Journal of Machine Learning Research 12 (2011) 2437-2459 Submitted 7/10; Revised 2/11; Published 7/11

Parallel Algorithm for Learning Optimal Bayesian Network Structure

Yoshinori Tamada[*] TAMADA@IMS.U-TOKYO.AC.JP
Seiya Imoto IMOTO@IMS.U-TOKYO.AC.JP
Satoru Miyano[†] MIYANO@IMS.U-TOKYO.AC.JP
Human Genome Center
Institute of Medical Science, The University of Tokyo
4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan

Editor: Russ Greiner

Abstract

We present a parallel algorithm for the score-based optimal structure search of Bayesian networks. This algorithm is based on a dynamic programming (DP) algorithm having $O(n \cdot 2^n)$ time and space complexity, which is known to be the fastest algorithm for the optimal structure search of networks with n nodes. The bottleneck of the problem is the memory requirement, and therefore, the algorithm is currently applicable for up to a few tens of nodes. While the recently proposed algorithm overcomes this limitation by a space-time trade-off, our proposed algorithm realizes direct parallelization of the original DP algorithm with $O(n^\sigma)$ time and space overhead calculations, where $\sigma > 0$ controls the communication-space trade-off. The overall time and space complexity is $O(n^{\sigma+1}2^n)$. This algorithm splits the search space so that the required communication between independent calculations is minimal. Because of this advantage, our algorithm can run on distributed memory supercomputers. Through computational experiments, we confirmed that our algorithm can run in parallel using up to 256 processors with a parallelization efficiency of 0.74, compared to the original DP algorithm with a single processor. We also demonstrate optimal structure search for a 32-node network without any constraints, which is the largest network search presented in literature.

Keywords: optimal Bayesian network structure, parallel algorithm

1. Introduction

A Bayesian network represents conditional dependencies among random variables via a directed acyclic graph (DAG). Several methods can be used to construct a DAG structure from observed data, such as score-based structure search (Heckerman et al., 1995; Friedman et al., 2000; Imoto et al., 2002), statistical hypothesis testing-based structure search (Pearl, 1988), and a hybrid of these two methods (Tsamardinos et al., 2006). In this paper, we focus on a score-based learning algorithm and formalize it as a problem to search for an optimal structure that derives the maximal (or minimal) score using a score function defined on a structure with respect to an observed data set. A score function has to be decomposed as the sum of the local score functions for each node in a network. In general, posterior probability-based score functions derived from Bayesian statistics are used. The optimal score-based structure search of Bayesian networks is known to be an NP-hard

∗. Currently at Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo. 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan. tamada@is.s.u-tokyo.ac.jp.
†. Also at the Computational Science Research Program, RIKEN, 2-1 Hirosawa, Wako, Saitama 351-0198, Japan.

problem (Chickering et al., 1995). Several efficient dynamic programming (DP) algorithms have been proposed to solve such problems (Ott et al., 2004; Koivisto and Sood, 2004). Such algorithms have $O(n \cdot 2^n)$ time and space complexity, where n is the number of nodes in the network. When these algorithms are applied to real problems, the main bottleneck is bound to be the memory space requirement rather than the time requirement, because these algorithms need to store the intermediate optimal structures of all combinations of node subsets during DP steps. Because of this limitation, such algorithm can be applied to networks of only up to around 25 nodes in a typical desktop computer. Thus far, the maximum number of nodes treated in an optimal search without any constraints is 29 (Silander and Myllymäki, 2006). This was realized by using a 100 GB external hard disk drive as the memory space instead of using the internal memory, which is currently limited to only up to several tens of GB and in a typical desktop computer.

To overcome the above mentioned limitation, Perrier et al. (2008) proposed an algorithm to reduce the search space using structural constraints. Their algorithm searches for the optimal structure on a given predefined super-structure, which is often available in actual problems. However, it is still important to search for a globally optimal structure because Bayesian networks find a wide range of applications. As another approach to overcome the limitation, Parviainen and Koivisto (2009) proposed a space-time trade-off algorithm that can search for a globally optimal structure with less space. From empirical results for a partial sub-problem, they showed that their algorithm is computationally feasible for up to 31 nodes. They also mentioned that their algorithm can be easily parallelized with up to 2^p processors, where $p = 0, 1, \ldots, n/2$ is a parameter that is used to control the space-time trade-off. Using parallelization, they suggested that it might be possible to search larger-scale network structures using their algorithm. The time and space complexities of their algorithm are $O(n \cdot 2^n (3/2)^p)$ and $O(n \cdot 2^n (3/4)^p)$, respectively.

Of course, the memory space limitation can be overcome by simply using a computer with sufficient memory. The DP algorithm remains computationally feasible even for a 29-node network in terms of the time requirement. For such a purpose, a supercomputer with shared memory is required. Modern supercomputers can be equipped with several terabytes of memory space. Therefore, they can be used to search larger networks than the current 29-node network that requires 100 GB of memory. However, such supercomputers are typically very expensive and are not scalable in terms of the memory size and the number of processors. In contrast, massively parallel computers, a much cheaper type of supercomputers, make use of distributed memory; in such systems many independent computers or computation nodes are combined and linked through high-speed connections. This type of supercomputers is less expensive, as mentioned, and it is scalable in terms of both the memory space and the number of processors. However the DP algorithm cannot be executed on such a distributed memory computer because it requires memory access to be performed over a wide region of data, and splitting the search space across distributed processors and storing the intermediate results in the distributed memory are not trivial problems.

Here, we present a parallelized optimal Bayesian network search algorithm called *Para-OS*. The proposed algorithm is based on the OS algorithm using DP that was proposed by Ott et al. (2004). Our algorithm realizes direct parallelization of the DP steps in the original algorithm by splitting the search space of DP with $O(n^\sigma)$ time and space overhead calculations, where $\sigma = 1, 2, \ldots > 0$ is a parameter that is used to split the search space and controls the trade-off between the number of communications (not the volume of communication) and the memory space requirement. The main feature of this algorithm is that it guarantees that the amount of intermediate results that are required to be shared redundantly among independently split calculations is minimal. In other words,

our algorithm guarantees that minimal communications are required between independent parallel processors. Because of this advantage, our algorithm can be easily parallelized, and, in practice, it can run very efficiently on massively parallel computers with several hundreds of processors. Another important feature of our algorithm is that it calculates no redundant score functions, and it can distribute the entire calculation almost equally across all processors. The main operation in our algorithm is the calculation of the score function. In practice, this feature is important to actually search for the optimal structure. The overall time and space complexity is $O(n^{\sigma+1}2^n)$. Our algorithm adopts an approach opposite to that of Parviainen and Koivisto (2009) to overcome the bottleneck of the memory space problem. Although our algorithm has slightly greater space and time complexties, it makes it possible to realize large-scale optimal network search in practice using widely available low-cost supercomputers.

Through computational experiments, we show that our algorithm is applicable to large-scale optimal structure search. First, the scalability of the proposed algorithm to the number of processors is evaluated through computational experiments with simulated data. We confirmed that the program can run efficiently in parallel using up to 256 processors (CPU cores) with a parallelization efficiency of more than 0.74 on a current supercomputer system, and acceptably using up to 512 processors with a parallelization efficiency of 0.59. Finally, we demonstrate the largest optimal Bayesian network search attempted thus far on a 32-node network with 256 processors using our proposed algorithm without any constraints and without an external hard disk drive. Our algorithm was found to complete the optimal search including the score calculation within a week.

The remainder of this paper is organized as follows. Section 2 presents an overview of the Bayesian network and the optimal search algorithm, which serves as the basis for our proposed algorithm. Section 3 describes the parallel optimal search algorithm in detail. Section 4 describes the computational experiments used for evaluating our proposed algorithm and presents the obtained results. Section 5 concludes the paper with a brief discussion. The appendix contains some proofs and corollaries related to those described in the main paper.

2. Preliminaries

In this section, we first present a brief introduction to the Bayesian network model, and then, we describe the optimal search (OS) algorithm, which is the basal algorithm that we parallelize in the proposed algorithm.

2.1 Bayesian Network

A Bayesian network is a graphical model that is used to represent a joint probability of random variables. By assuming the conditional independencies among variables, the joint probability of all the variables can be represented by the simple product of the conditional probabilities. These independencies can be represented via a directed acyclic graph (DAG). In a DAG, each node corresponds to a variable and a directed edge, to the conditional dependencies among variables or to the independencies from other variables. Suppose that we have n random variables, $V = \{X_1, X_2, \ldots, X_n\}$. The joint probability of variables in V is represented as

$$P(X_1, X_2, \ldots, X_n) = \prod_{j=1}^{n} P(X_j | Pa^G(X_j)),$$

where $Pa^G(X_j)$ represents the set of variables that are direct parents of the j-th variable X_j in network structure G and $P(X_j|Pa^G(X_j))$, a conditional probability for variable X_j.

A score-based Bayesian network structure search or a Bayesian network estimation problem is to search for the DAG structure fitted to the observed data, in which the fitness of the structure to the given data is measured by a score function. The score function is defined on a node and its parent set. The scores of nodes obtained by a score function are called local scores. A network score is defined simply as the sum of local scores of all nodes in a network. Using a score function, the Bayesian network structure search can be defined as a problem to find a network structure $\hat{G}$ that satisfies the following equation:

$$\hat{G} = \arg\min_{G} \sum_{j=1}^{n} s(X_j, Pa^G(X_j), X),$$

where $s(X_j, Pa^G(X_j), X)$ is a score function $s : V \times 2^V \times \mathbb{R}^{N,n} \to \mathbb{R}$ for node X_j given the observed input data of an $(N \times n)$-matrix X, where N is the number of observed samples.

2.2 Optimal Search Algorithm using Dynamic Programming

Next, we briefly introduce the OS algorithm using DP proposed by Ott et al. (2004). Our proposed algorithm is a parallelized version of this algorithm. We employ score functions described as in the original paper by Ott et al. (2004). That is, a smaller score represents better fitting of the model. Therefore, the problem becomes one of finding the structure that minimizes the score function. The optimal network structure search by DP can be regarded as an optimal permutation search problem. The algorithm consists of two-layer DP: one for obtaining the optimal choice of the parent set for each node and one for obtaining the optimal permutation of nodes. First, we introduce some definitions.

Definition 1 (Optimal local score) *We define the function $F : V \times 2^V \to \mathbb{R}$ as*

$$F(v,A) \stackrel{\text{def}}{=} \min_{B \subset A} s(v,B,X).$$

That is, $F(v,A)$ calculates the optimal choice of the parent set from A for node v and returns its optimal local score. $B \subset A$ represents the actual optimal choice for v, and generally, we also need to include it in the algorithm along with the score, in order to reconstruct the network structure later.

Definition 2 (Optimal network score on a permutation) *Let $\pi : \{1,2,\ldots,|A|\} \to A$ be a permutation on $A \subset V$ and Π^A be a set of all the permutations on A. Given a permutation $\pi \in \Pi^A$, the optimal network score on π can be described as*

$$Q^A(\pi) \stackrel{\text{def}}{=} \sum_{v \in A} F(v, \{u \in A : \pi^{-1}(u) < \pi^{-1}(v)\}).$$

Definition 3 (Optimal network score) *By using $Q^A(\pi)$ defined above, we can formalize the network structure search as a problem to find the optimal permutation that gives the minimal network score:*

$$M(A) \stackrel{\text{def}}{=} \arg\min_{\pi \in \Pi^A} Q^A(\pi).$$

Here, $M(A)$ represents the optimal permutation that derives the minimal score of the network consisting of nodes in A.

Finally, the following theorem provides an algorithm to calculate $F(v,A)$, $M(A)$, and $Q^A(M(A))$ by DP. See Ott et al. (2004) for the proof of this theorem.

Theorem 4 (Optimal network search by DP) *The functions $F(v,A)$, $M(A)$, and $Q^A(M(A))$ defined above can be respectively calculated by the following recursive formulae:*

$$F(v,A) = \min\{s(v,A,X), \min_{a\in A} F(v,A\setminus\{a\})\}, \tag{1}$$

$$M(A)(i) = \begin{cases} M(A\setminus\{v^*\})(i) & (i < |A|) \\ v^* & (i = |A|) \end{cases}, \tag{2}$$

$$Q^A(M(A)) = F(v^*, A\setminus\{v^*\}) + Q^{A\setminus\{v^*\}}(M(A\setminus\{v^*\})), \tag{3}$$

where

$$v^* = \arg\min_{v\in A}\{F(v,A\setminus\{v\}) + Q^{A\setminus\{v\}}(M(A\setminus\{v\}))\}.$$

By applying the above equations from $|A| = 0$ to $|A| = |V|$, we obtain the optimal permutation π on V and its score $Q^V(M(V))$ in $O(n\cdot 2^n)$ steps.

Note that in order to reconstruct the network structure, we need to keep the optimal choice of the parent set derived in Equation (1) and the optimal permutation $\pi = M(A)$ in Equation (3) for all the combinations of $A \subset V$ in an iterative loop for the next size of A.

3. Parallel Optimal Search Algorithm

The key to parallelizing the calculation of the optimal search algorithm by DP is splitting all the combinations of nodes in a single loop of DP for $F(v,A)$, $M(A)$, and $Q^A(M(A))$, given above by Equations (1), (2), and (3), respectively. Simultaneously, we need to consider how to reduce the amount of information that needs to be exchanged between processors. In the calculation of $M(A)$, we need to obtain all the results of $M(\cdot)$ for one-smaller subsets of A at hand, that is, $M(A\setminus\{a\})$ for all $a \in A$. Suppose that such $M(A\setminus\{a\})$'s are stored in the distributed memory space, and we have collected them for calculating $M(A)$. In order to reduce the number of communications, it would be better if we can re-use the collected results for another calculation. For example, we can calculate $M(B)$ ($|B| = |A|$) in the same processor that calculates $M(A)$ such that some of $M(B\setminus\{b\})$ ($b \in B$) *overlaps* $M(A\setminus\{a\})$. That is, if we can collect the maximal number of $M(X)$'s for any X such that $|X| = |A| - 1 \wedge X \subset \{(A\setminus\{a\}) \cap (B\setminus\{b\})\}$, then the number of communications required for calculating $M(A)$ and $M(B)$ can be minimized. Theorem 7 shows how we generate such a set of combinations, and we prove that it provides the optimally minimal choice of such combinations by allowing some redundant calculations.

In addition, as is evident from Equations (1), (2), and (3), the DP algorithm basically consists of simply searching for the best choice from the candidates that derive the minimal score. Time is mainly required to calculate the score function $s(v,A,X)$ for all the nodes and their parent combinations. Thus, our algorithm calculates $s(v,A,X)$ equally in independent processors without any redundant calculations for this part.

In this section, we first describe some basic definitions, and then, we present proofs of theorems that the proposed algorithm relies on. Finally, we present the proposed parallel algorithm.

3.1 Separation of Combinations

First, we define the combination, sub-combination, and super-combination of nodes.

Definition 5 (Combination) *In this paper, we refer to a set of nodes in V as a combination of nodes. We also assume a combination of k nodes, that is, $X = \{x_1, x_2, \ldots, x_k\} \subset V$ such that $ord(x_i) < ord(x_j)$ if $i < j$, where $ord : V \to \mathbb{N}$ is a function that returns the index of element $v \in V$. For example, suppose that $V = \{a, b, c, d\}$. $ord(a) = 1$ and $ord(d) = 4$.*

Definition 6 (Sub-/super-combination) *We define C' as a sub-combination of some combination C if $C' \subset C$, and C is a super-combination of C'. We say that a super-combination C is generated from C' if C is a super-combination of C'. In addition, we say that sub-combination C' is derived from C if C' is a sub-combination of C. For the sake of convenience, if we do not mention about the size of a sub-/super-combination of a combination, then we assume that it refers to a one-size smaller/larger sub-/super-combination. In addition, we say that two combinations A and B share sub-combinations if $\mathcal{A}' \cap \mathcal{B}' \neq \emptyset$, where $\mathcal{A}'$ and $\mathcal{B}'$ are sets of all the sub-combinations of A and B, respectively.*

We present two theorems that our algorithm relies on along with their proofs. These two theorems are used to split the calculation of $F(v,A)$, $M(A)$, and $Q^A(M(A))$; all these require the results for their sub-combinations. We show that the calculation can be split by the super-combination of A, and it is the optimal separation of the combinations in terms of the number of communications required.

Theorem 7 (Minimality of required sub-combinations) *Let $\mathcal{A}$ be a set of combinations of nodes in V, where $|A| = k > 0$ for $A \in \mathcal{A}$ and $|V| = n$. If $|\mathcal{A}| = \binom{k+\sigma}{k}$ ($\sigma > 0 \wedge \sigma + k \leq n$), then the minimal number of sub-combinations of length $k-1$ required to generate all the combinations in $\mathcal{A}$ is $\binom{k+\sigma}{k-1}$. Let S be a combination of nodes, where $|S| = k+\sigma$. We can generate a set of combinations of length k that satisfies the former condition by deriving all the sub-combinations of length k from S.*

Proof Because $\mathcal{A}$ contains $\binom{k+\sigma}{k}$ combinations of length k, the number of distinct elements (nodes) involved in $\mathcal{A}$ is $k+\sigma$ and is minimal. Therefore, the number of sub-combinations required to derive $\binom{k+\sigma}{k}$ combinations of length k in $\mathcal{A}$ is equal to the number of possible combinations that can be generated from $k+\sigma$ elements, and is $\binom{k+\sigma}{k-1}$. No more combinations can be generated from $\binom{k+\sigma}{k-1}$ sub-combinations. Therefore, it is the minimal number of required sub-combinations required to generate $\binom{k+\sigma}{k}$ combinations. A super-combination S of length $k+\sigma$ contains $k+\sigma$ elements and can derive $\binom{k+\sigma}{k}$ combinations of length k. Therefore, S can derive $\mathcal{A}$, and all the combinations of length $k-1$ derived from elements in S are the sub-combinations required to generate combinations in $\mathcal{A}$. ■

Theorem 8 (Minimality of required super-combinations) *The minimal number of super-combinations of length $k+\sigma$ from V that is required to generate all the sub-combinations of size k is $\binom{n-\sigma}{k}$.*

Proof Consider the set $T = V \setminus \{v_1, v_2, \ldots, v_\sigma\}$, where any $v_i \in V$, and thus, $|T| = n - \sigma$. If we generate all the combinations of length k taken from T, then these include all the combinations of length k from nodes in V without $v_1, \ldots, v_\sigma$, and the number of combinations is $\binom{n-\sigma}{k}$. Consider a set of combinations $\mathcal{S} = \{\{v_1, \ldots, v_\sigma\} \cup T' : T' \subset T \wedge |T'| = k\}$. Here, $|S| = k + \sigma$ for $S \in \mathcal{S}$ and $|\mathcal{S}| = \binom{n-\sigma}{k}$. Because $\mathcal{S}$ contains all the combinations of length k without $v_1, \ldots, v_\sigma$ and all the elements in $\mathcal{S}$ contain $v_1, \ldots, v_\sigma$, we can generate all the combinations in V of length k from some $S \in \mathcal{S}$ by combining $0 \leq \alpha \leq k$ nodes from $v_1, \ldots, v_\sigma$ and $k - \alpha$ nodes from $S \setminus \{v_1, \ldots, v_\sigma\}$. If we remove any $S \in \mathcal{S}$ from it, then there exist combinations that cannot be generated from another $S \in \mathcal{S}$ because $\mathcal{S}$ lists all the combinations except for nodes $v_1, \ldots, v_\sigma$. Therefore, $\mathcal{S}$ is the minimal set of super-combinations required to derive all the combinations of length k from V and its size is $|\mathcal{S}| = \binom{n-\sigma}{k}$. We can generate $\mathcal{S}$ by taking the first $\binom{n-\sigma}{k}$ combinations from $\binom{n}{k}$ combinations arranged in lexicographical order. ■

Theorem 7 can be used to split the search space of DP using by *super-combinations*, and Theorem 8 provides the number of super-combinations required in the parallel computation for a certain size of A for $M(A)$. From these two theorems, we can easily derive the following corollaries.

Corollary 9 (Optimal separation of combination) *The DP steps used to calculate $M(A)$ and $Q^A(M(A))$ in the OS algorithm can be split into $\binom{n-\sigma}{k}$ portions by super-combinations of A with length $k+\sigma$, where $|A| = k$. The size of each split problem is $\binom{k+\sigma}{k}$ and the number of required $M(B)$ for $B \subset A \wedge |B| = k-1$ is $\binom{k+\sigma}{k-1}$, which is the minimal number for $\binom{k+\sigma}{k}$ combinations of $M(A)$. Here, B is a set of sub-combinations of A. The calculation of $F(v,A)$ can also be split based on the sub-combinations B for $M(A)$.*

The separation of $M(A)$ by super-combinations causes some redundant calculations. The following corollary gives the amount of such overhead calculations and the overall complexity of the algorithm.

Corollary 10 (Amount of redundant calculations) *If we split the calculations of $M(A)$ ($|A| = k$) using super-combinations of size $k+\sigma$ ($\sigma > 0$), then the number of calculations of $M(A)$ for all $A \subset V$ is $\binom{n-\sigma}{k} \cdot \binom{k+\sigma}{k} = \binom{n}{k} O(n^\sigma)$. Thus, as compared to the original DP steps $\binom{n}{k}$, the overhead increment of the calculations for $M(A), Q^A(M(A))$, and $F(v,A)$ is at most $O(n^\sigma)$. The memory requirement to store the intermediate results is also dependent on the size of the sub-combinations for split calculations. Therefore, the overall time and space complexity of the algorithm is $O(n^{\sigma+1}2^n)$.*

We present a proof in Appendix B. The parameter $\sigma > 0$ can be used to control the size of split problems. Because using a large value of σ suppresses the number of required super-combinations, the number of communications required between independent calculations is also suppressed. Instead, the large value of σ requires a large memory space to store the sub-combinations in a processor. Therefore, σ can be used to control the trade-off between the number of communications and the memory space requirement. Because the algorithm requires the exchange of intermediate results $\binom{n-\sigma}{k}$ times for a loop with $|A| = k$ and is a relatively large number, decreasing the number of communications reduces the communication speed. In a case with many processors, however, a large value of σ can also reduce the communication speed because a large value of σ requires the transfer of a large amount of data, instead of reducing the number of communications. Table 1

	Increment for		
k	$\sigma = 1$	$\sigma = 2$	$\sigma = 3$
1	1	2	3
2	2	5	8
10	7	30	55
14	8	37	111
27	4	8	8

Table 1: Examples of the actual increment of various values of k and σ for $n = 32$. The increment is largest for all cases of σ for $k = 14$.

Algorithm 1 *Process-S*(S, a, n, n_p) calculates the functions $F(v,A)$, $M(A)$, and $Q^A(M(A))$ for combinations A derived from the given super-combination S.

Input: $S \subset V$: Super-combination, $a \in \mathbb{N}$: size of combination to be calculated, n: total number of nodes in the network, n_p: number of CPU processors (cores).

Output: $F(v,B)$, $Q(A)$, and $M(Q(A))$ for all sub-combinations of S with size a, $v \in A$, and $B = A \setminus \{v\}$.

1: $\mathcal{A} \leftarrow \{A \subset S : |A| = a\}$
2: Retrieve the local scores $s(v,B,X)$ for $B = A \setminus \{v\}$ $(v \in A \in \mathcal{A})$ from the $L^F(v,B,n,n_p)$-th processor.
3: Retrieve $F(u, B \setminus \{u\})$ for $u \in B$ $(B = A \setminus \{v\}, v \in A \in \mathcal{A})$ from the $L^F(u, B \setminus \{u\}, n, n_p)$-th processor.
4: Retrieve $Q^{A\setminus\{v\}}(M(A \setminus \{v\}))$ for $v \in A \in \mathcal{A}$ from the $L^Q(A \setminus \{v\}, n, n_p)$-th processor.
5: **for** each $A \in \mathcal{A}$ **do**
6: Calculate $F(v,B)$ for $v \in A, B = A \setminus \{v\}$ from $s(v,B)$ and $F(v, B \setminus \{u\})$ for $u \in B$ by Equation (1).
7: Calculate $M(A)$ and $Q^A(M(A))$ from $Q^{A\setminus\{v\}}(M(A \setminus \{v\}))$ and $F(v, A \setminus \{v\})$ by Equations (3) and (2).
8: **end for**
9: Store $Q(A)$ and $M^A(Q(A))$ $(A \in \mathcal{A})$ in the $L^Q(A, n, n_p)$-th processor.
10: Store $F(v,B)$ $(B = A \setminus \{v\}, A \in \mathcal{A})$ in the $L^F(v,B,n,n_p)$-th processor.

shows some examples of the actual overhead increment of the DP steps, that is, $\binom{n-\sigma}{k} \cdot \binom{k+\sigma}{k} / \binom{n}{k}$. As shown in the table, the increment because of redundant DP steps caused by the separation appears to be relatively small for a case of the practical size of n and σ. If the algorithm runs in parallel with hundreds of processors, the increment calculation in each processor is negligible as compared to the total amount of calculations, and thus, it does not noticeably affect the overall computation time. We discuss this later with the computational experiments presented in Section 4.2.

Algorithm 2 *Para-OS*(V, X, s, σ, n_p) calculates the exactly global optimal structure of the Bayesian network with respect to the input data X and the local score function s with n_p processors.

Input: V: set of input nodes (variables) where $|V| = n$, X: $(N \times n)$-input data matrix, $s(v, Pa, X)$: function $V \times 2^V \times \mathbb{R}^{N \times n} \to \mathbb{R}$ that returns the local Bayesian network score for variable v with its parent set $Pa \subset V$ w.r.t. the input data matrix X, $\sigma \in \mathbb{N}$: size of super-combination, n_p: number of CPU processors (cores).

Output: $G = (V, E)$: optimal Bayesian network structure.

1: {Initialization}
2: Calculate $F(v, \emptyset) = s(v, \emptyset, X)$ for all $v \in V$ and store it in the $L^F(v, \emptyset, n, n_p)$-th processor.
3: Store $F(v, \emptyset)$ as $Q^{\{v\}}(M(\{v\}))$ and $M(\{v\})(1) = v$ for all $v \in V$ in the $L^Q(\{v\}, n, n_p)$-th processor.
4: {Main Loop for size of A}
5: **for** $a = 1$ to $n - 1$ **do**
6: {S-phase: Execute the following for-loop on i in parallel. The $\{r = i \bmod n_p + 1\}$-th processor is responsible for the $(i+1)$-th loop.}
7: **for** $i = 0$ to $n\binom{n-1}{a} - 1$ **do**
8: $v \leftarrow i \bmod n + 1$
9: $j \leftarrow \lfloor i / n \rfloor + 1$
10: $Pa \leftarrow m(v, RLI^{-1}(j, n-1, a))$
11: Calculate $s(v, Pa, X)$ and store it in the local memory of the r-th processor.
12: **end for**
13: {Q-phase: Execute the following for-loop on i in parallel. The $\{r = i \bmod n_p + 1\}$-th processor is responsible for the $(i+1)$-th loop.}
14: **if** $a + \sigma + 1 > n$ **then**
15: $\sigma \leftarrow n - a - 1$.
16: **end if**
17: **for** $i = \binom{n}{a+\sigma+1} - \binom{n-\sigma}{a+1}$ to $\binom{n}{a+\sigma+1} - 1$ **do**
18: $S \leftarrow RLI^{-1}(i + 1, n, a + \sigma + 1)$.
19: Call *Process-S*$(S, a + 1, n, n_p)$.
20: **end for**
21: **end for**
22: Construct network $G = (V, E)$ by collecting the final sets of the parents selected in line 6 of *Process-S*$(\cdot)$.
23: **return** $G = (V, E)$.

3.2 Para-OS Algorithm

According to Theorems 7 and 8 and Corollary 9, the DP steps in Equations (1), (2), and (3) of Theorem 4 can be split by super-combinations of A. The pseudocode of the proposed algorithm is given by Algorithms 1 and 2. The former is a sub-routine of the latter main algorithm.

The algorithm consists of two phases: the S-phase and the Q-phase. In the former, each processor calculates the score function $s(v, Pa, X)$ independently without communication, whereas the latter calculates $F(v, A)$, $M(A)$, and $Q^A(M(A))$ along with communications among each other to exchange the results of $F(v, A)$, $M(A)$, and $Q^A(M(A))$. Note that in line 6 of Algorithm 1, we need

to store not only the local scores but also the optimal choices of parent node sets, although we do not describe this explicitly. This is required to reconstruct the optimal structure after the algorithm terminates.

In this algorithm, we need to determine which processor stores the calculated intermediate results. In order to calculate this, we define some functions as given below.

Definition 11 *We define function* $m' : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *as follows:*

$$m'(a,b) = \begin{cases} a & \text{if } a < b \\ a+1 & \text{otherwise} \end{cases}.$$

Using $m'(a,b)$*, we define function* $m : V \times 2^V \to 2^V$ *as follows:*

$$m(v,A) = \{ord^{-1}(m'(ord(u),ord(v))) : u \in A\}.$$

In addition, we define function $m'^{-1} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *as follows:*

$$m'^{-1}(a,b) = \begin{cases} a & \text{if } a < b \\ a-1 & \text{otherwise} \end{cases}.$$

Using $m'^{-1}(a,b)$*, we define function* $m^{-1} : V \times 2^V \to 2^V$ *as follows:*

$$m^{-1}(v,A) = \{ord^{-1}(m'^{-1}(ord(u),ord(v))) : u \in A\}.$$

The function $m(v,A)$ maps the combination A to a new combination in $V \setminus \{v\}$, and $m^{-1}(v,A)$ is the inverse function of $m(v,A)$. These are used in the proposed algorithm and the following function.

Definition 12 (Calculation of processor index to store and retrieve results) *We define functions* $L^Q : 2^V \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *and* $L^F : V \times 2^V \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *as follows:*

$$L^Q(A,n,n_p) \stackrel{\text{def}}{=} (RLI(A,n)-1) \bmod n_p + 1$$

and

$$L^F(v,A,n,n_p) \stackrel{\text{def}}{=} \left\{(RLI(m^{-1}(v,A),n-1)-1) \times n + (ord(v)-1)\right\} \bmod n_p + 1,$$

where $RLI(A,n)$ *is a function used to calculate the reverse lexicographical index (RLI) of combination* A *taken from* n *objects and* n_p*, the number of processors.*

Function $L^Q(A,n,n_p)$ locates the processor index used to store the results of $M(A)$ and $Q^A(M(A))$ and $L^F(v,A,n,n_p)$, the results of $F(v,A)$. By using RLIs, the algorithm can independently and discontinuously generate the required combinations and processor indices for storing/retrieving of intermediate results. We use RLIs instead of ordinal lexicographical indices because the conversion between a combination and the RLI can be calculated in linear time by preparing the index table once (Tamada et al., 2011). In Algorithm 2, the inverse function $RLI^{-1}(\cdot)$ is also used to reconstruct a combination from the index. See Appendix D for details of these calculations.

Figure 1 shows an example of the calculation of the DP for the super-combination S in a single processor in a single loop. Note that in the figure, although a super-combination is assigned to a single processor, $s(v,A,X)$ is calculated in a different processor from one that calculates $F(v,A)$ for the same $v \in V$ and $A \subset V \setminus \{v\}$.

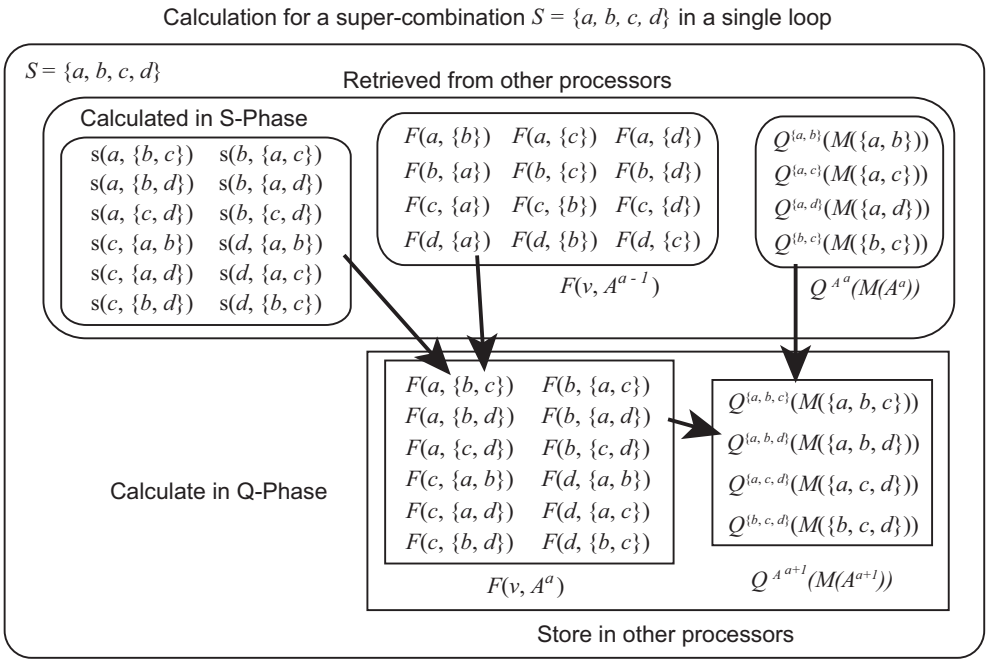


Figure 1: Schematic illustration of the calculation in a single loop on i for $a = 2$. A^a represents a subset $A \subset V$ where $|A| = a$.

4. Computational Experiments

In this section, we present computational experiments for evaluating the proposed algorithm. In the experiments, we first compared the running times and memory requirement for various values of σ. Next, we evaluated the running times of the original dynamic programming algorithm with a single processor and the proposed algorithm using 8 through 1024 processors. We also compared the results for different sizes of networks. In the experiments, we measured the running times using the continuous model score function BNRC proposed by Imoto et al. (2002). Finally, we tried to run the algorithm with as many nodes as possible on our supercomputers, as a proof of long-run practical execution that realizes the optimal large network structure learning. For this experiment, we used the discrete model score function BDe proposed by Heckerman et al. (1995), in addition to the BNRC score function. Brief definitions of BNRC and BDe are given in Appendix A. Before presenting the experimental results, we first describe the implementation of the algorithm and the computational environments used to execute the implemented programs.

4.1 Implementation and Computational Environment

We have implemented the proposed algorithm using the C programming language (ISO C99). The matrix computation in the BNRC score function is implemented using the BLAS/LAPACK library. The parallelization is implemented using MPI-1.1.

We have used two different supercomputer systems, RIKEN RICC and Human Genome Center Supercomputer System. The former is a massively parallel computer where each computation node has dual Intel Xeon 5570 (2.93 GHz) CPUs (8 CPU cores per node) and 12 GiB memory. The computation nodes are linked by X4 DDR InfiniBand. RICC employs Fujitsu's ParallelNavi that provides an MPI implementation, C compiler, BLAS/LAPACK library, and job scheduling. The latter system is similar to the former except that it has dual Intel Xeon 5450 (3 GHz) CPUs and 32 GiB memory per node. It employs OpenMPI 1.4 with Sun Grid Engine as a parallel computation environment. The compiler and the BLAS/LAPACK library are the Intel C compiler and Intel MKL, respectively.

In our implementation, each core in a CPU is treated equally as a single processor so that one MPI process runs in a single core. Therefore, 8 processes run in a computation node in both the systems. The memory in a single node is divided equally among these 8 processes.

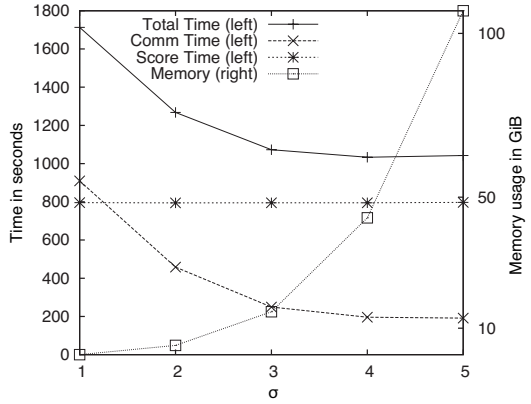
For the comparison presented later and the verification of the implementation, we also implemented the original *OS* algorithm proposed by Ott et al. (2004). The verification of the implementation was tested by comparing the optimal structures calculated by the implementations of both the original algorithm and the proposed algorithm for up to $p = 23$ using artificial simulated data with various numbers of processors. We also checked whether the greedy hill-climbing (HC) algorithm (Imoto et al., 2002) could search for a network structure having a better score than that of the optimal structure. We repeated the execution of the HC algorithm 10,000 times, and confirmed that no result was better than the optimal structure obtained using our algorithm.

4.2 Results

First, we generated artificial data with $N = 50$ (sample size) for the randomly generated DAG structure with $n = 23$ (node size). Refer to Appendix C for details on the generation of the artificial network and data. We used $n = 23$ because RICC has a limited running time of 72 hours. The calculation with a single processor for $n = 24$ exceeds this limit. In all the experiments, we carried out three measurements for each setting and took the average of these measured times as an observation for that setting. The total running times are measured for the entire execution of the program, including the input of the data from a file, output of the network to a file, and MPI initialization and finalization routine calls.

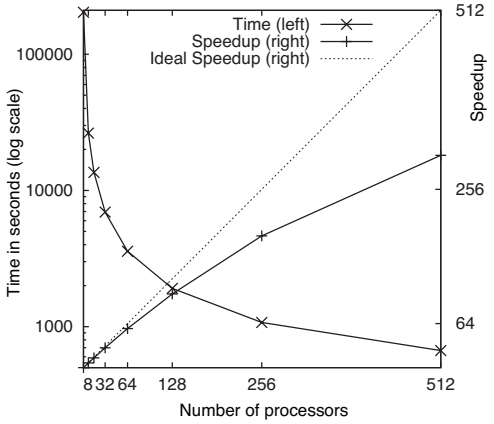
Figure 2 shows the result of the comparison of $\sigma = 1, \ldots, 5$ for $n = 23$. The row for $\sigma = 0$ shows the results of the original DP algorithm with a single processor. We measured the running times using 256 processors here. During the computation, we also measured the times required for calling MPI functions to exchange required data between processors, and the times required for calculating the score funtion $s(\cdot)$. As discussed in Section 3.1, σ controls the space-communication trade-off. We expected that an increase in the value of σ would reduce the time and increase the memory requirement. As expected, the total time decreased for up to $\sigma = 4$ with an increase in σ; however, it increased for $\sigma = 5$ and the memory requirement also increased significantly. As shown in the figure, σ does not affect the score calculation time. From these results, we employed $\sigma = 3$ for later experiments because the increase in the memory requirement and the decrease in the total time appeared reasonable.

Next, we compared the running times for various numbers of processors. We carried out measurements for $n_p = 8, 16, 32, 64, 128, 256, 512$, and 1024 processors using 1, 2, 4, 8, 16, 32, 64, and 128 computation nodes, respectively, on RICC. Here, n_p represents the number of processors.



σ	Total Time	Cm Time	Sc Time	Mem
0	-	-	-	1.27
1	1713.08	909.36	795.61	1.92
2	1267.98	458.32	794.27	4.75
3	1072.85	249.06	795.01	15.01
4	1033.20	196.04	794.75	43.69
5	1042.21	191.13	797.00	106.91

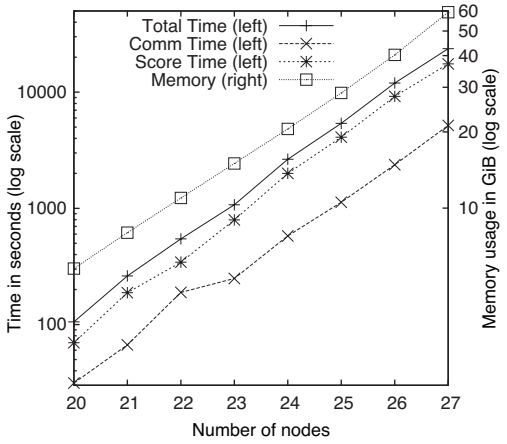
Figure 2: Running times and memory requirements with $\sigma = 1,\ldots,5$ for $n = 23$ and $N = 50$ with 256 processors. "Total Time" represents the total time required for execution in seconds, "Cm Time" represents the total communication time required for calling MPI functions within the total time; "Sc Time," the time required for score calculation; and "Mem," the memory requirement in GiB.



n_p	$T(n_p)$	$S(n_p)$	$E(n_p)$	$T_c(n_p)$	$R_c(n_p)$
1	203297.04	1.00	1.00	0.00	0.00
8	26367.58	7.71	0.96	145.24	0.01
16	13563.89	14.99	0.94	490.24	0.04
32	6932.09	29.33	0.92	398.29	0.06
64	3574.89	56.87	0.89	302.46	0.08
128	1909.72	106.45	0.83	269.90	0.14
256	1072.85	189.49	0.74	249.06	0.23
512	667.38	304.62	0.59	251.46	0.38
1024	515.29	394.53	0.39	305.12	0.59

Figure 3: Scalability test results for $n = 23$ and $N = 50$ with $\sigma = 3$. We did not present the result for $n_p = 1024$ in the graph on the left-hand side because the speedup was too low.

As mentioned above, we used $\sigma = 3$. For $n_p = 1$, we used the implementation of the original DP algorithm. Therefore, we do not use the super-combination-based separation of our proposed algorithm although it works for $n_p = 1$. Figure 3 shows the experimental result. We evaluated the parallelization scalability of the proposed algorithm from the speedup $S(n_p)$ and efficiency $E(n_p)$. The speedup $S(n_p)$ is defined as $S(n_p) = T(1)/T(n_p)$, where n_p is the number of processors and $T(n_p)$, the running time with n_p processors. If $S(n_p) = n_p$, then it is called the ideal speedup where n_p-hold speedup is obtained by n_p processors. The parallelization efficiency $E(n_p)$ is defined as $E(n_p) = S(n_p)/n_p$. In the case of ideal speedup, $E(n_p) = 1$ for any n_p. Generally, parallel programs



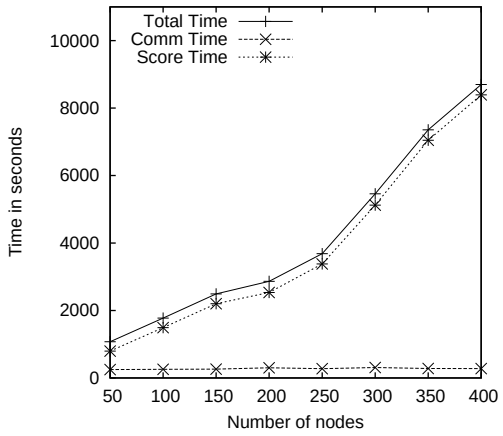
n	Total	Cm Time	(ratio)	Sc Time	Mem
20	105.54	31.35	0.30	69.71	5.77
21	261.84	66.77	0.26	187.91	8.00
22	546.07	188.80	0.35	344.00	10.97
23	1072.85	249.06	0.23	795.01	15.01
24	2645.04	579.61	0.22	1999.56	20.55
25	5386.38	1127.62	0.21	4095.74	28.50
26	11976.90	2374.81	0.20	9199.75	40.25
27	23686.51	5174.94	0.22	17517.04	59.31

Figure 4: Comparison of running tims for various network sizes. Column n represents the size of the network and "(ratio)," the ratio of "Cm Time" to "Total." "Mem" is represented in GiB. Other columns have the same meaning as in Figure 2.

that have $E(n_p) \geq 0.5$ are considered to be successfully parallelized. As shown in the table in Figure 3, the efficiencies are 0.74 and 0.59 for $n_p = 256$ and 512, respectively. However, with 1024 processors, the efficiency became 0.39 and the speedup was very low as compared to that with 512 processors, and therefore, it is not efficient and feasible. From these results, we can conclude that the program can run very efficiently in parallel for up to 256 processors, and acceptably for up to 512 processors.

$T_c(n_p)$ in Figure 3 represents the time required for calling MPI functions during the executions, and $R_c(n_p)$ is a ratio of $T_c(n_p)$ to the total time $T(n_p)$. Except for $n_p = 8$, $T_c(n_p)$ decreases with an increase in n_p because the amount of communication for which each processor is responsible decreases. However, it did not decrease linearly; in fact, for $n_p \geq 512$, it increased. This may indicate the current limitation of both our algorithm and the computer used to carry out this experiment. For $n_p = 8$, $T_c(n_p)$ was very small. This is mainly because communication between computation nodes was not required for this number of processors. If we subtract $T_c(n_p)$ from $T(n_p)$, then the efficiency $E(n_p)$ becomes 0.96, 0.95, and 0.94 for 256, 512, and 1024 processors, respectively. This result suggests that the redundant calculation in our proposed algorithm does not have a great effect, and the communication cost is the main cause of the inefficiency of our algorithm. Therefore, improving the communication speed in the future may significantly improve the efficiency of the algorithm with a larger number of processors.

Next, we compared the running times for various network sizes. We generated artificial simulated data for $n = 20$ to 27 as we did for the above experiment with $n = 23$. We measured the running times with 256 processors and $\sigma = 3$. Figure 4 shows the result. As shown in the figure, both the time and the space required increased exponentially. Note that both the left- and the right-hand side y-axes are in log scale. The communication time decreased slightly for up to $n = 26$ with an increase in n. However, for $n = 27$, it started to increase. From these results, we can say that the score calculation remains dominant and the communication does not contribute significantly to the total running times for this range of n with $n_p = 256$.



N	Total	Cm Time	Sc Time
50	1072.85	249.06	795.01
100	1775.73	255.55	1492.78
150	2490.77	261.01	2201.52
200	2863.91	302.04	2533.35
250	3684.08	276.14	3379.34
300	5460.16	309.71	5122.12
350	7354.91	281.40	7045.05
400	8698.11	276.68	8392.88

Figure 5: Comparison of running times for various sample sizes. Column N represents the number of samples. Other columns have the same meaning as in Figure 2.

To check the scalability of the algorithm to the sample size, we compared the running times for various sample sizes. We generated artificial simulated data with $N = 50, 100, 150, 200, 250, 300, 350,$ and 400 for the artificial network of $n = 23$, which is used for the previous analyses. Theoretically, the sample size does not affect the running times, except for the score calculation. Figure 5 shows the result. We confirmed that the communication times are almost constant for all the tested sample sizes and that the score calculation increased with the sample sizes, as was expected. Note that the calculation of the BNRC score function is not in linear time, and it is difficult to determine the exact time complexity because it involves an iterative optimization step (Imoto et al., 2002).

4.3 Structure Search for Large Networks

Finally, we tried to search for the optimal structure of nodes with as many nodes as possible in the HGC system because it allows long execution for up to two weeks with 256 processor cores and has a larger memory in each computation node.

As in the above experiment, we first generated random DAGs having various numbers of nodes, and then generated simulated data with 50 samples. With the BNRC score function, we have succeeded in searching for the optimal structure of a 31-node network by using 464.3 GiB memory in total (1.8 GiB per process) with 256 CPU cores in 32 computation nodes. For this calculation, we did not impose a restriction on the parent size or any other parameter restrictions. 8 days 6 hours 50 minutes 24 seconds were required to finish the calculation. The total time required for calling MPI functions was 2 days 15 hours 11 minutes 58 seconds. This is 32% of the total running time. Therefore, the communication time became a relatively large portion of the total computation time, relative to that in the case of $n = 27$ presented above. As described in Parviainen and Koivisto (2009), thus far, the largest network search that has been reported was for a 29-node network (Silander and Myllymäki, 2006). Therefore, our result improved upon this result without even using an external hard disk drive.

To search for the optimal structure of an even larger network, we used the BDe network score, which is a discrete model that is much faster than the BNRC score. Generally, the BDe score can be

calculated 100 times faster than the BNRC score (data not shown). Using the BDe score function, we successfully carried out optimal structure search for a 32-node network without any restriction using 836.1 GiB memory (3.3 GiB per process) in total with 256 CPU cores. The total computation time was 5 days 14 hours 24 minutes and 34 seconds. The MPI communication time was 4 days 12 hours 56 minutes 26 seconds, and this is 81% of the total time. Thus, for $n = 32$ with the BDe score function, the calculation of score functions requires relatively very little time (actually, it required only around 1.5 hours per process) as compared to the total time, and the communication cost becomes the dominant part and the bottleneck of the calculation.

These results show that our algorithm is applicable to the optimal structure search of relatively large-sized networks and it can be run on modern low-cost supercomputers.

5. Discussions

In this paper, we have presented a parallel algorithm to search for the score-based optimal structure of Bayesian networks. The main feature of our algorithm is that it can run very efficiently on massively parallel computers in parallel. We confirmed the scalability of the algorithm to the number of processors through computational experiments and successfully demonstrated optimal structure search for a 32-node network with 256 processors, an improvement over the most successful result reported thus far. Our algorithm overcomes the bottleneck of the previous algorithm by using a large amount of distributed memory for large-scale Bayesian network structure search.

Our algorithm has a feature similar to that of an algorithm recently proposed by Parviainen and Koivisto (2009) that requires less space. Both algorithms divide the search space of the problem, and provide a way to compute the optimal structure in parallel. Both are capable of breaking the current limitation of the network size in optimal network structure search. However, these two algorithms differ in several respects. First, Parviainen and Koivisto (2009) primarily intended to develop a space-time trade-off algorithm to overcome the bottleneck of the search problem. They found that the search problem can be divided into sub-problems and that these sub-problems can be solved independently with less space. Therefore, although the time requirement increases with a decrease in the space requirement, they mentioned that their algorithm can obtain the optimal structure for a 34-node network by massive parallelization. Our algorithm, on the other hand, overcomes the bottleneck in a more straightforward way. We found a way to divide the DP steps of the fastest known algorithm with a relatively low overhead cost. In terms of memory requirement, our algorithm consumes much more memory space than that of Parviainen and Koivisto (2009) and even more than the original DP algorithm to realize parallelization. However, our algorithm can actually search for the optimal structure of a 32-node network with 256 processors in less than a week, including score calculation, whereas Parviainen and Koivisto (2009) computed only the partial problems. Their estimate from their empirical result is 4 weeks using 100 processors to obtain the optimal structure for a 31-node network. In addition, their estimate ignores the parallelization overhead that generally becomes problematic in parallelization as well as score calculation, which requires the most time in the actual application. We showed that our algorithm works efficiently with up to 256 processors, and acceptably with up to 512 processors. An optimal search for even larger networks may be realized by improving the current implementation. Our implementation regards each processor core in a CPU equivalently. Therefore, exploiting the modern multi-core CPUs can reduce the communications required among computation nodes and increase the amount of memory space for independent calculations without requiring improved hardware relative to current supercomputers.

Acknowledgments

Computational time was provided by the Supercomputer System, Human Genome Center, Institute of Medical Science, The University of Tokyo; and RIKEN Supercomputer system, RICC. This research was supported by a Grant-in-Aid for Research and Development Project of the Next Generation Integrated Simulation of Living Matter at RIKEN and by MEXT, Japan. The authors would like to thank the anonymous referees for their valuable comments.

Appendix A. Definitions of Score Functions

In this paper, we use BNRC (Imoto et al., 2002) and BDe (Heckerman et al., 1995) as a score function $s(\cdot)$ for computational experiments of the proposed algorithm. Here, we present brief definitions of these score functions.

A.1 The BNRC Score Function

BNRC is a score function for modeling continuous variables. In a continuous model, we consider the joint density of the variables instead of their joint probability. We search the network structure G by maximizing the posterior of G for the input data matrix X. The posterior of G is given by

$$\pi(G|X) = \pi(G) \int \prod_{i=1}^{N} \prod_{j=1}^{n} f(x_{ij}|pa_{ij}^{G};\theta_j)\pi(\theta_G|\lambda)d\theta_G,$$

where $\pi(G)$ is the prior distribution of G; $f(x_{ij}|pa_{ij}^{G};\theta_j)$, the local conditional density for the j-th variable; $pa_{ij}^{G} = (pa_{i1}^{(j)},\ldots,pa_{iq_j}^{(j)})$, the set of observations in the i-th sample of q_j variables that represents the direct parents of the j-th node in a network; $\theta_G = (\theta_1,\ldots,\theta_n)$, the parameter vector of the conditional densities to be estimated; and $\pi(\theta_G|\lambda)$, the prior distribution of θ_G specified by the hyperparameter λ. Conditional density $f(x_{ij}|pa_{ij}^{G};\theta_j)$ is modeled by nonparametric regression with B-spline basis functions given by

$$f(x_{ij}|pa_{ij}^{G};\theta_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left[-\frac{\{x_{ij} - \sum_{k=1}^{q_j} m_{jk}(pa_{ik}^{(j)})\}^2}{2\sigma_j^2}\right],$$

where $m_{jk}(p_{ik}^{(j)}) = \sum_{l=1}^{M_{jk}} \gamma_{lk}^{(j)} b_{lk}^{(j)}(p_{ik}^{(j)})$, $\{b_{1k}^{(j)}(\cdot),\ldots,b_{M_{jk},k}^{(j)}(\cdot)\}$ is the prescribed set of M_{jk} B-splines; σ_j, the variance, and $\gamma_{lk}^{(j)}$, the coefficient parameters. By taking a $-2\log$ of the posterior, the BNRC score function for the j-th node is defined as

$$s_{BNRC}(X_j, Pa^G(X_j), X) = -2\log\left\{\pi_j^G \int \prod_{i=1}^{N} f(x_{ij}|pa_{ij}^{G};\theta_j)\pi_j(\theta_j|\lambda_j)d\theta_j\right\},$$

where π_j^G is the prior distribution of the local structure associated with the j-th node; and $\pi_j(\theta_j|\lambda_j)$, the decomposed prior distribution of θ_j specified by the hyperparamter λ_j.

A.2 The BDe Score Function

BDe, a score function for the discrete model, can be applied to discrete (categorical) data. As in the case of BNRC, BDe also considers the posterior of G; that is,

$$P(G|X) \propto P(G) \int P(X|G,\theta)P(\theta|G)d\theta,$$

where $P(X|G,\theta)$ is the product of local conditional probabilities (likelihood of X given G) and $P(\theta|G)$, the prior distribution for parameters θ. In the discrete model, we employ multinomial distribution for modeling the conditional probability and the Dirichlet distribution as its prior distribution. Let X_j be a discrete random variable corresponding to the j-th node, which takes one of r values $\{u_1,\ldots,u_r\}$, where r is the number of categories of X_j. In this model, the conditional probability of X_j is parameterized as

$$P(X_j = u_k | Pa^G(X_j) = u_{jl}) = \theta_{jlk},$$

where u_{jl} $(l = 1,\ldots,r^{q_j})$ is a combination of values for the parents and q_j, the number of parents of the j-th node. Note that $\sum_{k=1}^{r} \theta_{jlk} = 1$. For the discrete model, the likelihood can be expressed as

$$P(X|G,\theta) = \prod_{j=1}^{n} \prod_{l=1}^{r^{q_j}} \prod_{k=1}^{r} \theta_{jlk}^{N_{jlk}},$$

where N_{jlk} is the number of observations for the j-th node whose values equal u_k in the data matrix X with respect to a combination of the parents' observation l. $N_{jl} = \sum_{k=1}^{r} N_{jlk}$ and θ denotes a set of parameters θ_{jlk}. For the parameter set θ, we assume the Dirichlet distribution as $\pi(\theta|G)$; then, the marginal likelihood can be described as

$$\int P(X|G,\theta)P(\theta|G)d\theta = \prod_{j=1}^{n} \prod_{l=1}^{r^{q_j}} \frac{\Gamma(\alpha_{jl})}{\Gamma(\alpha_{jl}+N_{jl})} \prod_{k=1}^{r} \frac{\Gamma(\alpha_{jlk}+N_{jlk})}{\Gamma(\alpha_{jlk})},$$

where θ is a set of parameters; α_{jlk}, a hyperparameter for the Dirichlet distribution; and $\alpha_{jl} = \sum_{k=1}^{r} \alpha_{jlk}$. By taking $-\log$ of the posterior, the BDe score function for the j-th node is defined as

$$s_{BDe}(X_j, Pa^G(X_j), X) = -\log \pi_j^G - \log \left\{ \prod_{l=1}^{r^{q_j}} \frac{\Gamma(\alpha_{jl})}{\Gamma(\alpha_{jl}+N_{jl})} \prod_{k=1}^{r} \frac{\Gamma(\alpha_{jlk}+N_{jlk})}{\Gamma(\alpha_{jlk})} \right\},$$

where π_j^G is the prior probability of the local structure associated with the j-th node.

Appendix B. Proof of Corollary 10

Proof We prove that $\binom{n-\sigma}{k} \cdot \binom{k+\sigma}{k} = \binom{n}{k} O(n^\sigma)$.

$$\begin{aligned} \binom{n-\sigma}{k} \cdot \binom{k+\sigma}{k} &= \frac{(n-\sigma)!}{k!(n-\sigma-k)!} \cdot \frac{(k+\sigma)!}{k!(k+\sigma-k)!} \\ &= \frac{(n-\sigma)!}{k!(n-\sigma-k)!} \cdot \frac{(k+\sigma)!}{k!\sigma!} \cdot \frac{(n-k)!}{(n-k)!} \cdot \frac{n!}{n!} \end{aligned}$$

$$
\begin{aligned}
&= \frac{n!}{k!(n-k)!} \cdot \frac{(n-\sigma)!(k+\sigma)!(n-k)!}{(n-\sigma-k)!k!\sigma!n!} \\
&= \binom{n}{k} \cdot \frac{(n-\sigma)!}{(n-\sigma-k)!} \cdot \frac{(k+\sigma)!}{k!\sigma!} \cdot \frac{(n-k)!}{n!} \\
&= \binom{n}{k} \cdot \frac{(n-\sigma)}{n} \frac{(n-\sigma-1)}{(n-1)} \cdots \frac{(n-\sigma-k+1)}{(n-k+1)} \cdot \frac{(k+\sigma)\cdots(k+1)k!}{\sigma!k!} \\
&= \binom{n}{k} \cdot O(1) \cdot O((k+\sigma)^{\sigma}) = \binom{n}{k} \cdot O((k+\sigma)^{\sigma}).
\end{aligned}
$$

For $n < k+\sigma$, we consider only super-combinations of size n. Thus, for each k, there are at most $\binom{n}{k} O(n^{\sigma})$ calculations. ■

Appendix C. Method for Generating Artificial Network and Data

To generate the random DAG structure, we simply added edges at random to an empty graph so that the acyclic structure is maintained and the average degree becomes $d = 4.0$. Consequently, the number of total edges equals $n \cdot d/2$. Note that the degree of DAGs does not affect the execution time of the algorithm as the algorithm searches all possible structures. To generate the artificial data, we first randomly assigned 8 different nonlinear or linear equations to the edges and then generated the artificial numerical values based on the normal distribution and the assigned equations. Figure 6 shows the assigned equations and examples of the generated data. If a node has more than two parent nodes, the generated values are summed before adding the noise. We set the noise ratio to be 0.2.

To apply BDe to the artificial data on the $n = 32$ network, we discretized the continuous data generated by the same method for all variables into three categories ($r = 3$) and then executed the algorithm for these discretized values.

Appendix D. Efficient Indexing of Combinations

When running the algorithm, we need to generate combination vectors discontinuously and independently in a processor. To do this efficiently, we require an algorithm that calculates a combination vector from its index and the index from its combination vector. Buckles and Lybanon (1977) presented an efficient lexicographical index - vector conversion algorithm. However, this algorithm requires the calculation of binomial coefficients for every possible element in a combination every time. To speed up this calculation, we developed a linear time algorithm that needed polynomial time to construct a reusable table (Tamada et al., 2011). Our algorithm actually deals with *the reverse lexicographical index* instead of the ordinal lexicographical index; this enables us to calculate the table only once and to make it reusable. In this section, we present the algorithms as described in Tamada et al. (2011). See Tamada et al. (2011) for details and the proofs of the theorems. In this section, note that we assume that $\sum_{i=k}^{n} f_i = 0$ for any f_i if $n < k$.

Theorem 13 (RLI calculation) *Let $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ be the set of all the combinations of length k taken from n objects, arranged in lexicographical order, where $m = \binom{n}{k}$. We call i the lexicographical index of $C_i \in \mathcal{C}$. Let us define the reverse lexicographical index of $C_i \in \mathcal{C}$, $RLI(C_i, n)$*

(a) $f_a(X) = X + \varepsilon$

(b) $f_b(X) = -X + \varepsilon$

(c) $f_c(X) = \frac{1}{1+e^{-4X}} + \varepsilon$

(d) $f_d(X) = -\frac{1}{1+e^{-4X}} + \varepsilon$

(e) $f_e(X) = -(X-1.5)^2 + \varepsilon$

(f) $f_f(X) = (-X-2.5)^2 + \varepsilon$

(g) $f_g(X) = -(-X-2.5)^2 + \varepsilon$

(h) $f_h(X) = (X-1.5)^2 + \varepsilon$

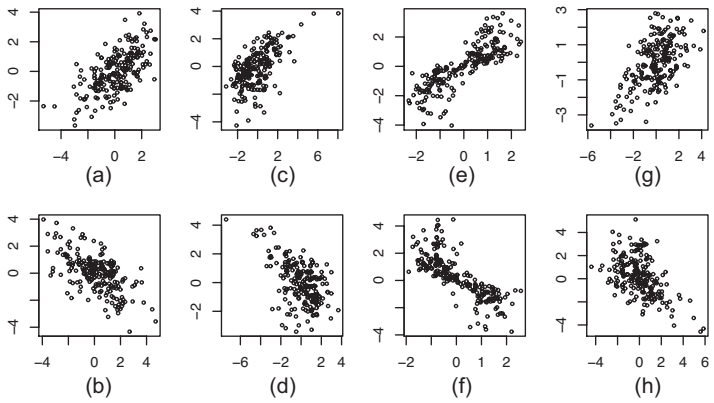


Figure 6: Left: Linear and nonlinear equations assigned to each edge in artificial networks. ε represents the noise based on the normal distribution. Right: Examples of the values generated by these equations.

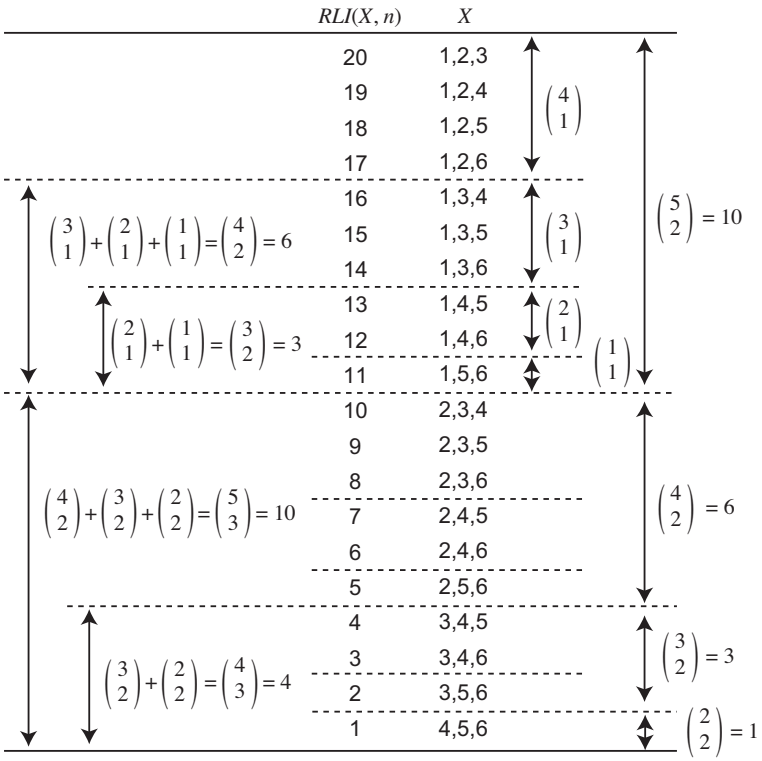


Figure 7: RLI calculation for $n = 6$ and $k = 3$.

$\stackrel{\text{def}}{=} m - i + 1$. *Suppose that we consider a combination of natural numbers, that is, some combination* $X = \{x_1, x_2, \ldots, x_k\} \in C$, *where* $x_i < x_j$ *if* $i < j$ *and* $x_i \in \{1, 2, \ldots, n\}$. *Then,* $RLI(X, n)$ *can be*

Algorithm 3 *RLITable*(m) generates the index table for conversion between a combination and the index.

Input: m: maximum number of elements appearing in a combination,
Output: $T : m \times m$ index table.

```
{Initialization}
T(i,1) ← 0   (1 ≤ i ≤ m).
T(1,j) ← j − 1   (2 ≤ j ≤ m).
{Main Routine}
for i = 2 to m do
    for j = 2 to m do
        T(i,j) ← T(i−1,j) + T(i,j−1).
    end for
end for
return T.
```

Algorithm 4 *RLI*(X,n,T) calculates the reverse lexicographical index of the given combination X.

Input: $X = \{x_1, x_2, \ldots x_k\}$ $(x_1 < \cdots < x_k \wedge 1 \le x_i \le n)$: input combination of length k taken from n objects, n: total number of elements, T: index table calculated by *RLITable*$(\cdot)$.
Output: reverse lexicographical index of combination X.

```
r ← 0
for i = 1 to k do
    r ← z + T(k−i+1, n−k−x_i+i+1).
end for
return r + 1.
```

calculated by

$$RLI(X,n) = \sum_{i=1}^{|X|} \binom{n-x_i}{|X|-i+1} + 1.$$

Figure 7 shows an example of the calculation of RLI for $n = 6$ and $k = 3$. For example, $RLI(\{1,3,5\},6) = \binom{5}{3} + \binom{3}{2} + \binom{1}{1} + 1 = 15$.

Corollary 14 (RLI calculation by the index table) *Let T be a $(k, n-k+1)$-size matrix whose element $T(\alpha,\beta) \stackrel{\text{def}}{=} \binom{\alpha+\beta-2}{\alpha}$. Matrix T can be calculated only by $(n-1)(n-k-1)$ time addition and by using T, $RLI(X,n)$ can be calculated in linear time by $RLI(X,n) = \sum_{i=1}^{k} T(k-i+1, n-k-x_i+i+1)$ + 1.*

Algorithm 3 shows the pseudocode used to generate T and Algorithm 4, the pseudocode of $RLI(X,n)$. The inverse function that generates the combination vector for an RLI can be calculated by simply finding the largest column position of T, subtracting the value in the table from the index, and then repeating this k times.

Corollary 15 *Let $RLI(X,n)$ be the reverse lexicographical index defined above for combination $X = \{x_1, x_2, \ldots, x_k\}$. The inverse function of $RLI(X,n)$, that is, the i-th element x_i of*

Algorithm 5 $RLI^{-1}(r,n,k,T)$ calculates the combination vector of length k from n elements, corresponding to the given reverse lexicographical index r.

Input: r: reverse lexicographical index of the combination to be calculated, n: total number of elements, k: length of the combination, T: index table calculated by $RLITable(\cdot)$.

Output: $X = \{x_1, x_2, \ldots, x_k\}$: combination corresponding to r.

```
r ← r − 1.
for i = 1 to k do
    for j = n − k + 1 to 1 do
        if r ≥ T(k − i + 1, j) then
            x_i ← n − k − j + i + 1.
            r ← r − T(k − i + 1, j).
            break
        end if
    end for
end for
return X = {x_1, x_2, ..., x_k}.
```

$RLI^{-1}(RLI(X,n),n,k)$ *can be calculated by, for* $i = 1,\ldots,k$,

$$x_i = \arg\max_j T(k-i+1, n-k-j+i+1) < RLI(X) - \sum_{\alpha=1}^{i-1} T(k-\alpha+1, n-k-x_\alpha+\alpha+1).$$

Algorithm 5 shows the pseudocode used to calculate $RLI^{-1}(r,n,k)$ *for RLI* r *in linear time.*

The search space of 15 is independent of k but dependent on n. This is because once x_i is identified, we need to search only x_j for $j > i$ such that $x_j > x_i$. Therefore, the inverse function $RLI^{-1}(\cdot)$ can generate the combination vector in linear time. By using the binary search, the search of proper objects in a vector can be calculated in log time. See Tamada et al. (2011) for the binary search version of this algorithm.

The advantage of using RLI is that once T is calculated for m, it can be used for calculating $RLI(X,n)$ and $RLI^{-1}(r,n,k)$ for any n and k, where $k \leq n \leq m$. The normal lexicographical order can also be calculated in a similar manner for fixed values of n and k. However, it is required for constructing a different table for different values of n and k.

References

B. P. Buckles and M. Lybanon. Algorithm 515: Generation of a vector from the lexicographical index [G6]. *ACM Transductions on Mathematical Software*, 3(2):180–182, 1977.

D. M. Chickering, D. Geiger, and D. Heckerman. Learning Bayesian networks: Search methods and experimental results. In *Proceedings of the Fifth Conference on Artificial Intelligence and Statistics*, pages 112–128, 1995.

N. Friedman, M. Linial, I. Nachman, and D. Pe'er. Using Bayesian networks to analyze expression data. *J. Computational Biology*, 7:601–620, 2000.

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

S. Imoto, T. Goto, and S. Miyano. Estimation of genetic networks and functional structures between genes by using Bayesian networks and nonparametric regression. *Pacific Symposium on Biocomputing*, 7:175–186, 2002.

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

S. Ott, S. Imoto, and S. Miyano. Finding optimal models for small gene networks. *Pacific Symposium on Biocomputing*, 9:557–567, 2004.

P. Parviainen and M. Koivisto. Exact structure discovery in Bayesian networks with less space. *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI 2009)*, 2009.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman Publishers, San Mateo, CA, 1988.

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

T. Silander and P. Myllymäki. A simple approach for finding the globally optimal Bayesian network structure. *Proceedings of the 22th Conference on Uncertainty in Artificial Intelligence (UAI 2006)*, pages 445–452, 2006.

Y. Tamada, S. Imoto, and S. Miyano. Conversion between a combination vector and the lexicographical index in linear time with polynomial time preprocessing, 2011. *submitted*.

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