End-to-end Bayesian Networks Exact Learning in Shared Memory

Subhadeep Karan, Zainul Abideen Sayed, and Jaroslaw Zola, *Senior Member, IEEE*

Abstract—Bayesian networks are important Machine Learning models with many practical applications in, e.g., biomedicine and bioinformatics. The problem of Bayesian networks learning is $\mathcal{NP}$-hard and computationally challenging. In this paper, we propose practical parallel exact algorithms to learn Bayesian networks from data. Our approach uses shared-memory task parallelism to realize exploration of dynamic programming lattices emerging in Bayesian networks structure learning, and introduces several optimization techniques to constraint and partition the underlying search space. Through extensive experimental testing we show that the resulting method is highly scalable, and it can be used to efficiently learn large globally optimal networks.

Index Terms—Bayesian networks, exact learning, task parallelism.

1 Introduction

Bayesian networks are important class of probabilistic graphical models used in Machine Learning [1]. In the most basic form, they provide computationally feasible representations of multivariate probability distributions. When extended with prior information, they can express causal relationships, which are essential for data-driven decision making [2]. Because Bayesian networks are generative, support speculative queries, and are relatively easy to interpret [3] they are frequently favored in real-world data analytics. For instance, in systems biology, Bayesian networks are built from gene expression data to simplify analysis of potential gene regulatory interactions [4]. In clinical decision support systems, Bayesian networks are frequently constructed from Electronic Health Records (EHRs) to deliver the most likely diagnosis and prognosis of patients [5]. In banking, Bayesian networks facilitate fraud detection while maintaining transparency and clarity behind flagging given transaction as fraudulent [6]. Finally, because Bayesian networks provide tractable representations of multivariate probability distributions, they are frequently used to augment other ML techniques (e.g., deep neural nets [7]).

The versatility of Bayesian networks comes at the price: to be meaningful and interpretable a Bayesian network should be based on a high quality structure (network structure is a directed acyclic graph linking random variables – we provide details in Section 2). When a Bayesian network comprises only a small number of variables, that additionally are well understood, its structure can be solicited from domain experts who use prior knowledge and experience to characterize (potentially causal) dependencies between the variables. However, this approach becomes infeasible when the number of variables is large, or the application involves *de novo* exploration with no prior background information. In such situations, computational approaches that learn network structure directly from the data become necessary. But the structure learning problem has been demonstrated to be $\mathcal{NP}$-hard [8], and is known to be computationally challenging [9]. As a result, the existing structure learning methods rely primarily on heuristics, compromising quality and interpretability to achieve tractability and speed.

Although in some applications heuristics deliver sufficiently accurate solutions, the question of finding provably optimal network structure given some network evaluation criterion (see Section 2) remains practically and theoretically relevant. First, the ability to compute exact structure simplifies comparison of different statistical methods (evaluation criteria) that are used in network learning. This is because given the exact structure, the quality of the Bayesian network depends only on the input data and the underlying evaluation criteria. Second, in many applications, especially in biomedical informatics, having exact structure eliminates the structure as a source of model uncertainty, and that is critical for interpretation (structure depends only on the data and not the learning algorithm). Finally, the problem is computationally interesting and scalable exact learning algorithms can serve as benchmarks for developing or improving approximations and heuristics.

In this paper, we propose a new parallel approach for end-to-end exact learning of Bayesian networks. Here, end-to-end means that our approach starts from a raw input data, and delivers a directed acyclic graph representing an optimal network structure given user-defined scoring criterion. Our approach uses shared-memory task parallelism to realize exploration of dynamic programming lattices emerging in Bayesian networks structure learning, and introduces several optimization techniques to constraint and partition the search space and to mitigate inter-task synchronization. In the paper, we report the following specific contributions:

- We describe the end-to-end Bayesian network learning process that decomposes learning into computationally independent stages of maximal parent set assignment and optimal structure search, which both involve dynamic programming.
- We propose Optimal Path Extension technique to

• *Authors are with the Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY, 14203.*
E-mail: {skaran,zsayed,jzola}@buffalo.edu

XXX

constrain the dynamic programming lattice in the structure search stage and prove that it is guaranteed to deliver an optimal network structure.
- We introduce parallel algorithms to efficiently traverse the dynamic programming space in both problems, and show their realization using task parallelism and custom data structure to manage search frontier.
- We report extensive experimental results using the classic benchmark networks to assess scalability of our methods and dissect effectiveness of our proposed optimization techniques.

The reminder of this paper is organized as follows. In Section 2, we provide basic definitions and we formulate the BN learning problem. In Section 3, we introduce our learning strategy, including problem decomposition and Optimal Path Extension technique. In Section 4, we describe our parallel algorithms and data structure to realize BN learning using task parallelism on shared memory systems. Finally, in Section 5, we report experimental results assessing scalability of our proposed solutions. We close the paper with a brief review of related work in Section 6 and conclusions in Section 7.

2 Preliminaries

Given a set of n random variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, Bayesian network (BN) is a tuple (G, P), where G is a directed acyclic graph (DAG) over $\mathcal{X}$, and P is a joint probability distribution of $\mathcal{X}$ that factorizes according to G. Specifically, we have that $P(\mathcal{X}) = \prod_{i=1}^{n} P(X_i|Pa(X_i))$, where $Pa(X_i) \subseteq \mathcal{X} - \{X_i\}$ is a set of all parents of X_i in G. Here parent of X_i is any variable $X_j \in \mathcal{X}$ for which there exists edge $X_j \rightarrow X_i$ in G. In practical applications, $\mathcal{X}$ represents objects that we wish to model (e.g., attributes of a patient in Electronic Health Records), and G captures statistical or causal relationships between the attributes (e.g., smoking causes cancer).

Suppose that we are given a complete categorical data table of m records, $D = [D_1, \ldots, D_m]$, where each row $|D_i| = n$ stores a single observation of all n variables in $\mathcal{X}$, and variable $X_j \in calX$ assumes r_j states. The goal of Bayesian network structure learning is to find a graph G that best explains (or fits) the data in D.

In this paper, we consider the score-based BN learning in which structure learning is formulated as an optimization problem. Let $Score(G{:}D)$ be a scoring function evaluating quality of the network structure G with respect to the input data D. Furthermore, let the function be decomposable, that is:

$$Score(G{:}D) = \sum_{X_i \in \mathcal{X}} s\left(X_i, Pa(X_i)\right),$$

where $s\left(X_i, Pa(X_i)\right)$ is a local score contribution of X_i when its parents in G are $Pa(X_i)$, evaluated directly from the data. Given D and s we want to efficiently find G that minimizes $Score(G{:}D)$. In practice, there are several commonly used scoring functions, for example, MDL [10], BIC [11], BDeu [12]. Here we do not focus on any particular one but we exploit the fact that $Score$ is decomposable with respect to s. Moreover, we leverage the fact that many of these functions can be efficiently bounded based on the input data D.

Before we proceed further, we should note that the search space of all potential network structures is exponential, and consists of

$$C(n) = \sum_{i=1}^{n} (-1)^{(i+1)} \binom{n}{i} 2^{i(n-i)} C(n-i)$$

DAGs with n nodes [13]. Hence, any approach based on the direct enumeration becomes impractical for both shared and distributed memory systems, necessitating that we constraint the search space while guaranteeing that a global minimum remains reachable. Over the years, the problem has been investigated by different teams using multiple approaches including classic dynamic programming [14], Integer Linear Programming [15] or shortest path search [16]. Interestingly, performance of the existing methods varies depending on the properties of the input data, and no one method dominates the others (we refer interested readers to [9] for the in-depth analysis).

3 Structure Learning Strategy

As we already mentioned, we treat the BN structure learning as an optimization problem in which the search space consists of all possible DAGs over the set of nodes $\mathcal{X}$. Now observe that any DAG is equivalently represented by one of its topological orderings over $\mathcal{X}$. A topological ordering implies that if $X_j \in \mathcal{X}$ is a parent of $X_i \in \mathcal{X}$, that is $X_j \in Pa(X_i)$, then it must precede X_i in that ordering, i.e., $X_j \prec X_i$. Moreover, the relative ordering of the parents of X_i is irrelevant. By combining this property with the fact that the scoring function is decomposable, we can express the search problem via dynamic programming. This general idea has been exploited in different variants, for example in [14], [16], [17], [18], [19], and it works as follows. Because any DAG must have at least one sink node (i.e., a node without descendants), we can first identify an optimal sink and find its optimal parents. Then, we can continue with the remaining nodes recursively organizing them into an optimal structure. Since we know that a sink node has no successors, it can be placed at the end of the topological order we are building. Let $d(X_i, U)$, $U \subseteq \mathcal{X} - \{X_i\}$, be the score of selecting *optimal parent set* of X_i from variables in U, that is $d(X_i, U) = \min_{Pa(X_i) \subseteq U} s(X_i, Pa(X_i))$. We can efficiently express d via the following recursion:

$$d(X_i, U) = \min \begin{cases} s(X_i, U), \\ \min_{X_j \in U} d(X_i, U - \{X_j\}). \end{cases} \quad (1)$$

Then, the optimal sink is a node that minimizes the sum of scores of two sub-networks, one that includes the sink, and the other that spans over the remaining nodes. If we denote an optimal score of a network over $U \subseteq \mathcal{X}$ by $Q^*(U)$, then we have:

$$Q^*(U) = \min_{X_i \in U} (d(X_i, U - \{X_i\}) + Q^*(U - \{X_i\})), \quad (2)$$

and by using dynamic programming to compute $Q^*(\mathcal{X})$ we can construct an optimal topological ordering of $\mathcal{X}$.

The dynamic programming for Q^* can be visualized as operating on the lattice $\mathcal{L}$ with $n+1$ levels formed by the partial order *set inclusion* on the power set of $\mathcal{X}$ [14], [16], [18] (see Figure 1a). Two nodes in the lattice, $U' \subseteq \mathcal{X}$ and $U \subseteq \mathcal{X}$, are connected only if $U' \subset U$ and $|U| = |U'| + 1$. Here we use U to denote both a subset of $\mathcal{X}$ and the corresponding node in the lattice $\mathcal{L}$. We may be tempted to attack the problem directly by handling recurrences for Q^* and d at the same time (as for example in [18]). However, that would necessitate memoizing both Q^* and d. If we imagine dynamic programming as progressing in the top-down manner over the lattice $\mathcal{L}$, then the memory complexity of memoization is $\Theta\left(\binom{n}{\frac{n}{2}}\right)$, which is the number of nodes in the largest layer of $\mathcal{L}$, and computational complexity is proportional to $\Theta\left(n \cdot 2^n\right)$ edges in the lattice. This quickly becomes prohibitive for any but small number of variables.

In [16], Yuan et al. observed that finding an optimal network structure (i.e., its topological ordering) is equivalent to finding a shortest path from the root to the sink in the lattice with weights prescribed by d. The key fact is that any path from the root of the lattice $\mathcal{L}$ to one of its nodes is equivalent to a specific ordering of variables in that node. Moreover, an edge (U', U) has weight $d(U - U', U')$. For instance, the path marked in Figure 1a represents ordering $[X_3, X_2, X_4, X_1]$, and edge $(\{X_3\}, \{X_2, X_3\})$ has weight computed as $d(X_2, \{X_3\})$. This formulation allows for easy decomposition of the structure learning problem into two sub-problems, referred to as *parent sets assignment* and *structure search*, which roughly correspond to Eqs. (1) and (2). This decomposition in turn provides ample opportunities for optimization, and hence we adopt it in our approach.

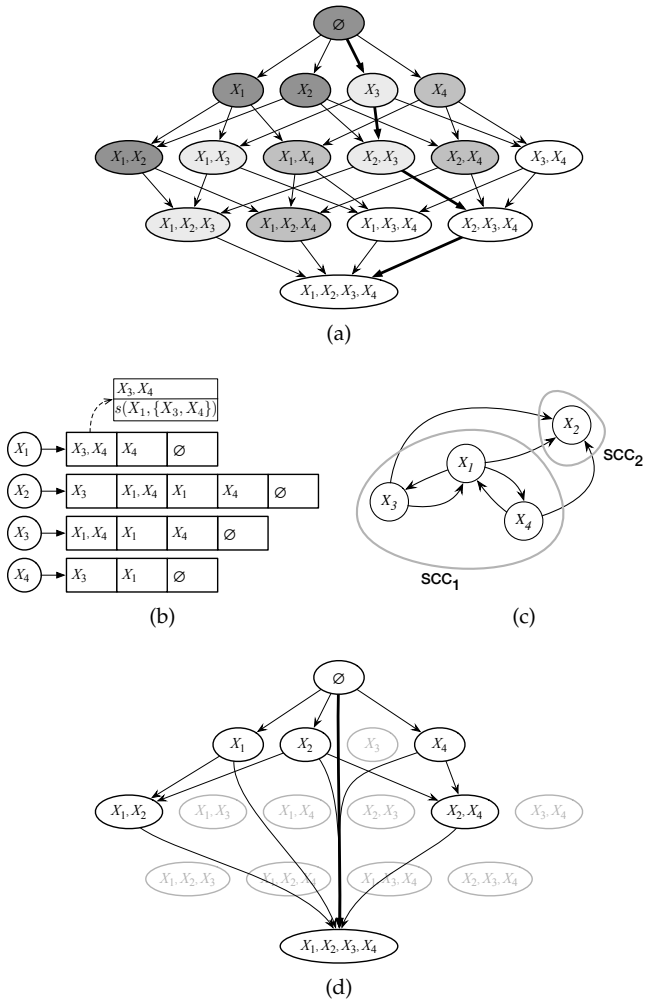

Fig. 1: (a) Dynamic programming lattice for the problem with four variables. (b) Example maximal parent sets list where for each variable X_i an ordered vector of tuples $(U, s(X_i, U))$ is stored. (c) Graph $G'_{\mathcal{X}}$ and its SCCs constructed from the parent sets in (b). (d) Constrained lattice created by applying OPE as prescribed by the parent sets list in (b). Let path marked in bold be the optimal solution. By constraining the dynamic programming lattice, after discovering node $\{X_3\}$ a search algorithm can follow directly to the final node.

3.1 Parent Sets Assignment

Because the function $Score(G{:}D)$ is decomposable with respect to s, we could start solving our problem by finding, for each $X_i \in \mathcal{X}$, a subset $Pa(X_i) \subseteq \mathcal{X} - \{X_i\}$ such that $s(X_i, Pa(X_i))$ is minimized. This task is known as the *parent set assignment* [20], and appears not only in BN learning but also in related ML techniques, e.g., Markov blankets. The task can be approached by solving the recursion in Eq. (1) for $U = \mathcal{X} - \{X_i\}$. However, to do this efficiently, it is advantageous to consider a slightly broader version of the parent set assignment problem.

We will say that $U \subseteq \mathcal{X} - \{X_i\}$ is a *maximal candidate parent set* of X_i if $d(X_i, U) = s(X_i, U)$. From Eq. (1) we have that if U is a maximal candidate parent set of X_i, then no subset of U has score better than $d(X_i, U)$, i.e., $\forall_{U' \subset U} d(X_i, U) < d(X_i, U')$. Hence, by identifying all maximal candidate parent sets of X_i and memoizing their corresponding scores s, we can efficiently answer queries about any optimal parent set of X_i. Specifically, to answer query $d(X_i, U')$ for any U' it is sufficient to check if U' is one of the maximal candidate parent sets of X_i. If it is, then all we have to do is to return the memoized score s of that maximal candidate parent set. Otherwise, $d(X_i, U')$ must be equal to the smallest s among all maximal candidate parent sets of X_i for which U' is a superset. This implies that the maximal candidate parent set with the smallest s will be representing answer to the query $d(X_i, \mathcal{X} - \{X_i\})$. For convenience, we will refer to such maximal candidate parent set as *optimal parents* of X_i.

Because the set of all maximal candidate parent sets is commonly several orders of magnitude smaller than all possible values of d, maintaining only that set is advantageous from the memory use perspective. In Section 4.1, we propose efficient parallel algorithm to enumerate and store all maximal candidate parent sets. Our method delivers a lookup table that for each X_i maintains an ordered vector of tuples $(U, s(X_i, U))$, where U is a maximal candidate parent set (see example in Figure 1b). Tuples are sorted in the ascending order of s, and we use binary encoding to represent set U, i.e., bitmap set representation. This binary representation allows for $O(1)$ set containment and set equality checking for queries involving U, as long as the

total number of variables is a small constant factor of the word size of the executing hardware (e.g., $n \leq 128$ on a 64-bit architecture). By keeping vectors ordered, we can get optimal parents of X_i and their score in $O(1)$, and we can answer arbitrary query $d(X_i, U)$ in $O(l)$, where l is the size of the vector for X_i. This is because for a given X_i its optimal parents $d(X_i, \mathcal{X} - \{X_i\})$ will be stored as the first entry of the corresponding sorted vector, and to answer $d(X_i, U)$ we have to find the first maximal candidate parent set that is a subset of U.

3.2 Structure Search

Given an efficient strategy to compute $d(X_i, U)$ for any pair (X_i, U), provided by the solution to the parent set assignment, the structure search becomes a variant of the shortest path problem. As explained earlier, by finding the shortest path from the root to the sink of the lattice $\mathcal{L}$ we are constructing a specific topological ordering of the input variables. While that may suggest a simple solution based on classic approaches like the Dijkstra's algorithm, BFS or A-star, we have to keep in mind that the size of the underlying graph (lattice) is exponential in n, and hence instantiating it entirely is infeasible. Furthermore, even if the graph is instantiated on the fly, the memory requirement becomes prohibitive for a single server. For example, in BFS at least two consecutive layers of the lattice $\mathcal{L}$ have to be maintained in memory, and in A-star open and closed lists may grow excessively depending on the quality of the heuristic function used. Consequently, to scale up in the shared memory regime, it is critical to further constraint the search space, i.e., reduce the number of nodes in the dynamic programming lattice that have to be actually visited. To achieve this, we introduce the *Optimal Path Extension* technique (OPE).

Optimal Path Extension

Consider a node U at the level k in the lattice $\mathcal{L}$ (the root of the lattice is at level $k = 0$). This node has k incoming edges and $n - k$ outgoing edges. Each of the outgoing edges corresponds to one particular way in which U, and thus any of the orderings that it represents, can be extended. However, in many cases we can immediately identify extensions (i.e., outgoing edges) of U that are guaranteed to be part of the optimal path from U to the sink of the lattice.

To decide if a node U can be optimally extended we build the following observation. Suppose that U is a superset of the optimal parents of some variable $X_i \notin U$. Then it follows that in any valid topological ordering of $\mathcal{X}$ induced by a path passing through U, X_i must be preceded by all variables in U (since X_i must be preceded by its parents). Equivalently, any path from U to the sink of the lattice $\mathcal{L}$ will have to pass through a node containing X_i. Furthermore, consider a node U' on a path from U to the sink, such that $X_i \notin U'$. Because $U \subset U'$ we have that $d(X_i, U') = d(X_i, U)$ as both U and U' are supersets of the optimal parents of X_i. Now take any optimal path from U to the sink of $\mathcal{L}$, and its corresponding ordering of variables in $\mathcal{X} - U$. We can transform that ordering such that X_i is at the first position and the remaining variables maintain their relative positions. Because placing X_i at the beginning guarantees that it precedes all variables it was preceding in the original ordering, we have that the cost of the path induced by the transformed ordering remains unchanged and optimal. This proves the following theorem:

Theorem 1 (Optimal Path Extension, Karan et al. [21][1])**.** *Let U be a superset of the optimal parents of $X_i \in \mathcal{X} - U - \{X_i\}$. Then, there exists an optimal path from U to the sink of the lattice $\mathcal{L}$, in which U is followed by $U \cup \{X_i\}$.*

To illustrate practical implications of the optimal path extension idea, consider example dynamic programming lattice $\mathcal{L}$, and maximal candidate parent sets in Figure 1. The optimal parents of X_2 and X_4 consist of X_3 only (in Figure 1b, X_3 is the first maximal candidate parent set for both X_2 and X_4). Now take node $U = \{X_3\}$. Since U is a superset of the optimal parents of X_2 and X_4, from Theorem 1 to extend U it is sufficient to consider one of only these two variables. Suppose that we extend U by adding X_2. The new node $\{X_2, X_3\}$ with the ordering $[X_3, X_2]$ remains the superset of the optimal parents of X_4. Thus, we can further extend $\{X_2, X_3\}$ by adding X_4 with the corresponding ordering $[X_3, X_2, X_4]$. In the final step, we can extend one more time by adding X_1, hence reaching the sink of the lattice. In a similar way, we can extend $\{X_1, X_4\}$ by including X_3 and then X_2. Naturally, it will not be always possible to apply the path extension. For example, nodes $\{X_1\}$ and $\{X_1, X_2\}$ cannot be extended as no variable has optimal parents that would be a subset of either of them. In such cases, a search procedure will have to explore all possible outgoing paths. In Figure 1d, we show the final compacted lattice obtained by applying all possible optimal path extensions.

By applying OPE, which can be easily integrated with any shortest path solver (e.g., BFS, A-star, etc.), we can significantly reduce the number of nodes and edges that have to be considered when searching the dynamic programming lattice. The extent to which the reduction can be performed will depend on the size of the set of maximal candidate parent sets of each variable – smaller the set, higher the chance that the optimal path extension can be applied. Moreover, OPE will be more effective for larger problem instances (i.e., problems with larger $\mathcal{X}$) because the dynamic programming lattice will include more nodes with a potential to extend.

4 Parallel Bayesian Networks Learning

We are now ready to present our parallel approach. As mentioned earlier, in this work our focus is on shared memory systems and we elect to use task-based parallelism as it provides ideal level of abstraction for the problem at hand.

4.1 Parallel Parent Sets Identification

Given a set of variables $\mathcal{X}$, database of observations D, and a scoring function s, our first task is to enumerate and store all maximal candidate parent sets for all $X_i \in \mathcal{X}$. Let $L(X_i)$ be a list of maximal candidate parent sets of X_i, where each parent set U is represented by a tuple $(U, s(X_i, U))$. To construct the list, we can directly apply recursion in Eq. (1) and starting from the empty set we can consider

1. The theorem and proof are corrected version of those from [21].

parent sets of growing size. Similar to structure search, this process can be though of as a top-down traversal of the dynamic programming lattice with n levels formed by set inclusion on the power set of $\mathcal{X} - \{X_i\}$ (see Figure 2a). A node U in this lattice represents a potential parent set of X_i, and its edges represent precedence constraints. When the node is discovered (processed), we compute $s(X_i, U)$, compare it with scores d computed for the predecessors of U, this way obtaining $d(X_i, U)$. Then, if U is a maximal candidate parent set we simply add it to $L(X_i)$ such that the ordering of $L(X_i)$ is preserved. These steps form a single computational task.

While the above strategy is clearly guaranteed to enumerate all maximal candidate parent sets, it is both computationally and memory challenging if used directly (due to the size of the lattice). Moreover, while lattices for each variable $X_i \in \mathcal{X}$ can be processed independently, the resulting embarrassing parallelism is highly limited. This is because the computational cost for a single lattice is exponential in n, which effectively constraints the total number of lattices (and hence variables) we may hope to process. For example, if we assume $n = 48$ then the estimated memory requirements to process a single lattice, with $O(2^n)$ nodes and 16 bytes to represent a node, is around 4 PB with a modest 48-way parallelism.

Constraining the Search Space

For every variable X_i it is reasonable to expect that its optimal parent set will not contain all other variables. In other words, there is a bound on the depth to which we should be exploring the dynamic programming lattice of X_i. It turns out that such bounds had been derived for all major scoring functions s, such as, e.g., AIC [22] or BDeu [23]. In short, these bounds exploit the fact that the input data D is finite, specifically, it has a limited number of observations, and hence usually does not provide sufficient information to justify conditioning a variable on more than few other variables (we previously demonstrated this effect for MDL function [24]). The example result of constraining the search space is presented in Figure 2b.

Folding Lattices

In the dynamic programming lattice for variable X_i, before search space bounds become effective, we have to manage $\binom{n-1}{l}$ nodes at the level l of the lattice. Consequently, the memory required to represent the entire layer l is bounded by $B_1 = c_1 \cdot n \cdot \binom{n-1}{l}$, assuming cost c_1 to store a node. This easily becomes problematic for larger problems as soon as $l > 2$. However, we can fold the dynamic programming lattices such that the nodes for the same set U across different lattices are represented by a single node, and handled by a single computational task (see Figure 2c). Let the memory taken by such combined node be c_2. The memory requirement of the new lattice is $B_2 = c_2 \cdot \binom{n}{l}$. This gives us $\frac{B_1}{B_2} = \frac{c_1}{c_2} \cdot (n-l)$ reduction in the memory complexity. To represent a node we use a bitmap in which i-th bit indicates whether element i is in the set. In such case, $c_2 = 2 \cdot c_1$, since

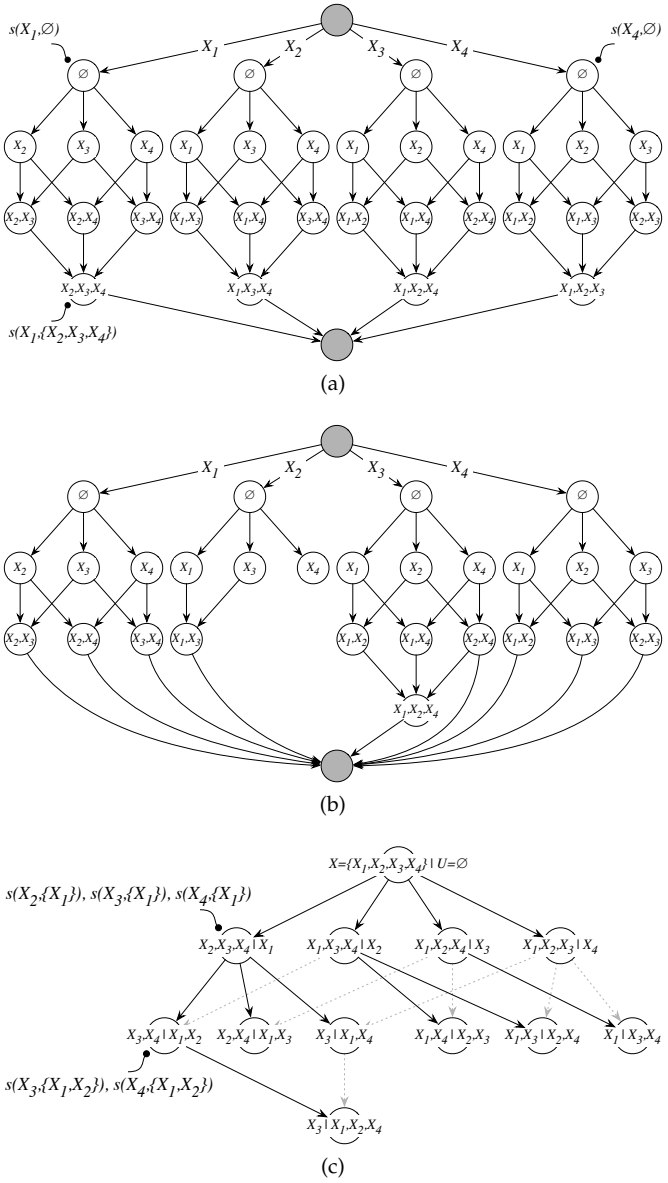

Fig. 2: (a) Example of the dynamic programming lattices for $\mathcal{X} = \{X_1, X_2, X_3, X_4\}$. Processing node U in the lattice for variable X_i requires computing $s(X_i, U)$ and access to $d(X_i, U')$ for each predecessor U' of U. (b) Example constrained lattices, and (c) their corresponding "folded" representation. A node U in the compacted lattice requires that s is evaluated for several variables that share candidate parents U. This improves efficiency of computing s, decreases memory requirements and increases computational density. The precedence constraints marked with dotted gray edges are eliminated to bypass synchronization overhead.

in the compacted lattice we require one bitmap to indicate which variables X_i share U, and one bitmap to represent the actual set U (versus storing only U as in the original lattice).

The main advantage of the folding step is significantly increased computational density. To process a node in the folded lattice, we perform multiple evaluations of s with the same parent set U. Without explaining details of how s is computed from D (we invite the reader to [25] for in-depth presentation), we note that by having the same parents in

the consecutive invocations of s, we can precompute statistics about D induced by U, and reuse them from one invocation to another. Consequently, the cost of processing a task in the folded lattice is higher than the cost of processing an individual related task in the original lattice, but it is lower than the total cost of processing all corresponding tasks from the original lattices, i.e., if X is a set of random variables sharing U we have that $T(s(X,U)) < \sum_{X_i \in X} T(s(X_i,U))$, where T is the processing cost.

Parallel Exploration

We traverse the folded dynamic programming lattice in the BFS order. Each node in the search front, where search front corresponds to a layer in the lattice, is handled by a task, and the entire front is processed in parallel (in our implementation, we use Intel TBB tasks abstraction). Consider a task for node U at the layer l. That task is responsible for performing evaluation of the function s, and checking the bounding conditions. Now suppose that the resulting search space constraints hold. In such case, no task that corresponds to a superset of U should be generated and included for processing in the layer $l+1$ (as it cannot contribute new maximal candidate parent sets). In other words, when processing given layer we should see only tasks whose predecessors all did not satisfy the search space constraining conditions. However, to enforce this requirement we would need either complex synchronization between all tasks within the same layer, or a filter operation on all possible tasks for the next layer, which effectively would defy the purpose of constraining the search space (as we would first generate all nodes/tasks for the next layer and only then prune them).

To address this challenge, we propose a new exploration strategy in which tasks for the next layer are enumerated such that the synchronization is bypassed at the small cost of considering a few unnecessary tasks in the next layer. To this end, we first order variables in $\mathcal{X}$ by the decreasing number of states they have in D, i.e., for any X_i and X_j, if $i < j$ then $r_i \geq r_j$ meaning that X_i has higher arity than X_j (recall that variable X_i can assume on of r_i states). When deciding whether a task should be generated for the next layer, instead of checking if any of its predecessors satisfied pruning condition, which would require synchronization, we check only one selected predecessor. Specifically, let $X_j \in U$ be the maximal element in U. To enumerate descendants of U, we consider only $U' = U \cup \{X_k\}$ for all $k > j$. Thus, node U becomes a predecessor to $n-j$ nodes (solid black edges in Figure 2c). For example, consider node $(X_3, X_4|X_1, X_2)$ in Figure 2c. For this node to be considered during the search, tasks corresponding to the nodes $(X_2, X_3, X_4|X_1)$ and $(X_1, X_3, X_4|X_2)$ would both have to agree whether pruning condition holds or not (by synchronization). In our proposed approach, only task $(X_2, X_3, X_4|X_1)$ makes this decision, eliminating the synchronization point.

The above procedure is motivated by the properties of bounding functions (we invite the reader to review our work in [24] for detailed discussion). In the essence, nodes that in our scheme are predecessors to the largest number of nodes in the next layer are most likely to satisfy the pruning conditions. While this approach does not guarantee that all tasks that should be pruned will not be generated, it works very well in practice. In fact, in our experiments we found that we remove no less than 97% of all tasks that should be pruned. The remaining 3% constitute an extra work of processing nodes that do not contribute maximal candidate parent sets. Note that these nodes once processed never create successors and thus the extra work overhead does not propagate between BFS layers.

Recall that to decide whether node U at layer l is a maximal candidate parent set for X_i, we require scores $d(X_i, U')$ for all subsets $U' \subset U$ from the layer $l-1$. Because each task at layer l may contribute new maximal candidate parent sets that must be available to all tasks in the subsequent layers, we maintain all $L(X_i)$ as a global state that is updated between BFS layers. Each thread maintains a local copy of $L(X_i)$ on which parallel reduction with list merging as an operator is performed after given layer is entirely processed. This step is easy to execute efficiently considering a small size of $L(X_i)$.

4.2 Parallel Structure Search

Given the list $L(X_i)$ of maximal candidate parent sets for each variable X_i, our final step is to find an optimal network structure. Recall that this problem is a variant of the shortest path problem in the corresponding lattice $\mathcal{L}$. It is challenging due to the memory requirements, which can be addressed by the application of our OPE technique, and due to the non-trivial precedence constraints, which OPE amplifies. Specifically, because OPE introduces shortcuts in the lattice (see Figure 1c) we need a strategy to efficiently maintain parallel search front. Here standard methods commonly used in parallel BFS algorithms are infeasible since they assume that all nodes in the search space are known and indexed *a priori*. Hence in our approach, we first augment OPE with search space decomposition to further reduce the search space size, then we propose a hashing scheme to efficiently keep track of the search front while executing parallel BFS.

Strongly Connected Components Decomposition

Consider a graph $G'_{\mathcal{X}}$ over all variables in $\mathcal{X}$ such that every variable X_i has incoming edges only from variables contained in its maximal candidate parent sets $L(X_i)$, i.e., $G'_{\mathcal{X}} = (\mathcal{X}, E), E = \{(X_j, X_i) | \exists_{U \in L(X_i)} X_j \in U\}$. By construction, this directed and potentially cyclic graph must contain a DAG that represents optimal network structure. It means also that edges that do not appear in $G'_{\mathcal{X}}$ cannot be part of the final optimal network structure for $\mathcal{X}$. We can decompose $G'_{\mathcal{X}}$ into strongly connected components (thus partitioning $\mathcal{X}$), learn sub-network for each partition, and obtain the final structure by considering edges between the strongly connected components (SCCs). We note that this fairly intuitive idea was reported before and independently by several groups, e.g., [26], [27]. In Figure 1c, we provide example SCCs for maximal candidate parent sets in Figure 1b. Here we would first learn network for component denoted SCC_1 and then following edges to SCC_2 we could obtain the final network.

The SCC decomposition is guaranteed to preserve optimal network structure as long as each sub-network is optimal (that can be proved by, e.g., observing that ordering of variables between SCCs will always be consistent with an optimal ordering). It has the advantage that learning optimal network for each partition is more tractable since even small decrease in the number of input variables translates into significant savings due to the exponential cost of learning. However, depending on the size of $L(X_i)$ the partitioning may not be feasible since the resulting $G'_{\mathcal{X}}$ may consist of single SCC. Moreover, the SCCs have to be processed following precedence constraints induced by the edges between SCCs (e.g., in Figure 1c, SCC_1 must be processed before SCC_2). This is because ordering of the variables in a preceding SCC decides which variable will be at the root of the subsequent SCC. To tackle these issues we show how to effectively combine SCC decomposition with OPE.

Parallel OPE with SCC Decomposition

To execute the parallel structure search we perform BFS traversal starting from the root of the lattice $\mathcal{L}$. When processing a node $U' = U \cup \{X_i\}$ of the lattice, we first try to apply OPE, which essentially means detecting a shortcut within the lattice (as we explain in Section 3.2). Once possible extensions from U' are exhausted (i.e., we cannot apply OPE anymore) we narrow down the processing to the remaining variables $\mathcal{X} - U'$. Specifically, we consider graph $G'_{\mathcal{X}-U'}$ which we decompose into SCCs that are next handled one-by-one to discover more possible extensions. These high level processing steps are applied to every node in the search front in parallel, and are summarized in Algorithms 1 and 2.

Algorithm 1 StructureSearch()

```
1: Q[0] ← {∅}
2: for k ← 1 ... n pardo
3:    Q[k] ← ∅
4: for k ← 0 ... n do
5:    if Q[k] ≠ ∅ then
6:       for all U ∈ Q[k] and X_i ∈ {𝒳 − U} pardo
7:          ProcessNode(Q, U, X_i)
8: return Q[n]
```

Our BFS search strategy, outlined in Algorithm 1, processes the search front in parallel. The key enabling element is data structure Q, which is our custom hash table (described in detail in the following section). This structure tracks the search front by maintaining a list of nodes from the lattice $\mathcal{L}$ that should be processed at level k. Hence, in line 1, $Q[0]$ is initialized with the root of the lattice, and the final result is stored in $Q[n]$ (see line 8). Moreover, for the purpose of algorithm explanation and correctness analysis, we assume that Q guarantees strict consistency and atomic execution of Find-Update-Or-Insert operation. We should note also that in our implementation, every node U keeps information about the ordering of its constituent variables (denoted by $U.path$) as well as the length of the path ending in that node (denoted by $U.d$). While storing path information in a node may seem wasteful, the space overhead is actually small while it enables us to eliminate synchronization that would be required to keep track of the optimal path, and simplifies implementation of the OPE and SCC optimizations. Finally, the main computational work happens in line 6, where for each node U in the current search front we consider all outgoing edges indexed by variables $X_i \notin U$. As previously, the entire loop is executed by parallel tasks.

Algorithm 2 ProcessNode(Q, U, X_i)

```
1: if X_i ∈ U.subopt then
2:    return
3: U' ← U ∪ {X_i}
4: U'.path ← U'.path + X_i
5: U'.d ← U'.d + d(X_i, U)   ◁ retrieved efficiently from L(X_i)
6: U'' ← OPE(U')   ◁ apply Optimal Path Extension to U'
7: R ← Q[|U''|].Find(U'')
8: if R ≠ ∅ then
9:    if R.d > U''.d then
10:      Q[|U''|].Update(U'')   ◁ keep shorter path
11:   else
12:      return
13: S ← TarjanSCC(G'_{𝒳−U''})   ◁ get ordered list of SCCs
14: k ← 0
15: while k < |S| and |S_k| < τ do
16:   U'' ← U'' + OptOrder(S_k)   ◁ optimal structure for S_k
17:   k ← k + 1
18: k ← k + 1   ◁ advance k to skip one SCC
19: while k < |S| do
20:   U''.subopt ← U''.subopt ∪ S_k
21:   k ← k + 1
22: R ← Q[|U''|].Find(U'')
23: if R ≠ ∅ then
24:   if R.d > U''.d then
25:      Q[|U''|].Update(U'')
26:   else
27:      Q[|U''|].Insert(U'')
```

The computational core of our approach is Algorithm 2. We start its explanation from the steps in lines 3-5 (we will return to the condition in line 1 later on). Here we first generate node U' and then compute its distance from the root leveraging a fast access to the read-only list $L(X_i)$, as discussed in Section 3.1. Then, in line 6, we apply the optimal path extension such that U'' becomes the end of the optimal extension starting at U'. When applying OPE we update $U''.path$ and $U''.d$ accordingly. Because it is possible that an alternative path passing through U'' has been discovered in the previous iterations, or by different processing threads in the current iteration of StructureSearch, we update the search front in lines 7-12. The update depends on efficient execution of Find and Update operations over Q, which we discuss in the next section. In the case when the current node U'' does not improve over the one previously found, referenced by R in line 7, we simply end the process in lines 11-12. This is because applying our optimizations (OPE and SCC) will not provide further extensions to U'' – if existed, these extensions would had been discovered for node R in earlier iterations. Observe however that U'' will be processed in the subsequent iterations of Struc-

TURESEARCH to explore paths that may be optimal but not discoverable by OPE or SCC.

In the case when we have a new best path to U'' (i.e., condition in line 8 does not hold, or condition in line 9 holds) we try to further extend U'' by applying SCC decomposition. Specifically, we observe that finding an optimal path from U'' to the sink of the lattice $\mathcal{L}$ is equivalent to learning an optimal network over variables $\mathcal{X} - U''$ (this follows from the basic properties of the poset lattice [18]). Consequently, we can continue search from U'' by applying SCC decomposition over $G'_{\mathcal{X}-U''}$. We start by enumerating SCCs in line 13. We use the classic Tarjan algorithm which delivers SCCs in a topological order over the DAG formed by all SCCs in $G'_{\mathcal{X}-U''}$ [28]. Next, using any sequential solver denoted by OPTORDER (e.g., basic BFS search) we find optimal ordering of variables, and hence optimal structure, for each component S_k whose size is smaller than user-defined threshold τ (lines 15-17). The reason we put the threshold on the size of connected component is to avoid a situation where the cost of OPTORDER becomes prohibitive (recall that the cost is exponential in the size of the component). We note also that it is possible to make τ a function that changes as the algorithm progresses. If we encounter component whose size is above the threshold, we advance beyond that component (line 18) and mark the variables belonging to the remaining unprocessed components as suboptimal. This is done by adding them to the set referenced by U''.*subopt* in lines 19-20. Although at this stage we stop extending node U'', the variables marked as suboptimal can be safely ignored as possible extensions for node U'' in subsequent iterations, which we prove below. This in turn enables us to further constraint the search space, as expressed by condition in line 1 (in the actual implementation, the condition is tested before the task PROCESSNODE is even created). We note also, that by performing more extensive analysis of the DAG of SCCs (compared to testing only SCC size) we could potentially process more SCCs (hence extend U'' even more) and have tighter list U''.*subopt*. However, that would require more complex implementation with potentially significant performance overheads. The processing is finalized in lines 22-27 where we update the search front with newly discovered best path to U''.

Theorem 2. *Given input lists $L(X_i)$, Algorithms 1 and 2 are guaranteed to find an exact network structure for $\mathcal{X}$.*

Proof. Since our algorithms perform BFS search over the lattice $\mathcal{L}$ for $\mathcal{X}$, it is sufficient to show that the search space pruning in Algorithm 2 preserves at least one optimal path from the root to the sink of $\mathcal{L}$. The correctness of pruning by OPE in line 6 follows directly from Theorem 1. To show that pruning by condition in line 1 will preserve an optimal path, we observe the following. The SCC decomposition in lines 13-21 is guaranteed to find an optimal path from U'' to the sink of $\mathcal{L}$ as long as all components in S are smaller than τ (which means that condition in line 1 for U'' will not hold). From the property of SCCs we have that there is an optimal path from U'' to the sink in which all variables from component $S_k \in S$ precede all variables from component $S_l \in S$, for all $k < l$ in the topological order of SCCs returned by the Tarjan algorithm in line 13. Consequently, by advancing k in line 18 we can be certain that at least one optimal path from U'' will be directly passing via variables in S_k, and thus can postpone processing of the remaining variables as implemented in lines 19-21 and the condition in line 1. □

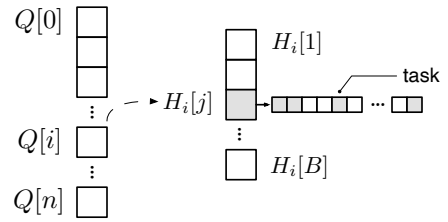

Fig. 3: High level overview of the data structure Q.

One final observation is that the SCC optimization involves balancing between computational overheads (due to internal SCC algorithms, e.g., TARJANSCC) and memory overheads (due to the nodes not pruned by SCC). In other words, by using SCC we reduce the size of the search space, but that reduction comes at the cost that may offset the resulting performance gains.

Maintaining Search Front

As mentioned earlier, the performance and correctness of our parallel BFS algorithm depends on Q, which is the data structure maintaining the search front. More precisely, the sequence of operations FIND-UPDATE-OR-INSERT in Q must correctly maintain information about shortest paths discovered while minimizing synchronization overheads. The correctness can be easily achieved by guarding the entire sequence of operations on Q, e.g., lines 22-27 in Algorithm 2, with a mutex lock. However, in such case Q becomes a contention point for the processing threads. At the same time, the existing fully concurrent hash tables do not support arbitrary types (critical to implementing BFS) or if they do, like `hash_map` in Intel TBB, they tend to be very slow [29].

To address this challenge we designed Q such that the tasks are partitioned across and within the layers of the search lattice. The memory organization of our data structure is outlined in Figure 3. Technically, Q is simply an array of arrays of hash tables. The array $Q[i]$ stores all tasks representing the search frontier for level i of the lattice (recall that level 0 is the root of the lattice). At the given level, the tasks are distributed into B individual hash tables, where each hash table $H_i[j]$ is guarded by a mutex. Here we use a high performance hash table implementation from the excellent `phmap` project [30] that is based on closed hashing with SIMD parallelism.

Spreading tasks within a layer across B hash tables allows for higher concurrency with modest additional memory use. However, to be effective this strategy requires a hashing function that will distribute lattice nodes between B hash tables such that the locking within table is limited. To achieve such function, we follow observations from [18], and exploit properties of the search space. Specifically, because the lattice $\mathcal{L}$ is an n-dimensional hypercube, we can partition it into d-dimensional hypercubes, $d < n$, by first assigning to each node U a bitmap that represents variables

in that node, and then considering a sub-bitmap of $n-d$ bits. Any node U that shares the same sub-bitmap belongs to the same d-dimensional hypercube out of the resulting 2^{n-d} hypercubes. For instance, in Figure 1a we use $d=2$ and mark nodes in the same partition with the same shade of gray. Our partitioning has the property, that for a node U in the partition its immediate successors are most likely in the same partition. Hence, when U is processed, and our optimizations (OPE and SCC) are effective, locking happens only within the same partition. In fact, if we consider all $i \cdot \binom{n}{i}$ incoming edges at level i of $\mathcal{L}$, then only $\binom{n-1}{i-1} \cdot (n-d)$ are across the partitions and may trigger locking collision.

The one final element is memory management in Q. As the search front in Algorithm 1 advances, the memory used by the processed lattice nodes can be released. This process is transparently handled by the data structure itself. When processing layer k a dedicated task releases memory taken by the array $Q[k-1]$. That task is handled together with the ProcessNode tasks in line 7 of Algorithm 1. It turns out that this simple optimization has significant performance implications, eliminating non-trivial sequential step in the execution.

5 Experimental Results

5.1 Software Implementation and Test Hardware

We implemented all proposed algorithms as part of the `SABNA` Bayesian Networks learning toolkit (ver. 3.010) [31]. `SABNA` is a C++17 software platform that offers an end-to-end Bayesian networks learning experience, including routines for input data preparation, Bayesian networks structure and parameter learning under various scoring criteria, as well as Bayesian networks post-processing. As such, it is a complete and user-oriented platform that supports shared memory systems.

To implement various scoring functions s, as required by the parent sets identification step, the framework relies on `SABNAtk` library [25]. This library provides efficient abstractions over the input data D, and optimizes evaluations of s and the underlying counting queries as required by our strategy in Section 4.1.

To implement the task parallelism components we used the Intel oneAPI TBB with its task and scheduling abstractions. To ensure portability and simplify reproducibility, we packaged the toolkit into a `Singularity` container, which was deployed on the target test server. The container definition relies on the `gcc-11` compiler and Intel oneAPI 2021.3, as provided in the Debian GNU/Linux. We make the container and its definition available together with our software [31] that is open source and distributed under the MIT license, and can be downloaded from `https://gitlab.com/SCoRe-Group/SABNA-Release`.

We run all our tests on a standalone server with four 13-core Intel Xeon 2.10GHz processors and 1.2TB of RAM.

5.2 Test Data

To perform the tests we used a set of standard benchmarks from the Bayesian Network Repository maintained by M. Scutari under the `bnlearn` project [32]. We summarize properties of our test data in Table 1, where $z = \sum_{X_i} |L(X_i)|$.

TABLE 1: Properties of the input data used in experiments.

Network	Variables, n	Observations, m	Parent sets, z
Alarm	37	12000	5839
Barley	48	12000	999
Hailfinder	56	1000	630
HEPAR II	70	350	306
Insurance	27	200000	34356
Mildew	35	160000	1856
Water	32	100000	1807
Win95pts	76	100	977

TABLE 2: Median runtime of parents search (in minutes).

	2	8	16	24	32	40	52
Alarm	117.95	31.44	16.42	11.00	9.25	8.10	6.72
Barley	5.76	1.54	0.80	0.53	0.45	0.38	0.32
Hailfinder	4.82	1.31	0.68	0.46	0.41	0.34	0.40
HEPAR II	40.75	11.67	6.92	5.22	4.64	4.15	3.95
Insurance	526.23	143.90	76.10	50.95	42.20	37.51	31.39
Mildew	148.01	39.68	20.60	13.68	11.34	10.02	8.39
Water	OT	OT	1612.02	1066.35	889.75	785.76	660.10
Win95pts	185.95	80.74	64.12	56.71	59.68	61.68	83.97

We focused on the *medium* and *large* networks, defined as networks with $25 \leq n \leq 80$ variables. These networks were used to simulate random samples, i.e., observations, delivering the actual input data for learning. The number of observations, m, sampled from each benchmark network was selected to cover a broad range of use scenarios, and to expose properties of both the parent sets identification and the structure search stages.

5.3 Scalability Tests

In the first set of experiments, we focused on the scalability of our method. To this end, we executed the maximal parents sets search and structure search steps using a varying number of cores. Each experiment was run five times to account for variability due to Intel TBB scheduling and work-stealing policies, and we report median runtime (we note that in both tests the standard deviation was under 3%). In all experiments, we used MDL scoring function. When running structure search we used $\tau = 6$ with all optimizations enabled, which we consider a meaningful default for large networks. Obtained results are summarized in Tables 2-3 and Figures 4-5.

We begin our discussion with the results for the maximal parent sets search. From Table 2 and Figure 4 we can see

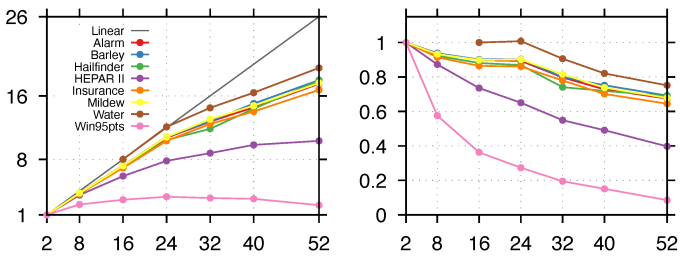

Fig. 4: Relative speedup (left), and the corresponding efficiency (right), of the maximal parent sets search on varying number of cores (x-axis). Please view in color.

TABLE 3: Median runtime of structure search (in minutes).

	2	8	16	24	32	40	52
Alarm	923.43	243.16	134.11	89.99	73.22	62.82	53.43
Barley	3902.50	1033.60	557.14	376.97	304.91	262.89	219.81
Hailfinder	OT	OT	2396.02	1676.77	1410.45	1178.19	979.93
HEPAR II	OT	2740.00	1438.04	959.26	769.91	661.24	549.21
Insurance	15.65	4.16	2.24	1.52	1.25	1.09	0.93
Mildew	260.91	68.69	36.76	25.44	20.59	18.70	15.08
Water	2.52	0.70	0.42	0.30	0.26	0.25	0.23
Win95pts	1353.08	359.20	187.85	126.25	101.98	86.98	73.27

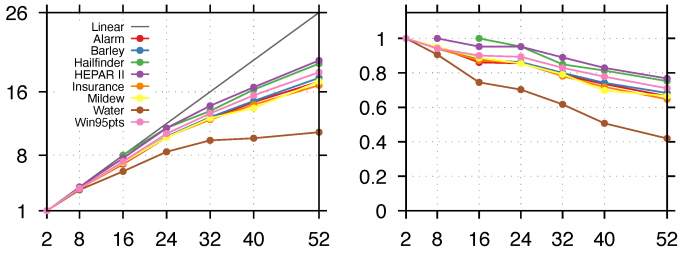

Fig. 5: Relative speedup (left), and the corresponding efficiency (right), of the structure search on varying number of cores (x-axis). Please view in color.

that for the majority of the test cases our software achieves over 60% parallel efficiency when using all 52 cores, and over 80% when using 24 cores. For two datasets, namely *HEPAR II* and *Win95pts*, the performance is significantly lower. This is explained by the inherent overheads in tasks management. Specifically, both datasets are characterized by a high number of variables n, and a small number of observations m. This implies a large number of very lightweight tasks to consider (recall Figure 2b). Consequently, even though TBB applies work stealing the internal cost of task maintenance becomes significant enough to impact scaling. This effect becomes less and less pronounced as we consider datasets with larger m. For instance, the efficiency of both *Insurance* and *Mildew* is close to 70% on 52 cores, despite significant differences in the runtime (*Insurance* is almost 4× more expensive to process than *Mildew*). Overall, the results show very good performance of our solution, and in some cases we are able to reduce the execution time from 6h to 30m (e.g., the *Insurance* dataset).

One interesting observation is with respect to the problem complexity in relation to the input data. As we can see from Table 2, for the *Water* dataset we are unable to identify maximal parent sets using fewer than 16 cores as the total execution time exceeds the maximum allowed of three days. This result is in line with earlier reports [9] showing that it is difficult to *a priori* assess how challenging is a given dataset. While *Water* is of moderate size it is definitely the most challenging for searching maximal parent sets (it takes over 11h to solve on 52 cores).

In Table 3 and Figure 5, we report performance results for the structure search. Again, our algorithms show excellent performance with efficiency over 60% when using 52 cores. One exception is the *Water* dataset. This set however takes only seconds to process and hence in this case it is hard to talk about scaling. Two datasets, *Hailfinder* and *HEPAR II*, cannot be solved unless more than two and more

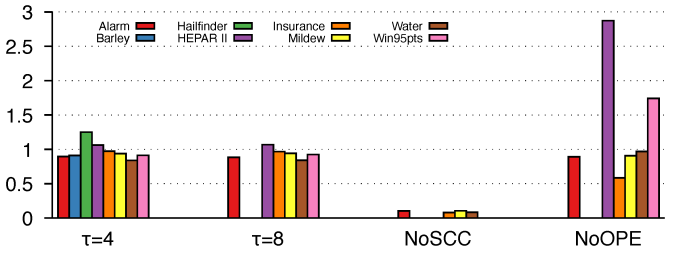

Fig. 6: Relative speed on 52 cores of the structure search depending on which search optimizations are enabled. The speed is computed with respect to the solver running with $\tau = 6$ and both OPE and SCC optimizations enabled. Please view in color.

than eight cores respectively are used (otherwise we run out of time). This further demonstrates the need for efficient parallel exact learning solvers.

To conclude the scaling analysis, we contrasted the performance of our SABNA software with GOBNILP ver. 1.6.3 [15]. Currently, there are no parallel BN exact learning tools available (see Section 6), and GOBNILP provides a sequential exact solver albeit using entirely different search strategy based on linear programming. Because of that, GOBNILP is not necessarily directly comparable to our solution (see [9] for the in-depth analysis). With that caveat in mind, when executed without constraints, GOBNILP was unable to learn any of the test networks within the three days time limit. When we constrained the number of parents each node may have in the network to six, i.e., we assumed that $Pa(X_i) \leq 6$, GOBNILP was able to learn networks for: *Alarm* in 73 minutes, *Insurance* in 418 minutes, and *Water* in 77 minutes. Here we should keep in mind that the resulting networks found by GOBNILP are not guaranteed to be optimal. The performance of GOBNILP seems to be primarily affected by how it handles the parent sets identification step, where it spent the majority of the compute cycles.

To summarize, the scaling results show very good performance of our proposed algorithms. They also demonstrate the need for efficient parallelization of both stages of the learning process. This is because different datasets pose different challenges for the learning process. For example, while we can enumerate maximal parent sets for *Hailfinder* in just few seconds, it takes over 16h to perform structure search. On the other hand, the maximal parent sets search takes hours for *Water*, yet the structure search can be done in seconds.

5.4 Impact of Optimizations

In the second set of experiments, we assessed how OPE and SCC optimizations affect the performance of the structure search. To this end, we executed the structure search for different parameters τ and with OPE and SCC components switched on and off. Results of these experiments are summarized in Figure 6 and Tables 4-5. We note that the fact that we enable or disable given optimizations does not influence the final network structure learned, i.e., in all cases we find optimal network structures. However, the optimizations do affect the final runtime.

TABLE 4: Search space reduction factor (RF) and runtime (T in minutes) for different τ.

	RF				T			
τ	4	5	6	7	4	5	6	7
Alarm	155	164	172	180	49	50	55	57
Barley	70941	83453	93198	$1 \cdot 10^5$	212	212	233	259
Hailfinder	$5.4 \cdot 10^6$	$7.1 \cdot 10^6$	$1.1 \cdot 10^7$	$8.7 \cdot 10^6$	803	818	642	1030
Hepar II	$1.8 \cdot 10^{11}$	$2.8 \cdot 10^{11}$	$3.7 \cdot 10^{11}$	$4.6 \cdot 10^{11}$	543	544	511	572
Insurance	6	6	6	6	1	1	1	1
Mildew	112	112	113	114	15	14	16	16
Water	900	929	960	1001	1	1	1	1
Win95pts	$1.8 \cdot 10^{14}$	$2.3 \cdot 10^{14}$	$2.4 \cdot 10^{14}$	$2.6 \cdot 10^{14}$	64	68	70	77

TABLE 5: Search space reduction factor for OPE and SCC.

	NoSCC	NoOPE	OPE + SCC
Alarm	81.42	143.32	172.31
Barley	OM	OM	1.6457×10^6
Hailfinder	OM	OM	1.1249×10^7
Hepar II	OM	116.35×10^9	371.05×10^9
Insurance	5.49	5.48	6.04
Mildew	63.36	88.72	113.41
Water	664.24	860.99	960.60
Win95pts	OM	146.05×10^{12}	249.38×10^{12}

The first observation we make is with respect to the parameter τ, which controls the size of the connected components considered by the SCC optimization. From Figure 6 and Table 4, we can see that changing τ to $\tau = 4$ (from the default $\tau = 6$) has somewhat limited impact on the performance: for some datasets, e.g., *Barley*, we observe a slight improvement, and for some others, e.g., *Hailfinder*, we see a slowdown. On the other hand, increasing τ to $\tau = 8$ becomes problematic for some datasets. Specifically, we are unable to process *Barley* and *Hailfinder* (the structure search runs out of time). This is because considering larger components in the SCC phase is too computationally expensive compared to the gains it offers by reducing the size of the search space. Indeed, Table 4 shows how runtime and the space reduction are affected by the choice of τ. Here the space reduction factor is the ratio between the total number of 2^n nodes in the dynamic programming lattice and the actual number of nodes visited by the search algorithm with given optimization enabled. As τ increases so does the space reduction factor, with a notable exception of *Insurance* and *Mildew* networks. The gains in the space reduction are especially visible for large networks, however, this does not translate into runtime reduction. This means that by increasing τ we are able to reduce the memory footprint of the search but at the expense of extra computational work. This issue is further highlighted if we consider the effect of disabling SCC entirely (NoSCC in Figure 6). Although disabling SCC significantly improves the performance (by eliminating the cost of SCC decomposition), this is not entirely practical. Specifically, with SCC disabled we are unable to process four out of the eight test datasets as our solver runs out of memory. This shows that reducing the search space via SCC may be necessary in some cases, even if it comes at the significant computational overhead.

Table 5 offers further insights into the performance of both optimization techniques. It shows that SCC and OPE provide the space reduction at the level of up to 10^{12}. For the majority of the datasets SCC results with somewhat higher space reduction factor than OPE. On the other hand, the results suggest that OPE is indeed inexpensive while still offering significant space reduction. Moreover, OPE remains essential: when it is disabled (NoOPE in Figure 6), we are again unable to process *Barley* and *Hailfinder*, and the cost of processing the largest datasets increases more than $1.5\times$. Overall, these results show that both optimizations should be considered in tandem as none is sufficient to handle all the test cases on its own. Here we should clarify that using a naive exhaustive search, i.e., with both SCC and OPE disabled, we are unable to process any of the datasets.

6 Related Work

There is a significant body of work on parallelizing Bayesian networks learning, including for distributed and shared memory systems, as well as for GPGPU accelerators. However, the existing methods either focus on heuristics or ignore properties of the search space thus they remain very limited.

One of the early attempts at exact parallel algorithms are due to Tamada et al. [33] and Nikolova et al. [18]. Both these works start with the dynamic programming formulation that considers recursions in Eqs. (1) and (2) together, and they target distributed memory clusters. Because of that, they do not introduce any techniques to constrain the search space. The resulting algorithms, while work-efficient under the MPI model, still have to consider all 2^n nodes in the dynamic programming lattice. Consequently, the resulting methods although theoretically elegant are of very limited practical application.

In [34], Gao et al. describe algorithm that decomposes the learning process into learning of sub-networks that are next combined into the final target network. Because each sub-network can be learned independently, the algorithm exploits the resulting embarrassing parallelism. However, although each sub-network can be learned using optimal algorithms, the final network cannot be guaranteed to be optimal. Hence, the method remains heuristic. Interestingly, Nikolova et al. proposed a similar idea in [35] (both works differ in how the decomposition is achieved).

To conclude the review, we should note also that there are many publications focusing on Bayesian networks learning based on the PC algorithm, for example [36], [37], [38]. However, the PC algorithm belongs to the class of so called constraint-based learning methods, which are another set of strategies that complement but do not substitute the score-based techniques we consider in the paper. The PC algorithms provide no guarantees with respect to the learning process and as such they are considered heuristics. Finally, these solutions are tailored for continuous data and exploit numerical properties of the underlying conditional independence tests. While that makes them relatively easier to parallelize, they still do not guarantee any optimality. Because of that they are not directly comparable to our score-based formulation.

7 Conclusion

Bayesian networks are an important tool for reasoning and data analytics. Their key advantage is interpretability –

both the model graph, which describes conditional independencies, and the model parameters, which quantify conditional probabilities, have clear and intuitive interpretation. However, the extent to which the interpretation is viable depends on the quality of the model itself.

In this paper, we presented a set of parallel computing techniques that push the scale of exact learning of BNs. Since exact learning eliminates model uncertainty due to structure search, it effectively improves quality of the learned models and hence contributes to their better interpretability. High quality BN models are of the essence, for example in biomedicine or fraud detection, and our solvers are directly applicable in such situations. Our solutions are packaged into an ease-to-deploy and open source software with a permissive license, and that should simplify their broader adoption. Furthermore, because our methods advance both parent sets identification and structure search, they can be piecemeal combined to aid the future development of new solvers.

Despite the improvements we report in this paper, we believe that the problem of exact learning is far from solved. For example, many of the test networks we considered in our experiments become intractable (e.g., due to memory or time constraints) if the number of observations is increased. Moreover, as reported in [9], depending on the input data, some optimizations may be more effective than other, yet the choice of the best set of optimizations is not obvious. In our opinion, all this necessitates further research especially to improve the search space constraining during the structure search. We hope that by establishing a new baseline for exact learning, our results will contribute to further advancements in exact learning.

Acknowledgments

This research is supported by the NSF under the award OAC-1845840. We wish to acknowledge technical support provided by the Center for Computational Research at the University at Buffalo.

References

[1] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[2] J. Pearl and D. Mackenzie, *The Book of Why: The New Science of Cause and Effect*, 1st ed. Basic Books, 2018.

[3] J. Pearl, *Causality: Models, Reasoning and Inference*, 2nd ed. Cambridge University Press, 2009.

[4] J. Yu, V. Smith, P. Wang, A. Hartemink, and E. Jarvis, "Advances to Bayesian network inference for generating causal networks from observational biological data," *Bioinformatics*, vol. 20, no. 18, pp. 3594–3603, 2004.

[5] P. Lucas, L. van der Gaag, and A. Abu-Hanna, "Bayesian networks in biomedicine and health-care," *Artificial Intelligence in Medicine*, vol. 30, no. 3, pp. 201–214, 2004.

[6] L. Mukhanov, "Using Bayesian belief networks for credit card fraud detection," in *International Conference on Artificial Intelligence and Applications*, 2008, pp. 221–225.

[7] R. Rohekar, S. Nisimov, Y. Gurwicz, G. Koren, and G. Novik, "Constructing deep neural networks by Bayesian network structure learning," in *Neural Information Processing Systems*, 2018, p. 12.

[8] D. Chickering, *Learning from Data: Artificial Intelligence and Statistics V*. Springer-Verlag, 1996, ch. Learning Bayesian Networks is NP-Complete, pp. 121–130.

[9] B. Malone, K. Kangas, M. Jarvisalo, M. Koivisto, and P. Myllymaki, "Empirical hardness of finding optimal Bayesian network structures: algorithm selection and runtime prediction," *Machine Learning*, vol. 107, no. 1, 2018.

[10] G. Schwarz, "Estimating the dimension of a model," *The Annals of Statistics*, vol. 6, pp. 461–464, 1978.

[11] H. Akaike, "Information theory and an extension of the maximum likelihood principle," in *Second International Symposium on Information Theory*, 1973, pp. 267–281.

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

[13] V. Rodionov, "On the number of labeled acyclic digraphs," *Discrete Mathemathics*, vol. 105, no. 1-3, pp. 319–321, 1992.

[14] M. Koivisto and K. Sood, "Exact Bayesian structure discovery in Bayesian networks," *Journal of Machine Learning Research*, vol. 5, pp. 549–573, 2004.

[15] J. Cussens, M. Jarvisalo, J. Korhonen, and M. Bartlett, "Bayesian network structure learning with integer programming: Polytopes, facets and complexity," *Journal of Artificial Intelligence Research*, vol. 58, no. 1, pp. 185–229, 2017.

[16] C. Yuan, B. Malone, and X. Wu, "Learning optimal Bayesian networks using A* search," in *International Joint Conference on Artificial Intelligence*, 2011, pp. 2186–2191.

[17] S. Ott, S. Imoto, and S. Miyano, "Finding optimal models for small gene networks," in *Pacific Symposium on Biocomputing*, 2004, pp. 557–567.

[18] O. Nikolova, J. Zola, and S. Aluru, "Parallel globally optimal structure learning of Bayesian networks," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1039–1048, 2013.

[19] A. Singh and A. Moore, "Finding optimal Bayesian networks by dynamic programming," Carnegie Mellon University, Tech. Rep., 2005.

[20] M. Koivisto, "Parent assignment is hard for the MDL, AIC, and NML costs," in *International Conference on Computational Learning Theory*, 2006, pp. 289–303.

[21] S. Karan and J. Zola, "Exact structure learning of Bayesian networks by optimal path extension," in *IEEE International Conference on Big Data*, 2016, pp. 48–55.

[22] C. de Campos and Q. Ji, "Efficient structure learning of Bayesian networks using constraints," *Journal of Machine Learning Research*, pp. 663–689, 2011.

[23] A. Correia, J. Cussens, and C. de Campos, "On pruning for score-based Bayesian network structure learning," in *International Conference on Artificial Intelligence and Statistics*, 2020, p. 9.

[24] S. Karan and J. Zola, "Scalable exact parent sets identification in Bayesian networks learning with Apache Spark," in *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2017, pp. 33–41.

[25] S. Karan, M. Eichhorn, B. Hurlburt, G. Iraci, and J. Zola, "Fast counting in machine learning applications," in *Uncertainty in Artificial Intelligence (UAI)*, 2018.

[26] X. Fan, B. Malone, and C. Yuan, "Finding optimal Bayesian network structures with constraints learned from data," in *Uncertainty in Artificial Intelligence (UAI)*, 2014, p. 10.

[27] J. Schreiber and W. S. Noble, "Finding the optimal Bayesian network given a constraint graph," *PeerJ Computer Science*, vol. 3, 2017.

[28] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[29] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general(?)!" *ACM Transactions on Parallel Computing*, vol. 5, no. 4, pp. 16:1–16:32, 2019.

[30] G. Popovitch, "The Parallel Hashmap," https://github.com/greg7mdp/parallel-hashmap, 2022.

[31] "SABNA: Scalable accelerated bayesian network analytics," https://gitlab.com/SCoRe-Group/SABNA-Release.

[32] "bnlearn – Bayesian network structure learning," http://www.bnlearn.com/.

[33] Y. Tamada, S. Imoto, and S. Miyano, "Parallel algorithm for learning optimal Bayesian network structure," *Journal of Machine Learning Research*, vol. 12, pp. 2437–2459, 2011.

[34] T. Gao and D. Wei, "Parallel Bayesian network structure learning," in *International Conference on Machine Learning (ICML)*, 2018, pp. 1685–1694.

[35] O. Nikolova and S. Aluru, "Parallel Bayesian network structure learning with application to gene networks," in *International Con-*

ference on High Performance Computing, Networking, Storage and Analysis (SC), 2012, pp. 1–9.

[36] A. Madsen, F. Jensen, A. Salmeron, H. Langseth, and T. Nielsen, "A parallel algorithm for Bayesian network structure learning from large data sets," *Knowledge-Based Systems*, vol. 117, pp. 46–55, 2017.

[37] T. Le, T. Hoang, J. Li, L. Liu, H. Liu, and S. Hu, "A fast PC algorithm for high dimensional causal discovery with multi-core PCs," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 16, no. 5, 2019.

[38] B. Zarebavani, F. Jafarinejad, M. Hashemi, and S. Salehkaleybar, "cuPC: CUDA-based parallel PC algorithm for causal structure learning on GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 530–542, 2020.

Subhadeep Karan received his BSc degree and MEng degree in information technology from the University of Mumbai and IIIT-Allahabad, India, in 2010 and 213, respectively. He received his PhD in computer science from the University at Buffalo in 2019. His research interests span high performance computing and machine learning. Post PhD, he has been working for Google, and Meta (Facebook), primarily in the machine learning infrastructure space.

Zainul Abideen Sayed received his BSc degree in computer engineering from the University of Mumbai, India, in 2016. He is currently pursuing a PhD degree in computer science at the University at Buffalo. His research interests are in high performance computing, parallel runtime systems, and graph algorithms.

Jaroslaw Zola received MSc degree in computer science from the Czestochowa University of Technology, Poland, in 2001, and PhD in computer science from the Grenoble Institute of Technology, France, in 2005. His research interests are in applications of high performance and mobile computing techniques in biomedical applications, machine learning and scientific computing. He is an Associate Professor in the Department of Computer Science and Engineering at the University at Buffalo. He is a Senior Member of IEEE and serves as an Associate Editor for the IEEE Transactions on Parallel and Distributed Systems.