and Yuan 2013). These algorithms generally take as input the potentially optimal parent sets (POPS) for each variable. They all improve upon dynamic programming by, either implicitly or explicity, pruning the search space and considering only promising structures.

In this paper, we focus on the heuristic search approach first proposed by Yuan *et al.* (2011) in which BNSL is formulated as a shortest-path finding problem. A state space search strategy like A* or breadth-first branch and bound is then used to solve the transformed problem. Previous work in heuristic search for BNSL has focused on pruning unpromising structures based on bounds derived from admissible heuristic functions. In this work, we show that *POPS constraints*, which are implicit in the problem input, significantly improve the efficiency of the search by pruning large portions of the search space.

The remainder of this paper is structured as follows. Section 2 provides an overview of BNSL and the shortest-path finding formulation of the problem. Section 3 introduces POPS constraints and shows how they can be used to prune the search space. Additionally, we describe a pruning strategy which uses the constraints to trade guaranteed bounded optimality for more scalable performance in Section 4. The POPS constraints also reduce the space required by the heuristics used for pruning during search, as described in Section 5. In Section 6, we compare POPS constraints to related work. Section 7 gives empirical results on a set of benchmark datasets, and Section 8 concludes the paper.

2 BACKGROUND

This section reviews BNSL and the shortest-path finding formulation of the learning problem (Yuan and Malone 2013), which is the basis of our new algorithm.

2.1 BAYESIAN NETWORK STRUCTURE LEARNING

A Bayesian network (BN) consists of a directed acyclic graph (DAG) in which the vertices correspond to a set of random variables $\mathbf{V} = \{X_1, ..., X_n\}$ and a set of conditional probability distributions $P(X_i|PA_i)$, where all parents of X_i are referred to as PA_i. The joint probability over all variables factorizes as the product of the conditional probability distributions.

We consider the problem of learning a network structure from a discrete dataset $\mathbf{D} = \{D_1, ..., D_N\}$, where D_i is an instantiation of all the variables in $\mathbf{V}$. A scoring function s measures the goodness of fit of a network structure to $\mathbf{D}$ (Heckerman 1998). The goal is to find a structure which optimizes the score. We only require that the scoring function is *decomposable* (Heckerman 1998); that is, the score of a network $s(N) = \sum_i s_i(PA_i)$. The $s_i(PA_i)$ values are often called *local scores*. Many commonly used scoring functions, such as MDL (Lam and Bacchus 1994) and BDe (Buntine 1991; Heckerman, Geiger, and Chickering 1995), are decomposable.

2.2 LOCAL SCORES

While the local scores are defined for all 2^{n-1} possible parent sets for each variable, this number is greatly reduced by pruning parent sets that are provably never optimal (de Campos and Ji 2011). We refer to this as *lossless score pruning* because it is guaranteed to not remove the optimal network from consideration. We refer to the scores remaining after pruning as *potentially optimal parent sets* (POPS).

Other pruning strategies, such as restricting the cardinality of parent sets, are also possible, but these techniques could eliminate parent sets which are in the globally optimal network; we refer to pruning strategies which might remove the optimal network from consideration as *lossy score pruning*. Of course, these, and any other, score pruning strategies can be combined.

Regardless of the score pruning strategies used, we still refer to the set of unpruned local scores as POPS and denote the set of POPS for X_i as $\mathcal{P}_i$. The POPS are given as input to the learning problem. We define the *Bayesian network structure learning* problem (BNSL) as follows.

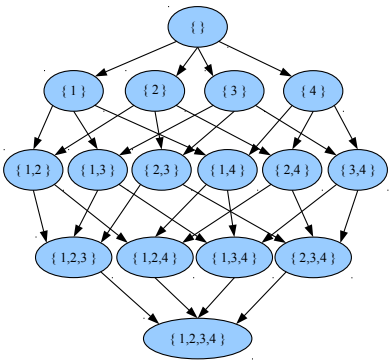

Figure 1: An order graph for four variables.

The BNSL Problem

INPUT: A set $\mathbf{V} = \{X_1, \ldots, X_n\}$ of variables and a set of POPS $\mathcal{P}_i$ for each X_i.

TASK: Find a DAG N^* such that

$$N^* \in \arg\min_N \sum_{i=1}^{n} s_i(PA_i),$$

where PA_i is the parent set of X_i in N and $PA_i \in \mathcal{P}_i$.

2.3 SHORTEST-PATH FINDING FORMULATION

Yuan and Malone (2013) formulated BNSL as a shortest-path finding problem. 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 $\mathbf{U} \cup \{X_i\}$ in the graph represents generating a successor node by adding a new variable X_i as a leaf to an existing subnetwork of variables $\mathbf{U}$; the cost of the arc is equal to the score of the optimal parent set for X_i out of $\mathbf{U}$, which is computed by considering all subsets of the variables in $PA \subseteq \mathbf{U}, PA \in \mathcal{P}_i$, i.e.,

$$cost(\mathbf{U} \to \mathbf{U} \cup \{X_i\}) = BestScore(X_i, \mathbf{U}) \quad (1)$$

$$= \min_{PA_i \subseteq \mathbf{U}, PA_i \in \mathcal{P}_i} s_i(PA_i). \quad (2)$$

In this search graph, each path from *start* to *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.

2.4 HEURISTIC SEARCH ALGORITHMS

This shortest path problem has been solved using several heuristic search algorithms, including A* (Yuan, Malone, and Wu 2011), anytime window A* (AWA*) (Malone and Yuan 2013) and breadth-first branch and bound (BFBnB) (Malone et al. 2011).

In A* (Hart, Nilsson, and Raphael 1968), an admissible heuristic function is used to calculate a lower bound on the cost from a node $\mathbf{U}$ in the order graph to $goal$. An f-cost is calculated for $\mathbf{U}$ by summing the cost from $start$ to $\mathbf{U}$ (called $g(\mathbf{U})$) and the lower bound from $\mathbf{U}$ to $goal$ (called $h(\mathbf{U})$). For BNSL, $g(\mathbf{U})$ corresponds to the score of the subnetwork over the variables $\mathbf{U}$, and $h(\mathbf{U})$ estimates the score of the remaining variables. So $f(\mathbf{U}) = g(\mathbf{U})+h(\mathbf{U})$. The f-cost provides an optimistic estimation on how good a path through $\mathbf{U}$ can be. The search maintains a list of nodes to be expanded sorted by f-costs called $open$ and a list of already-expanded nodes called $closed$. Initially, $open$ contains just $start$, and $closed$ is empty. Nodes are then expanded from $open$ in best-first order according to f-costs. Expanded nodes are added to $closed$. As better paths to nodes are discovered, they are added to $open$. Upon expanding $goal$, the shortest path from $start$ to $goal$ has been found.

In AWA* (Aine, Chakrabarti, and Kumar 2007), a sliding window search strategy is used 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. For the order graph, the layer of a node corresponds to the number of variables in its subnetwork. 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 $goal$ is found in each iteration, which gives an upper bound solution. After finding the path to $goal$, the window size is increased by 1 and the frozen nodes become $open$. The iterative process continues until no nodes are frozen during an iteration, which means the upper bound solution is optimal. Alternatively, the search can be stopped early if a resource bound, such as running time, is exceeded; the best solution found so far is output.

In BFBnB (Zhou and Hansen 2006), nodes are expanded one layer at a time. Before beginning the BFBnB search, a quick search strategy, such as AWA* for a few iterations or greedy hill climbing, is used to find a "good" network and its score. The score is used as an upper bound. During the BFBnB search, any node with an f-cost greater than the upper bound can safely be pruned.

Yuan *et al.* (2011) gave a simple heuristic function. Later, tighter heuristics based on pattern databases were developed (Yuan and Malone 2012). All of the heuristics were shown to be *admissible*, i.e., to always give a lower bound on the cost from $\mathbf{U}$ to $goal$. Furthermore, the heuristics have been shown to be *consistent*, which is a property similar to non-negativity required by Dijkstra's algorithm. Consistent heuristics always underestimate the cost of the path between *any* two nodes (Edelkamp and Schrodl 2012). Primarily, in A*, consistency ensures that the first time a node is expanded, the shortest path to that node has been found, so no node ever needs to be re-expanded.

3 LEARNING UNDER POPS CONSTRAINTS

The main contribtion of our current work focuses on taking advantage of the implicit information encoded in the POPS. We will first motivate our approach using a simple example and then describe the technical details.

3.1 A SIMPLE EXAMPLE

Table 1 shows the POPS for six variables. Based on these sets, we can see that not all variables can select all other variables as parents. For example, X_1 can only select X_2 as its parent (due to score pruning). We collect all of the potential parents for X_i by taking the union of all $PA \in \mathcal{P}_i$. Figure 2 shows the resulting *parent relation graph* for the POPS in Table 1. The parent relation graph includes an edge from $X_j$to X_i if X_j is a potential parent of X_i.

Naively, the complete order graph for six variables contains 2^6 nodes. However, from the parent relation graph, we see that none of $\{X_3, X_4, X_5, X_6\}$ can be a parent of X_1 or X_2. Consequently, we can split the problem into two subproblems as shown in Figure 3: first, finding the shortest path from $start$ to $\{X_1, X_2\}$, and then, finding the shortest path from $\{X_1, X_2\}$ to $goal$. Thus, the size of the search space is reduced to $2^2 + 2^4$.

3.2 ANCESTOR RELATIONS

This simple example shows that the parent relation graph can be used to prune the order graph without bounds. In general, we must consider *ancestor relations* to prune the order graph. In particular, if X_i can be an ancestor of X_j, and X_j cannot be an ancestor of X_i (due to local score pruning), then no node in the order graph which contains X_j but not X_i needs to be generated.

As a proof sketch, we can consider a node $\mathbf{U}$ which includes neither X_i nor X_j. If we add X_i and then X_j, then the cost from $\mathbf{U}$ to $\mathbf{U} \cup \{X_i, X_j\}$ is $BestScore(X_i, \mathbf{U}) + BestScore(X_j, \mathbf{U} \cup \{X_i\})$. On the other hand, if we add X_j first, then the cost from $\mathbf{U}$ to $\mathbf{U} \cup \{X_i, X_j\}$ is $BestScore(X_j, \mathbf{U}) + BestScore(X_i, \mathbf{U} \cup \{X_j\})$. However, due to the ancestor relations, we know that $BestScore(X_i, \mathbf{U} \cup \{X_j\}) = BestScore(X_i, \mathbf{U})$. So, regardless of the order we add the two variables, X_i will have the same parent choices. If we add X_j first, though, then X_j

variable	POPS					
X_1	$\{X_2\}$	$\{\}$				
X_2	$\{X_1\}$	$\{\}$				
X_3	$\{X_1, X_2\}$	$\{X_2, X_6\}$	$\{X_1, X_6\}$	$\{X_2\}$	$\{X_6\}$	$\{\}$
X_4	$\{X_1, X_3\}$	$\{X_1\}$	$\{X_3\}$	$\{\}$		
X_5	$\{X_4\}$	$\{X_2\}$	$\{\}$			
X_6	$\{X_2, X_5\}$	$\{X_2\}$	$\{\}$			

Table 1: The POPS for six variables. The ith row shows $\mathcal{P}_i$.

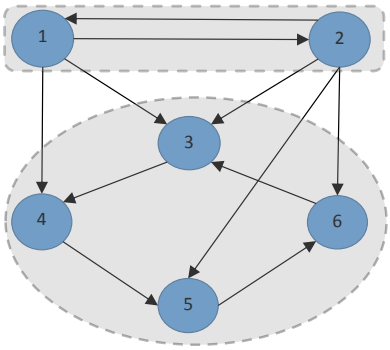

Figure 2: The parent relation graph constructed by aggregating the POPS in Table 1. The strongly connected components are surrounded by shaded shapes.

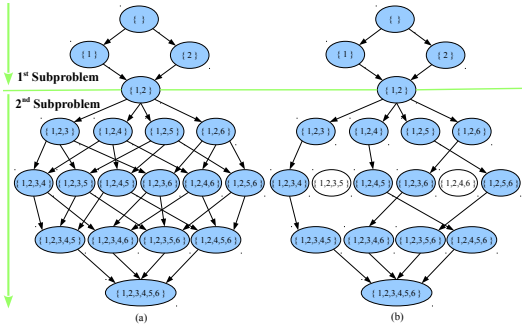

Figure 3: Order graphs after applying the POPS constraints. (a) The order graph after applying the POPS constraints once. (b) The order graph after recursively applying the POPS constraints on the second subproblem.

will have fewer choices. Therefore, adding X_j as a leaf first can never be better than adding X_i first (Yuan and Malone 2013).

3.3 POPS CONSTRAINTS PRUNING

We find the ancestor relations by constructing the parent relation graph and extracting its strongly connected components (SCCs). The SCCs of the parent relation graph form the *component graph*, which is a DAG (Cormen et al. 2001); each component graph node c_i corresponds to an SCC scc_i from the parent relation graph (which in turn corresponds to a set of variables in the Bayesian network). The component graph includes a directed edge from c_i to c_j if the parent relation graph includes an edge from a variable $X_i \in scc_i$ to $X_j \in scc_j$.

The component graph gives the ancestor constraints: if c_j is a descendent of c_i in the component graph, then variables in scc_j cannot be ancestors of variables in scc_i. Consequently, the component graph gives *POPS constraints* which allow the order graph to be pruned without considering bounds. In particular, the POPS constraints allow us to prune nodes in the order graph which do not respect the ancestor relations.

Tarjan's algorithm (Tarjan 1972) extracts the SCCs from directed graphs, like the parent relation graph. We chose to use it because, in addition to its polynomial complexity, it extracts the SCCs from the parent relation graph consistent with their topological order in the component graph. Consequently, all of the parent candidates of $X \in scc_i$ appear in $\mathbf{PC}_i = \cup_{k=1}^{i} scc_k$[1]. After extracting the m SCCs, the search can be split into m indepedent subproblems: one for each SCC where $start_i$ is $\mathbf{PC}_{i-1}$ and $goal_i$ is $\mathbf{PC}_i$. That is, during the i^{th} subproblem, we select the optimal parents for the variables in scc_i. Of course, $start_0 = \emptyset$ and $goal_m = \mathbf{V}$. The worst-case complexity of subproblem i is then $O(2^{|scc_i|})$. Figure 3(a) shows the pruned order graph resulting from the parent relation graph in Figure 2. In particular, it shows the first subproblem, from $\mathbf{PC}_0 = \emptyset$ to $\mathbf{PC}_1 = \{X_1, X_2\}$, and the second subproblem, from $\mathbf{PC}_1$ to $\mathbf{PC}_2 = \mathbf{V}$.

The (worst-case) size of the original order graph for n variables is as follows.

$$O(2^{|scc_1|+\ldots+|scc_m|}) = O(2^n) \tag{3}$$

The worst-case size of the search space after splitting into subproblems using the SCCs is as follows.

$$O(2^{|scc_1|} + \ldots + 2^{|scc_m|}) = O(m \cdot \max_{|i|} 2^{|scc_i|}) \tag{4}$$

That is, the complexity is at worst exponential in the size of the largest SCC. Consequently, our method can scale to

[1]Depending on the structure of the component graph, this may be a superset of the parent candidates for X.

datasets with many variables if the largest SCC is of manageable size.

3.4 RECURSIVE POPS CONSTRAINTS PRUNING

As described in Section 3.3, the size of the search space for the i^{th} subproblem is $O(2^{|scc_i|})$, which can still be intractable for large SCCs. However, recursive application of the POPS constraints can further reduce the size of the search space. We refer to the constraints added by this strategy as *recursive POPS constraints*.

The intuition is the same as that behind the POPS constraints. As an example, consider the subproblem associated with scc_2 in Figure 3, which includes variables X_3, X_4, X_5 and X_6. Naively, the order graph associated with this subproblem has $O(2^4)$ nodes. However, suppose we add variable X_3 as a leaf first. Then, the remaining variables split into three SCCs, and their order is completely determined. Similarly, selecting any of the other variables as the first to add as a leaf completely determines the order of the rest. Figure 3(b) shows the order graph after applying recursive POPS constraints.

In general, selecting the parents for one of the variables has the effect of removing that variable from the parent relation graph. After removing it, the remaining variables may split into smaller SCCs, and the resulting smaller subproblems can be solved recursively. These SCC checks can be implemented efficiently by again appealing to Tarjan's algorithm. In particular, after adding variable X_i as a leaf from $\mathbf{U}$, we remove all of those variables from the parent relation graph. We then find the topologically first SCC and expand just the variables in that component. As we recursively explore the remaining variables, they will all eventually appear in the first SCC of the updated parent relation graph.

4 TOP-p POPS CONSTRAINT

As shown in Equation 4, the complexity of the search largely depends on the size of the largest strongly connected component. The recursive splitting described in Section 3.4 helps reduce this complexity, but for large, highly connected SCCs, the subproblems may still be too large to solve. For these cases, we can tradeoff between the complexity of the search and a bound on the optimality of the solution. In particular, rather than constructing the parent relation graph by aggregating *all* of the POPS, we can instead create the graph by considering only the best p POPS for each variable. We consider the minimization version of BNSL, so the best POPS are those with the lowest scores. This yields a set of parent candidates for each variable, and only POPS which are subsets of these parent candidates are retained. The empty set is always a subset of the parent candidates, so some DAG (e.g., the DAG with no edges) is always consistent with the resulting pruned set of POPS. We call this score pruning strategy the *top-p POPS constraint*.

By removing some of the POPS in this manner, though, we can no longer guarantee to find the globally optimal Bayesian network. That is, this score pruning strategy is lossy. Despite losing the globally optimal guarantee, though, we can still offer a *bounded suboptimality* guarantee. In particular, suppose we apply the top-p POPS constraint and learn a BN N with score $s(N)$ in which X_i selects parents PA_i with score $s_i(PA_i)$. Additionally, suppose the best pruned parent set PA'_i for X_i has score $s_i(PA'_i)$. Then, the most improvement we could have in the score by including the pruned POPS for X_i is $\delta_i = \max(0, s_i(PA_i) - s_i(PA'_i))$. The $\max$ is necessary when the selected parent set is better than the best excluded parent set. Consequently, a suboptimality bound ϵ on the score of the unconstrained optimal network relative to $s(N)$ is as follows.

$$\epsilon = \frac{s(N)}{\sum_i (s_i(PA_i) - \delta_i)} \tag{5}$$

When ϵ is 1, N is the globally optimal network.

5 REDUCING THE SPACE REQUIREMENTS OF THE HEURISTIC

In this section we show that the POPS constraints can reduce the space requirements of the lower bound heuristc used during search.

A simple heuristic function was introduced for computing lower bounds for the order graph (Yuan and Malone 2013) which allows each remaining variable to choose optimal parents from all the other variables. This completely relaxes the acyclicity constraint on the BN structure. The heuristic was proven to be admissible, meaning it never overestimates the distance to $goal$ (Yuan and Malone 2013). However, because of the complete relaxation of the acyclicity constraint, the simple heuristic may generate loose lower bounds.

5.1 THE k-CYCLE CONFLICT HEURISTIC

In (Yuan and Malone 2012), an improved heuristic function called *k-cycle conflict heuristic* was proposed which reduces 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. Each group (subset of variables) is called a *pattern*. One approach to creating the patterns is to divide the variables $\mathbf{V}$ into l approximately equal-sized static subsets $\mathbf{V}_i$ (typically $l = 2$, so $k = n/2$). For each $\mathbf{V}_i$, a pattern database h_i is created by performing a breadth-first search in a "reverse" order graph in which $start$ is $\mathbf{V}$ and $goal$ is $\mathbf{V}_i$. A node $\mathbf{U}'$ in the graph is expanded by removing each of the variables $X \in \mathbf{V}_i$.

An 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 g cost for node $\mathbf{U}'$ gives the cost of the pattern $\mathbf{V} \setminus \mathbf{U}'$. The patterns from different groups are guaranteed to be mutually exclusive, so the heuristic value of a node $\mathbf{U}$ in the order graph is the sum of the pattern costs for the variables remaining in each partition. That is, $h(\mathbf{U}) = \sum_i^l h_i(\mathbf{V}_i \cap (\mathbf{V} \setminus \mathbf{U}))$. This approach is a *statically-partitioned additive pattern database heuristic* (Felner, Korf, and Hanan 2004) referred to as *static pattern databases*. Static pattern databases were shown to be consistent (Yuan and Malone 2013).

5.2 CREATING PATTERN DATABASES FOR SUBPROBLEMS

As described in Section 3, the search is split into an independent subproblem for each SCC. Furthermore, using Tarjan's algorithm, the SCCs are ordered according to their topological order in the component graph. Consequently, we construct static pattern databases using a similar strategy as before. Namely, each SCC is partitioned into l groups $scc_i = scc_{i1} \dots scc_{il}$ (typically $l = 2$). For each partition, a pattern database h_{ik} is created. For h_{ik}, the pattern costs are calculated using a breadth-first search in a reverse order graph in which $start$ is $\mathbf{PC}_{i-1} \cup scc_{ik}$ and $goal$ is $\mathbf{PC}_{i-1}$. The arc costs in this graph are $BestScore(X, (\bigcup_{j \neq k} scc_{ij}) \cup \mathbf{U})$. Thus, the heuristic value from $\mathbf{U}$ to $\mathbf{PC}_i$, referred to as h_1, is as follows.

$$h_1(\mathbf{U}) = \sum_k^l h_{ik}(scc_{ik} \cap (\mathbf{V} \setminus \mathbf{U})) \tag{6}$$

The pattern databases are constructed at the beginning of the search based on the parent relation graph. That is, new pattern databases are not created for recursive subproblems.

The pattern databases based on the SCCs are typically smaller than those previously proposed for the entire space. The space complexity of pattern databases created based on l balanced partitions is $O(l \cdot 2^{n/l})$. On the other hand, the space complexity of pattern databases created based on l balanced partitions separately for m SCCs of size $O(\max_{|scc_i|} 2^{|scc_i|})$ is $O(m \cdot \max_{|scc_i|} 2^{|scc_i|/l})$. Thus, the space complexity of the pattern databases based on the SCCs is less than that based on the balanced partitions alone, unless there is only one SCC. In that case, the space complexity is the same.

5.3 CALCULATING THE HEURISTIC VALUE

The heuristic value for node $\mathbf{U}$ in the subproblem for scc_i is calculated in two steps. We first calculate $h_1(\mathbf{U})$, the heuristic value from $\mathbf{U}$ to $\mathbf{PC}_i$, using the the pattern databases described in Section 5.2. Second, we calculate $h_2(\mathbf{U})$, the estimated distance from $\mathbf{PC}_i$ to $\mathbf{V}$, as follows.

$$h_2(\mathbf{U}) = \sum_{j=i+1}^{m} h_1(scc_j) \tag{7}$$

That is, the h_2 value is the sum of h_1 values for the start nodes of the remaining subproblems. Due to the POPS constraints, none of these variables will have been added as leaves when considering the i^{th} subproblem. The total heuristic value is then $h'(\mathbf{U}) = h_1(\mathbf{U}) + h_2(\mathbf{U})$. The h_2 values are the same for all nodes in the i^{th} subproblem, so they can be precomputed.

Theorem 1. *The new heuristic h' is consistent.*

Proof. We prove the theorem by showing that both h_1 and h_2 are consistent. The consistency of h_1 follows from the consistency of the static pattern databases (Yuan and Malone 2013). The h_2 value is a sum of h_1 values for mutually exclusive patterns, so it is also consistent. Therefore, the entire heuristic is consistent. □

6 RELATED WORK

The parent relation graph is, in effect, a *directed superstructure*. Consequently, the work presented in this paper is quite related to the work dealing with superstructures. To the best of our knowledge, Ordyniak and Szeider (2013) are the only other authors to consider *directed* superstructures. They prove that BNSL is solvable in polynomial time for acyclic directed superstructures; our algorithm agrees with this theoretical result because, if the parent relation graph is acyclic, then it will have n SCCs of size 1. Thus, the complexity of our algorithm would be $O(n)$.

The work on undirected superstructures, e.g., (Perrier, Imoto, and Miyano 2008), is also related to our work. Any undirected superstructure can be translated into a parent relation graph by replacing the undirected edges in the superstructure with directed edges in both directions. However, edges directed in only one direction give an order to the SCCs which further reduce the search space. So, our algorithm leverages all of the information available from the undirected superstructure, but further makes use of constraints those structures cannot express.

Recently, Parviainen and Koivisto (2013) explored precedence constraints, which are similar to our POPS constraints. In their work, ideals of partial orders on the variables are used to reduce the search space of dynamic programming for BNSL. This approach is similar in spirit to our use of the component graph to reduce the search space. In fact, the component graph could be used to reduce the search space of dynamic programming. However, after selecting the ideals, they are fixed. So the recursive decomposition described in Section 3.4 is not compatible with the ideals formulation. Experimentally, we show that the

recursive application of constraints is important for some datasets.

Integer linear programming (ILP) (Bartlett and Cussens 2013) is another successful strategy for BNSL. A recent study (Malone et al. 2014) found that the performances of ILP and heuristic search are largely orthogonal, particularly with respect to the number of POPS. Consequently, this work has focused on improvements to heuristic search. Nevertheless, the component graph is readily applicable to ILP by similarly creating subproblems and solving them with independent ILP instances. Indeed, an interesting avenue for future research is to dynamically select between ILP and heuristic search for each subproblem.

6.1 EXPERT KNOWLEDGE CONSTRAINTS

The formulation of BNSL as an optimization over POPS gives a natural method for including expert knowledge in the form of hard constraints on the structure to be learned, such as those proposed by, e.g., (de Campos and Ji 2011). In particular, given expert knowledge about required or forbidden parent relationships and maximum parent set cardinalities, we omit POPS which violate these constraints. The POPS constraints automatically prune the parts of the search space which violate the expert knowledge.

In general, hard expert knowledge constraints are lossy because they could disallow parent sets which would appear in an optimal structure based solely on the data and scoring function. Nevertheless, we still consider the network learned under expert knowledge constraints as optimal. For cases in which we use expert knowledge constraints and the top-p POPS constraint, parent sets disallowed by the expert knowledge constraints are not considered in the suboptimality bound calculation in Equation 5.

7 EMPIRICAL EVALUATION

In order to evaluate the efficacy of the POPS constraints and top-p POPS constraint, we ran a set of experiments on benchmark datasets from the UCI machine learning repository[2] and the Bayesian network repository[3]. We generated $1,000$ records from the benchmark networks in the repository using logic sampling. The experiments were performed on an IBM System x3850 X5 with 16 2.67GHz Intel Xeon processors and 512G RAM; 1TB disk space was used. Our code is available online[4].

Several heuristic search algorithms have been adapted for BNSL. We chose to evaluate A* (Yuan, Malone, and Wu 2011) because of its guarantee to expand a minimal number of nodes; AWA* (Malone and Yuan 2013) because it has been shown to find high quality, often optimal, solutions very quickly; and breadth-first branch and bound (BFBnB) (Malone et al. 2011) because it has been shown to scale to larger datasets by using external memory. We used MDL as the scoring function. In all cases, we used static pattern databases; the variable groups were determined by partitioning the parent relation graph after applying the top-$p = 1$ POPS constraint (Fan, Yuan, and Malone 2014). Pattern database construction occurs only once after constructing the parent relation graph.

7.1 POPS CONSTRAINTS

We first tested the effect of the POPS constraints, which always guarantee learning the globally optimal structure. Table 2 compares the original version of each algorithm to versions using the POPS constraints.

We first considered three variants of A*: a basic version not using POPS constraints; a version using the POPS constraints but not applying them recursively as described in Section 3.4; and a version which uses the recursive POPS constraints. As the table shows, the versions of A* augmented with the POPS constraints always outperform the basic version. The improvement in running time ranges from two times on several of the datasets to over an order of magnitude on three of the datasets. Additionally, the basic version is unable to solve Mildew, Soybean and Barley within the time limit (30 minutes); however, with the POPS constraints, all of the datasets are easily solved within the limit. The number of nodes expanded, and, hence, memory requirements, are similarly reduced.

The recursive POPS constraints always reduce the number of nodes expanded[5]. However, it sometimes increases the running time. The overhead of Tarjan's algorithm to recursively look for SCCs is small; however, in some cases, such as when the parent relation graph is dense, the additional work yields minimal savings. In these cases, despite the reduction in nodes expanded, the running time may increase.

On the other hand, when the parent relation graph is sparse, the advantages of the recursive POPS constraints are sometimes more pronounced. For example, the running time of Mildew is reduced in half by recursively applying POPS constraints. Most networks constructed by domain experts, including those evaluated in this study, are sparse. Our analysis shows that these datasets also yield sparse parent relation graphs. Thus, our results suggest that the recursive constraints are sometimes effective when the generative process of the data is sparse. The overhead of looking for the recursive POPS constraints is minimal, and it sometimes offers substantial improvement for sparse generative processes. So we always use it in the remaining experiments.

[2] http://archive.ics.uci.edu/ml

[3] http://compbio.cs.huji.ac. il/Repository/

[4] http://url.cs.qc.cuny.edu/software/URLearning.html

[5] For some datasets, the precision shown in the table is too coarse to capture the change.

Dataset						Results							
Name	n	N	POPS	Density	PD (s)		A*	A*, O	A*,R	AWA*	AWA*,R	BFBnB	BFBnB,R
Mushroom	23	8124	13025	0.87	0.15	Time (s)	0.74	0.41	0.67	0.75	0.47	0.61	0.78
						Nodes	0.05	0.04	0.04	0.06	0.05	0.06	0.04
Autos	26	159	2391	0.75	0.17	Time (s)	46.62	20.93	26.76	44.70	19.86	11.24	6.68
						Nodes	3.26	1.63	1.63	4.92	2.47	3.26	1.63
Insurance*	27	1000	560	0.35	0.21	Time (s)	98.08	52.81	50.97	118.23	66.75	47.46	28.58
						Nodes	7.83	3.92	3.77	14.51	6.51	8.16	3.77
Water*	32	1000	4022	0.24	0.49	Time (s)	14.10	0.03	0.03	14.10	0.03	32.82	0.80
						Nodes	1.59	0.02	0.02	1.59	0.01	7.10	0.01
Mildew*	35	1000	360	0.16	0.50	Time (s)	OT	5.20	2.33	OT	3.71	OT	3.18
						Nodes	OT	0.56	0.37	OT	0.44	OT	0.36
Soybean	36	307	5926	0.58	0.54	Time (s)	OT	435.65	511.55	OT	526.13	OT	1230.41
						Nodes	OT	9.78	9.64	OT	11.36	OT	129.77
Alarm*	37	1000	672	0.16	1.39	Time (s)	76.51	6.47	4.06	46.98	4.80	22.32	3.83
						Nodes	2.75	0.33	0.24	3.49	0.30	2.75	0.24
Bands	39	277	892	0.26	2.03	Time (s)	109.75	0.39	0.47	74.02	0.40	249.04	1.34
						Nodes	3.63	0.03	0.03	3.99	0.03	41.81	0.03
Spectf	45	267	610	0.24	43.46	Time (s)	89.08	90.62	92.17	44.41	36.79	32.34	29.59
						Nodes	2.26	2.26	2.17	3.17	3.17	2.53	2.44
Barley*	48	1000	634	0.1	0.73	Time (s)	OT	2.51	1.28	OT	1.10	OT	1.85
						Nodes	OT	0.64	0.08	OT	0.21	OT	0.08

Table 2: The number of expanded nodes (in millions) and running time (in seconds) of A*, AWA* and BFBnB with/without the POPS constraints on a set of benchmark datasets. "n" gives the number of variables in the dataset, "N" gives the number of records in the dataset, "POPS" gives the number of POPS after lossless pruning, "Density" gives the density of the parent relation graph constructed from the POPS (defined as the number of edges divided by the number of possible edges), and "PD" gives the time (in seconds) to construct the pattern database. "A*,O" gives the statistics for A* using the POPS constraints, but not applying them recursively. "A*,R" gives the statistics for A* using the recursive POPS constraints. Similarly, "AWA*", "AWA*, R", "BFBnB" and "BFBnB, R" refer to the respective basic algorithms or the algorithm using recursive POPS constraints. "*" indicates the dataset was generated from a repository network using logic sampling; all other datasets are from UCI. OT means out of time (30 minutes).

The anytime window A* algorithm enjoyed improvements similar to those seen in A*. As the table shows, A* always expanded fewer nodes than AWA*; nevertheless, the runtimes of AWA* are often shorter than those of A*. This is because AWA* performs a series of iterations, and *open* is cleared after each of those iterations. Consequently, the associated priority queue operations are often faster for AWA* than A*.

A key factor in the performance for BFBnB is the upper bound it uses for pruning. Previous results (Malone and Yuan 2013) have shown that AWA* is effective at finding high quality solutions quickly, so we found the bound by running AWA* for 5 seconds on datasets with less than 35 variables and 10 seconds for larger datasets. AWA* used the POPS constraints when BFBnB used them. BFBnB exhibited improvements in line with those for A* and AWA*.

7.2 TOP-p POPS CONSTRAINT

We tested AWA* on the dataset Hailfinder, which has 56 variables. Even when using the recursive POPS constraints, though, AWA* was unable to prove optimality within the 30-minute time limit. Therefore, we used this dataset to test the effect of the top-p POPS constraint by varying p from 1 to 13. The upper bound on p was set to 13 because AWA* was unable to complete within the time limit for $p = 13$.

Primarily, we evaluated the running time and associated suboptimality bound as we increased p (which has the effect of pruning fewer POPS). As Figure 4 (top) shows, the recursive order constraints are quite effective under the top-p POPS constraint; the constrained problems are solved in under 15 seconds for p up to 12, and the provable suboptimality bound calculated using Equation 5 decreases very rapidly. This provable suboptimality between the learned network and global optimum is less than 1% even when p is only 7.

The suboptimality bound usually decreases as p increases. From $p = 7$ to $p = 8$, though, it slightly increases; the scores of the learned networks were the same (not shown). This is a result of equivalence classes of Bayesian networks. The suboptimality bound calculation in Equation 5 focuses on parent sets of individual variables, so it is sensitive to which member of an equivalence class the algorithm learns. Future work could investigate tightening the bound by considering all members in the same equivalence class as the learned network.

As mentioned, AWA* was unable to find the provably optimal network under the $p = 13$ constraint. Equation 4 suggests that the size of the largest SCC in the parent relation graph is a key factor in determining the difficulty of an instance of BNSL. However, the recursive POPS constraints offer the potential to split large SCCs after considering a few of their variables. Figure 4 (middle) shows that the size of the largest SCC does substantially increase from $p = 12$ to $p = 13$, which empirically confirms our theoretical result. Somewhat unexpectedly, though, the figure also shows that the density of the parent relation graph does not significantly increase as more POPS are included. So, at least in this case, despite the sparsity of the parent relation graph, the recursive order constraints are unable to break the large SCC into manageable subproblems. This result agrees with those in Table 2 which show that sparsity does not necessarily indicate the efficacy of the recursive POPS constraints.

In addition to the characterstics of the parent relation graph, we also considered the number of POPS included as p in-

creases. Figure 4 (bottom) shows that the number of included POPS follows a similar trend to the density of the parent relation graph. That is, even as p increases, more POPS are not necessarily included for all variables. This is because we already include all subsets of parent candidates from the top p POPS at earlier iterations. Additionally, the number of POPS (479 when $p = 13$) is quite small for this dataset, although the number of variables is relatively large (56). Previous studies (Malone and Yuan 2013) have shown that basic heuristic search methods struggle with datasets like this; however, when augmented with the POPS constraints, heuristic search very quickly finds a network that is provably quite close to optimal. This result clearly shows that the POPS constraints significantly expand the applicablility of heuristic search-based structure learning.

Despite the inability of AWA* to find the provably optimal network under the top-p POPS constraint when $p = 13$, we can nevertheless take advantage of its anytime behavior to calculate a suboptimality bound. At each iteration, AWA* produces an optimal network with respect to its current window size. We can then use Equation 5 to bound the suboptimality of the learned network. In principle, this even suggests that we may be able to prove global optimality before completing the AWA* search, although this likely would require a tighter bound under the top-p POPS constraint than the naive one proposed in this paper.

Even for very small values of p, though the top-p POPS constraint results in networks provably very close to the globally optimal solution. In order to more thoroughly understand why such constrained problems still give provably very high quality solutions, we plotted the scores of the top p POPS for variable X_{37} from Hailfinder in Figure 5. The figure shows that the first 4 scores are much better than the remaining ones; consequently, the globally optimal network is more likely to include one of these parent sets for X_{37} than the others, which are much worse. Most of the other variables behaved similarly; consequently, p does not need to be very large to still encompass most of the parent selections in the globally optimal network.

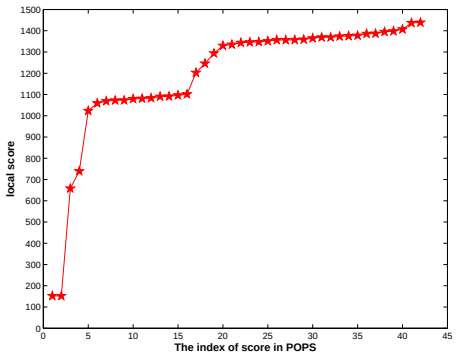

Figure 5: The POPS of variable X_{37} from Hailfinder, sorted in ascending order

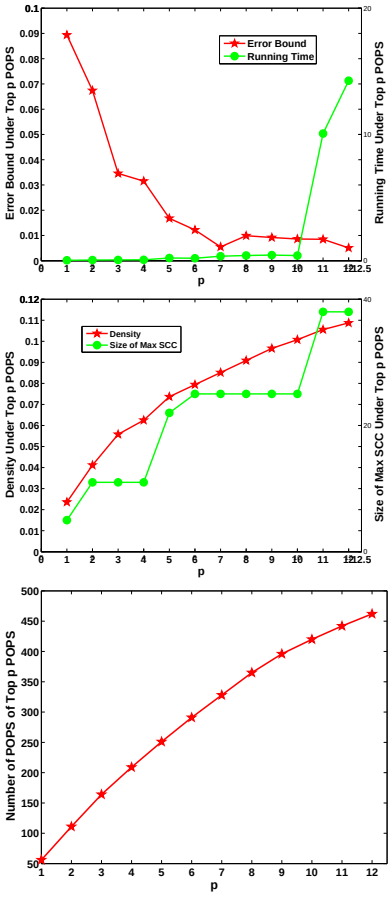

Figure 4: The behavior of Hailfinder under the top-p POPS constraint as p varies. (top) Running time and suboptimality (middle) Size of the largest SCC and density of the parent relation graph (bottom) Number of POPS included

8 CONCLUSION

In this work, we have shown how POPS constraints, which are implicit in the input to a BNSL instance, can significantly improve the performance of heuristic search on the problem. Other algorithms, such as integer linear programming, can also benefit from the POPS constraints. We also introduced the top-p POPS constraint and showed how it can be used to further take advantage of the POPS constraints while still providing guaranteed error bounds. Empirically, we showed that the POPS constraints are practically effective and that the top-p POPS constraint can yield provably very high quality solutions very quickly. Future work includes more thorough empirical evaluation and comparison with other BNSL techniques as well as investigation into conditions when the POPS constraints are most effective.

Acknowledgements This research was supported by NSF grants IIS-0953723, IIS-1219114 and the Academy of Finland (COIN, 251170).

References

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 Artificial Intelligence*, 2250–2255.

Bartlett, M., and Cussens, J. 2013. Advances in Bayesian network learning using integer programming. In *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence*.

Buntine, W. 1991. Theory refinement on Bayesian networks. In *Proceedings of the 7th Conference on Uncertainty in Artificial Intelligence*, 52–60.

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

Cormen, T. H.; Stein, C.; Rivest, R. L.; and Leiserson, C. E. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education.

Cussens, J. 2011. Bayesian network learning with cutting planes. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, 153–160.

de Campos, C. P., and Ji, Q. 2011. Efficient learning of Bayesian networks using constraints. *Journal of Machine Learning Research* 12:663–689.

Edelkamp, S., and Schrodl, S. 2012. *Heuristic Search: Theory and Applications*. Morgan Kaufmann.

Fan, X.; Yuan, C.; and Malone, B. 2014. Tightening bounds for Bayesian network structure learning. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.

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

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions On Systems Science And Cybernetics* 4(2):100–107.

Heckerman, D.; Geiger, D.; and Chickering, D. M. 1995. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning* 20:197–243.

Heckerman, D. 1998. A tutorial on learning with Bayesian networks. In Jordan, M., ed., *Learning in Graphical Models*, volume 89 of *NATO ASI Series*. Springer Netherlands. 301–354.

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*.

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

Lam, W., and Bacchus, F. 1994. Learning Bayesian belief networks: An approach based on the MDL principle. *Computational Intelligence* 10:269–293.

Malone, B., and Yuan, C. 2013. Evaluating anytime algorithms for learning optimal Bayesian networks. 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*, 479–488.

Malone, B.; Kangas, K.; Järvisalo, M.; Koivisto, M.; and Myllymäki, P. 2014. Predicting the hardness of learning Bayesian networks. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.

Ordyniak, S., and Szeider, S. 2013. Parameterized complexity results for exact Bayesian network structure learning. *Journal of Artificial Intelligence Research* 46:263–302.

Ott, S.; Imoto, S.; and Miyano, S. 2004. Finding optimal models for small gene networks. In *Pacific Symposium on Biocomputing*, 557–567.

Parviainen, P., and Koivisto, M. 2013. Finding optimal Bayesian networks using precedence constraints. *Journal of Machine Learning Research* 14:1387–1415.

Perrier, E.; Imoto, S.; and Miyano, S. 2008. Finding optimal Bayesian network given a super-structure. *Journal of Machine Learning Research* 9:2251–2286.

Silander, T., and Myllymäki, 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.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2):146–160.

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.