A Learning Algorithm for Bayesian Networks and Its Efficient Implementation on GPUs

Yu Wang, *Student Member, IEEE*, Weikang Qian, *Member, IEEE*, Shuchang Zhang, Xiaoyao Liang, *Member, IEEE*, and Bo Yuan

Abstract—The wide application of omics research has produced a burst of biological data in recent years, which has in turn increased the need to infer biological networks from data. Learning biological networks from experimental data can help detect and analyze aberrant signaling pathways, which can be used in diagnosis of diseases at an early stage. Most networks can be modeled as Bayesian networks (BNs). However, because of its combinatorial nature, computational learning of dependent relationships underlying complex networks is NP-complete. To reduce the complexity, researchers have proposed to use Markov chain Monte Carlo (MCMC) methods to sample the solution space. MCMC methods guarantee convergence and traversability. However, MCMC is not scalable for networks with more than 40 nodes because of the computational complexity. In this work, we optimize an MCMC-based learning algorithm and implement it on a general-purpose graphics processing unit (GPGPU). We achieve a $2.46\times$ speedup by optimizing the algorithm and an additional 58-fold acceleration by implementing it on a GPU. In total, we speed up the algorithm by $143\times$. As a result, we can apply this system to networks with up to 125 nodes, a size that is of interest to many biologists. Furthermore, we add artificial interventions to the scores in order to incorporate prior knowledge of interactions into the Bayesian inference, which increases the accuracy of the results. Our system provides biologists with a more computational efficient tool at a lower cost than previous works.

Index Terms—Bayesian networks, GPU, MCMC, priors, parallel computing

1 INTRODUCTION

Recently, a vast amount of microarray data has become available for public use, such as ArrayExpress [1], GENEVESTIGATOR [2], and NASCArrays [3]. The wide application of omics research has promoted the growth of computational biology, a field using the combination of theoretical methods, data analysis techniques, and simulations to study biological systems. Experiments have been carried out at whole-genome scale. Data from various resources, such as transcriptomics, proteomics, genomics, and metabolics, need to be integrated to learn connections among targeted entities. Analyzing the connections, for example, in signaling transduction networks, is helpful to detect diseases caused by abnormal interactions. The study conducted by Irish et al. showed that slight differences in the connections of signaling transduction networks are correlated with different clinical results [4]. Among the various modeling methods, the Bayesian network (BN) is regarded as one of the most effective models for constructing biological networks. In the BN model, target entities (such as genes, proteins, and molecules) are modeled as nodes, and interactions between those entities are modeled as edges.

However, the problem of learning networks from data has been proven to be NP-complete [5]. Therefore, in practice, Markov chain Monte Carlo (MCMC) methods, which are *randomized* algorithms, have been used to reduce the complexity [6]. However, those algorithms are still too time-consuming. In order to make it practical for biological use, we propose two major algorithmic improvements. Our experiment results showed that the resulting algorithm is 2.46-fold faster than the basic algorithm. We also propose a way of adding artificial interventions to take prior knowledge for edges into consideration to increase the accuracy of the learning results.

Besides algorithmic improvements, in this work, we also apply graphics processing units (GPUs) to further accelerate the BN learning algorithm. GPUs are known to be highly efficient for massively parallel computation, which is a good fit for our BN learning algorithm, since the algorithm is inherently parallelizable. The combination of our algorithmic improvements and the GPU implementation allows us to learn graphs of up to 125 nodes on a single compute node and achieve $143\times$ speedup. The result is better than any other published GPU implementations of a BN learning algorithm. In the field of biology, many laboratories need to process networks with fewer than 100 nodes. For example, studies conducted by Ao et al. showed that the sizes of endogenous networks are usually within 100 nodes [14], [15]. Thus, our proposed approach provides a powerful tool for these studies.

In summary, we have made the following contributions in this work:

- We make two major improvements to the MCMC-based BN learning algorithm. They are:

- *Y. Wang, S. Zhang, X. Liang, and B. Yuan are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China 200240. E-mail: ywang03@eecs.harvard.edu, {zhangwfjh, liang-xy, boyuan}@sjtu.edu.cn.*
- *W. Qian is with the UM-SJTU Joint Institute, Shanghai Jiao Tong University, Shanghai, China 200240. E-mail: qianwk@sjtu.edu.cn.*

Manuscript received 25 Mar. 2014; revised 27 Nov. 2014; accepted 29 Nov. 2014. Date of publication 31 Dec. 2014; date of current version 16 Dec. 2015. Recommended for acceptance by S. Yu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2387285

1045-9219 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

1) a procedure that maps a k-combination to an integer and a reverse procedure that maps an integer to a k-combination. The former procedure is used in fetching the "scores" from the memory for evaluating different BNs. It replaces hashing that was used in the previous work [8]. Since the major work of the algorithm is computing the index corresponding to a k-combination in order to fetch the score for that k-combination, the function that calculates the index for a k-combination is called extensively. Our proposed approach gives us a much faster way to obtain index than hashing. The latter procedure is used in the GPU implementation to assign tasks to different threads. The procedure makes it convenient to distribute tasks to GPU threads evenly and hence, improves the performance.
2) A new scoring function that is easy to compute. In traditional approaches, scoring procedure is sum-based and calculated in the logarithmic space, which requires a number of time-consuming logarithmic and exponential operations. Our new scoring is max-based, which only requires addition and comparison in the logarithmic space, greatly simplifying the computation.

- We implement our algorithm efficiently on a GPU by exploiting the parallelism of our algorithm.
- We add artificial interventions in order to incorporate prior knowledge of interactions into the Bayesian inference.

The remainder of the paper is organized as follows. In Section 2, we discuss some related works. In Section 3, we give the background on the problem of learning BNs. In Section 4, we describe our algorithms that map between an integer and a k-combination, which are important in the optimization and implementation of the BN learning algorithm. In Section 5, we describe the optimized learning algorithm. In Section 6, we describe our method for adding artificial interventions. In Section 7, we discuss the implementation of the proposed algorithm on a GPU. In Section 8, we show the experimental results of the proposed system for learning BNs. In Section 9, we conclude the paper.

2 Related Works

In order to accelerate the learning of BNs, novel computational platforms such as field-programmable gate array (FPGA), computing clusters, and GPUs have been applied [7], [8], [9]. Linderman et al. implemented an MCMC algorithm on a GPU and achieved a 7.5× speedup over the implementation on a general-purpose processor (GPP) [8]. Asadi et al. implemented a similar algorithm on a multi-FPGA system and achieved more than 10,000× speedup on the scoring subroutine [7]. Tamada et al. proposed a novel algorithm to calculate the probability of edges by sampling subnetworks [10]. With their supercomputing system of 64 compute nodes, they could learn networks of up to 10,540 nodes within 301 seconds. Tamada et al. also presented a parallel algorithm which makes it possible to run their algorithm in parallel using 256 processors [11]. Nikolova et al. proposed a heuristic algorithm, which was able to handle a 500-node network within 107 seconds using 1,024 computation cores [12]. Nikolova et al. also proposed and extended their exact learning algorithm on an IBM Blue Gene/P system and an AMD Opteron InfiniBand cluster [13].

Our baseline implementation adopts the techniques described by Linderman et al. [8], which is the state-of-the-art MCMC-based BN learning algorithm and an improvement over a classic algorithm described by Heckerman et al. [17]. Compared with the work of Linderman et al. [8], we further optimize the algorithm by replacing its hashing-based method for finding index by a direct mapping method and replacing the sum-based scoring function by a max-based scoring function. As a result, our improved algorithm is much faster and can be applied to larger networks than the algorithm in [8], which will be demonstrated by our experimental results in Section 8. Compared with the work of Tamada et al. [10] and the work of Nikolova et al. [12], our order-based sampling is more theoretically robust, since it can avoid being trapped in a local optimum thanks to traversibility of MCMC. Unlike the systems of Tamada et al. [10] and Nikolova et al. [12], we do not focus on extremely large networks. Our work is better suited for laboratories with limited computing resources that are interested in learning smaller networks.

3 Background

3.1 Bayesian Network

A Bayesian network G is a probabilistic graphical model representing a set of random variables and their conditional dependencies by a directed acyclic graph (DAG). The parent set π_i of a given node v_i is the set of nodes which have a directed edge to v_i. Each node v_i is associated with a probability distribution conditioned on its parent set, $P(v_i|\pi_i)$. The joint probability distribution of all the random variables in a Bayesian network can be written as a product of the conditional distributions for all the nodes:

$$P(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} P(v_i|\pi_i). \quad (1)$$

In this work, we focus on BNs composed of *binary* random variables, which has two states. Our methods can also be applied to BNs with multiple states. The only difference between learning two-state network and learning multiple-state network lies in the preprocessing subroutine. The problem here is to learn the BN structure from a set of experimental data.

3.2 Markov Chain Monte Carlo

BN learning is an NP-complete problem. Table 1 shows the total number of possible graphs and the total number of topological orders for different numbers of nodes. Note that all the topological orders for a given number of nodes correspond to all the permutations on those nodes. Thus, given n nodes, the number of topological orders is $n!$. The table shows that the number of possible graphs grows super-exponentially with the number of nodes. To reduce the complexity, greedy search [16], [17], [18], constrain-based methods [19], [20] and heuristic search [21], [22] are proposed. Compared with sampling methods, those approaches easily get trapped in local optimal solutions.

TABLE 1
The Number of Graphs and the Number of Topological Orders versus Different Numbers of Nodes

# of nodes	# of graphs	# of orders
4	453	24
5	29,281	120
10	4.7×10^{17}	3.6×10^{6}
20	2.34×10^{72}	2.43×10^{18}
30	2.71×10^{158}	2.65×10^{32}
40	1.12×10^{276}	8.16×10^{47}

Markov Chain Monte Carlo, a type of theoretically robust sampling method [23], is recently applied to efficiently solve the problem of learning BNs [24]. MCMC method is typically used to draw samples from a complex probability distribution. It achieves this by constructing a proper Markov chain such that its steady state distribution is equivalent to the probability distribution to be sampled. The result improves as more samples are drawn from the Markov chain.

One MCMC-based method proposed to solve the problem of learning BNs is the *graph* sampling, which explores the huge graph space for the best graph. Another method is to sample the *order* space, which consists of all topological orders for DAGs, and return the best order. This approach explores a much smaller space than the graph space. There is also order-graph sampling, which samples graphs for a sampled order [6]. In this work, we choose to optimize the approach that samples the order space because it requires fewer steps to converge than other methods and provides more opportunities to be paralleled on GPUs. Note that each DAG has at least one topological order, which we denote as $\prec$.

3.3 Scoring Metric

Learning BNs aims at finding a graph structure which best explains the data. We can measure each BN structure with a Bayesian scoring metric, which is defined as [17]:

$$P(G, D) = \prod_{i=1}^{n} P(v_i, \pi_i; D), \quad (2)$$

where D denotes the experimental data. $P(v_i, \pi_i; D)$ is known as the local probability and can be calculated as

$$P(v_i, \pi_i; D) = \gamma^{|\pi_i|} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ik})}{\Gamma(\alpha_{ik} + N_{ik})} \prod_{j=1}^{|v_i|} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}, \quad (3)$$

where γ serves as a penalty for complex structures [6], α is the hyperparameter for prior of Bayesian Dirichlet score, r_i is the number of different states of the parents set π_i, $|v_i|$ is the number of possible states of the random variable v_i, N_{ik} and N_{ijk} are obtained from the experimental data D [17], and Γ is the gamma function [25].

Since the local probability calculated by Equation (3) is very small, we perform the computation in the log-space, i.e., instead of calculating $P(G, D)$ and $P(v_i, \pi_i; D)$, we compute $\log P(G, D)$ and $\log P(v_i, \pi_i; D)$. We refer to $\log P(G, D)$ as the *score* of a BN and $\log P(v_i, \pi_i; D)$ as the *local score*. For simplicity, we also denote $\log P(v_i, \pi_i; D)$ as $ls(i, \pi_i)$.

When scoring an order $\prec$, the posterior probability of $\prec$ given the experimental data D, $P(\prec |D)$, has been shown to be proportional to the sum of all local probabilities $P(v_i, \pi_i; D)$ over all the graphs G that are consistent with the topological order $\prec$ [24]:

$$P(\prec |D) \propto \sum_{G \in \prec} \prod_{i=1}^{n} P(v_i, \pi_i; D),$$

which can be further efficiently calculated as

$$P(\prec |D) \propto \sum_{G \in \prec} \prod_{i=1}^{n} P(v_i, \pi_i; D) = \prod_{i=1}^{n} \sum_{\pi_i \in \Pi_\prec} P(v_i, \pi_i; D), \quad (4)$$

where $\Pi_\prec$ is the set of all parent sets that are consistent with the order $\prec$.

In order to reduce the overall complexity, we limit the maximal size of the parent set for a node to a constant s. In other words, we only consider graphs such that for each node v_i in the graph, it has at most s parents. This strategy was proposed by de Campos and Ji [26]. It is very important because it reduces the complexity of the scoring function from exponential to polynomial. Please note that both our serial (including baseline and optimized implementations) and parallel implementations are based on this constraint.

4 Bijection between Integers and k-Combinations

In this section, we illustrate two mapping algorithms that are important subroutines of our efficient implementation of the BN learning algorithm. The first algorithm maps a k-combination to a positive integer and the second one maps a positive integer to a k-combination. Here, a *k-combination* means a subset of k elements from a given set $S = \{1, 2, \ldots, n\}$. For example, the set $\{1, 2, 4\}$ is a three-combination of the set S. Since a k-combination does not contain duplicates and further, we could sort the elements in a k-combination in ascending order, therefore, for ease of exposition, we will represent a k-combination by a vector $(a_1, a_2, \ldots, a_k)$ such that $1 \le a_1 < a_2 < \cdots < a_k \le n$.

4.1 Mapping a k-Combination to an Integer

In our BN learning algorithm, each local score $ls(i, \pi_i)$ is used many times. To save the time of computing each local score (which requires calculation through Equation (3)), we pre-compute all the local scores and store them in a table, which we refer to as a *local score table*. Note that each local score $ls(i, \pi_i)$ is keyed by the combination of a node i and a parent set π_i. Previous investigators used a hash function to assign a random index based on the node and the parent set. However, we observed that we can sort all the $\binom{n}{k}$ k-combinations of the set $S = \{1, 2, \ldots, n\}$ in lexicographic order. Therefore, we do not need to store the local score for each node i and each parent set π_i in a random index given by the hash function. Instead, we can store the score in a position corresponding to the index of that k-combination in the lexicographic order of all k-combinations.

Algorithm 1. Procedure *findIndex*: it returns the index of a given k-combination $(a_1, \dots, a_k)$ with $1 \le a_1 < \cdots < a_k \le n$ in the lexicographic order of all k-combinations of the set $S = \{1, \dots, n\}$.

1: {Given two integers n and k and a vector $(a_1, \dots, a_k)$ denoting a k-combination, where $1 \le a_1 < \cdots < a_k \le n$.}
2: $index \Leftarrow 1$; $a_0 \Leftarrow 0$;
3: **for** $i = 1$ to k **do**
4: **for** $j = a_{i-1} + 1$ to $a_i - 1$ **do**
5: $index \Leftarrow index + \binom{n-j}{k-i}$;
6: **end for**
7: **end for**
8: **return** $index$;

We proposed a procedure *findIndex* to obtain the lexicographic order of a k-combination, which is shown in Algorithm 1. The basic idea to get the index of a k-combination is to count the number of k-combinations that are before the current one. We use an example to illustrate the algorithm. Suppose that we want to get the index of the three-combination $(3, 7, 10)$ of the set $S = \{1, 2, \dots, 12\}$. The combinations that are before $(3, 7, 10)$ in the lexicographic order can be divided into three categories:

(1) (b_1, b_2, b_3), with $1 \le b_1 \le 2$ and $b_1 < b_2 < b_3 \le 12$;
(2) $(3, b_2, b_3)$, with $4 \le b_2 \le 6$ and $b_2 < b_3 \le 12$;
(3) $(3, 7, b_3)$, with $7 < b_3 \le 9$.

To count the number of combinations belonging to Category 1, we further divide them into two sub-categories: $(1, b_2, b_3)$ with $1 < b_2 < b_3 \le 12$ and $(2, b_2, b_3)$ with $2 < b_2 < b_3 \le 12$. The former has $\binom{11}{2} = 55$ combinations and the latter has $\binom{10}{2} = 45$ combinations. Similarly, we divide Category 2 into three sub-categories: $(3, 4, b_3)$ with $4 < b_3 \le 12$, $(3, 5, b_3)$ with $5 < b_3 \le 12$, and $(3, 6, b_3)$ with $6 < b_3 \le 12$. The numbers of combinations contained in these three sub-categories are $\binom{8}{1} = 8$, $\binom{7}{1} = 7$, and $\binom{6}{1} = 6$, respectively. Finally, we divide Category 3 into two sub-categories: $(3, 6, 8)$ and $(3, 6, 9)$. The numbers of combinations contained in these two sub-categories are both 1. Then, the number of combinations before the three-combination $(3, 7, 10)$ is

$$55 + 45 + 8 + 7 + 6 + 1 + 1 = 123.$$

The index of $(3, 7, 10)$ is 124.

In the general situation, we split the set of combinations before the input combination $(a_1, \dots, a_k)$ into k categories. Assuming that $a_0 = 0$, the combinations in the ith category are of the form $(a_1, a_2, \dots, a_{i-1}, b_i, b_{i+1}, \dots, b_k)$ with $a_{i-1} + 1 \le b_i \le a_i - 1$ and $b_i < b_{i+1} < \cdots < b_k \le n$. Thus, the number of combinations in the ith category is

$$\sum_{j=a_{i-1}+1}^{a_i - 1} \binom{n-j}{k-i}.$$

The procedure *findIndex* shown in Algorithm 1 implements the above idea.

Experimental results demonstrated that using this procedure we can calculate the location for storing the local score for a given node index i and a parent set π_i much faster than using a hash function.

4.2 Mapping an Integer to a k-Combination

As we will show in Section 7, when we implement the BN learning algorithm on a GPU, we assign each thread to process one of the local scores stored in the memory. In order to distribute different tasks to different threads, we need to map each thread ID to a unique parent set. Therefore, we require a procedure that could map an integer to a k-combination. We proposed a procedure *findComb* for this purpose, which is shown in Algorithm 2. Given an integer l, the procedure returns the lth k-combination according to the lexicographic order of all k-combinations of the set $S = \{1, \dots, n\}$, without explicitly counting them one by one. This procedure can be thought as an inverse function of the procedure *findIndex*.

Algorithm 2. Procedure *findComb*: given an integer l, it obtains the lth k-combination in the lexicographic order of all k-combinations of the set $S = \{1, \dots, n\}$.

1: {Given three integers l, k, and n, returns a k-combination in vector form as $(a_1, a_2, \dots, a_k)$.}
2: $base \Leftarrow 0$;
3: **for** $i = 1$ to $k - 1$ **do** {Obtain a_i in the k-combination.}
4: $sum \Leftarrow 0$;
5: **for** $shift = 1$ to n **do**
6: **if** $sum + \binom{n-shift}{k-i} < l$ **then**
7: $sum \Leftarrow sum + \binom{n-shift}{k-i}$;
8: **else**
9: **break**;
10: **end if**
11: **end for**
12: $a_i \Leftarrow base + shift$;
13: {Update the parameters for obtaining the next a_i.}
14: $n \Leftarrow n - shift$; $l \Leftarrow l - sum$; $base \Leftarrow a_i$;
15: **end for**
16: $a_k \Leftarrow base + l$;
17: **return** $(a_1, a_2, \dots, a_k)$;

The procedure obtains each entry in the k-combination $(a_1, a_2, \dots, a_k)$ one by one from the first to the last. First notice that the index l of a combination with $a_1 = m$ satisfies that

$$1 + \sum_{j=1}^{m-1} \binom{n-j}{k-1} \le l \le \sum_{j=1}^{m} \binom{n-j}{k-1}.$$

This is because the first $\binom{n-1}{k-1}$ k-combinations are of the form $(1, b_2, \dots, b_k)$ $(2 \le b_2 < b_3 < \cdots < b_k \le n)$, the next $\binom{n-2}{k-1}$ k-combinations are of the form $(2, b_2, \dots, b_k)$ $(3 \le b_2 < b_3 < \cdots < b_k \le n)$, and so on.

Thus, the first entry a_1 should be the *largest* number m satisfying that $\sum_{i=1}^{m-1} \binom{n-i}{k-1} < l$. For example, for a given l, if we find that $\sum_{i=1}^{m^*-1} \binom{n-i}{k-1} < l \le \sum_{i=1}^{m^*} \binom{n-i}{k-1}$, then we can conclude that the entry a_1 of the lth combination should be m^*, i.e., $a_1 = m^*$.

In order to get the second entry a_2, it is equivalent to obtaining the $(l - sum)$th $(k-1)$-combination of the set $\{1, 2, \dots, n - a_1\}$, where $sum = \sum_{i=1}^{a_1 - 1} \binom{n-i}{k-1}$. This can be proved as follows. Suppose that the lth k-combination is

$$(a_1, a_2, \dots, a_k) = (a_1, a_1 + a'_2, \dots, a_1 + a'_k).$$

Since $a_1 < a_2 < \cdots < a_k \leq n$ by the definition of k-combination, we have $1 \leq a'_2 < \cdots < a'_k \leq n - a_1$. Thus, $(a'_2, a'_3, \ldots, a'_k)$ is a $(k-1)$-combination of the set $\{1, 2, \ldots, n - a_1\}$. Thus, the original problem reduces to finding a $(k-1)$-combination of the set $\{1, 2, \ldots, n - a_1\}$. Since the first $(k-1)$-combination $(1, 2, \ldots, k-1)$ is mapped to the k-combination $(a_1, a_1 + 1, \ldots, a_1 + k - 1)$, which is the $(\sum_{i=1}^{a_1-1}\binom{n-i}{k-1} + 1)$th k-combination. Thus, the reduced problem is to find the $(l - sum)$th $(k-1)$-combination of the set $\{1, 2, \ldots, n - a_1\}$, where $sum = \sum_{i=1}^{a_1-1}\binom{n-i}{k-1}$.

Therefore, in order to get a_2, we first get the *largest* number m such that $\sum_{i=1}^{m-1}\binom{(n-a_1)-i}{(k-1)-1} < (l - sum)$. Let that number be m^*. Then, we can obtain $a_2 = a_1 + m^*$. We compute all the remaining entries in the k-combination in a similar way.

Algorithm 2 implements the above idea in an *iterative* way. We use an example to illustrate the algorithm. Assume that we want to get the eighth three-combination of the set $S = \{1, 2, 3, 4, 5\}$, which should be $\{2, 3, 5\}$. The input arguments to Algorithm 2 is $l = 8$, $k = 3$, and $n = 5$.

First, we get a_1. We go through the first iteration of the outer loop with $i = 1$ (Line 3). $shift$ is first set to 1. Line 6 gives

$$sum + \binom{n - shift}{k - i} = 0 + \binom{5-1}{3-1} = \binom{4}{2} = 6 < l = 8.$$

Thus, we set sum to 6 (Line 7) and increase $shift$ to 2. We evaluate Line 6 again, which gives

$$sum + \binom{n - shift}{k - i} = 6 + \binom{5-2}{3-1} = 6 + 3 > l = 8.$$

Thus, we jump out of the inner loop (ending at Line 11) and get $a_1 = base + shift = 2$ (Line 12). At this point, n, l, and $base$ are updated to $n - shift = 5 - 2 = 3$, $l - sum = 8 - 6 = 2$, and $a_1 = 2$, respectively (Line 14).

To get a_2, we go through the second iteration of the outer loop with $i = 2$. $shift$ is first set to 1. Line 6 gives

$$sum + \binom{n - shift}{k - i} = 0 + \binom{3-1}{3-2} = 2 = l.$$

Thus, we jump out of the inner loop (ending at Line 11) and get $a_2 = base + shift = 2 + 1 = 3$ (Line 12). At this point, n, l, and $base$ are updated to $n - shift = 3 - 1 = 2$, $l - sum = 2 - 0 = 2$, and $a_2 = 3$, respectively (Line 14).

Now, the outer loop is terminated. The algorithm finally sets $a_3 = base + l = 3 + 2 = 5$ (Line 16). As a result, the returned three-combination is $(2, 3, 5)$, which is expected.

Note that the application of this procedure is not just limited to our BN learning problem, it is also useful in many other applications, such as software testing [27]. It is inspired by the algorithm proposed in [28]. We made some revisions in order to make it fit for our application.

5 The BN Learning Algorithm

In this section, we discuss our algorithm. Its overall flow is shown in Fig. 1, while its pseudocode is shown in Algorithm 3. In the pre-processing procedure, we compute all local scores, store them in the global memory, and

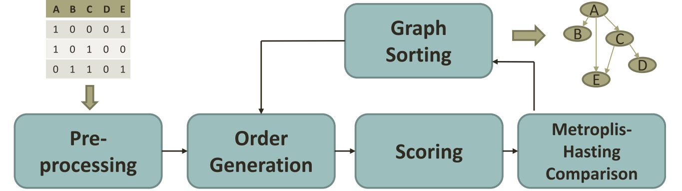


Fig. 1. The overall flow of the proposed BN learning algorithm.

randomly select an initial order. After the pre-processing step, we perform the MCMC iteration. In each iteration, we begin with generating a new order from the previous one by randomly selecting two nodes from the previous order and swapping them. Then we score the new order and accept or reject it by applying the Metropolis-Hastings rule [23]. The top graphs are then sorted and stored. After a specified number of MCMC iterations, we return the best graph we have obtained so far. The major subroutines of our algorithm are discussed in detail in this section.

Algorithm 3. The proposed BN Learning algorithm.

```
1:  Compute local scores ls(i, π_i) for all possible combinations
    of node v_i and parent set π_i with |π_i| ≤ s;
2:  Store all local scores in the local score table;
3:  Randomly select an initial order ≺_old;
4:  oldScore ⇐ −∞; globalBestScore ⇐ −∞;
5:  globalBestGraph ⇐ NULL;
6:  for iter = 1 to maxIterNums do
7:    Generate a new order ≺_new by swapping two nodes in the
      old order ≺_old;
8:    newScore ⇐ 0; newBestGraph ⇐ NULL;
9:    for each node v_i in the order ≺_new do
10:     maxLocalScore ⇐ −∞;
11:     for each parent set π_i consistent with the order ≺_new do
12:       Look up local score ls(i, π_i) in the local score table;
13:       if maxLocalScore < ls(i, π_i) then
14:         maxLocalScore ⇐ ls(i, π_i);
15:         bestParentSet ⇐ π_i;
16:       end if
17:     end for
18:     newScore ⇐ maxLocalScore + newScore;
19:     newBestGraph.connect(v_i, bestParentSet);
20:   end for
21:   Applying the Metropolis-Hastings rule on oldScore and
      newScore to decide whether the order ≺_new should be
      accepted;
22:   if the order ≺_new is accepted then
23:     ≺_old⇐≺_new; oldScore ⇐ newScore;
24:   end if
25:   if newScore > globalBestScore then
26:     globalBestScore ⇐ newScore;
27:     globalBestGraph ⇐ newBestGraph;
28:   end if
29: end for
30: return globalBestGraph;
```

5.1 Pre-Processing

As shown in Fig. 1, our learning algorithm begins with a pre-processing step, which includes initializing the order and computing all local scores (refer to Equation (3)). As we will show in Section 5.2, the scoring subroutine requires

computing each local score. Since the scoring subroutine is repeated in each MCMC iteration, we need to repeatedly compute each local score in each iteration. However, calculating each local score is time-consuming. Thus, instead of recomputing local scores each time when they are needed, we choose to compute local scores for all the possible combinations of the node and its parent set at the pre-processing stage. This strategy was also used in some previous works [7], [8], [9]. We note that if we do not limit the size of a parent set, then the number of all possible parent sets for the *last* node in any given order is 2^{n-1}, where n is the size of the network. To reduce the complexity, we limit the size of a parent set to a constant s, as proposed by de Campos and Ji [26]. Constraining the size of parent sets reduces the memory requirement from exponential to polynomial. In summary, in the pre-processing step, we compute the local score $ls(i, \pi_i)$ for each node v_i and each possible parent set π_i with size at most s, and store the result in the local score table. The position in the table to store a local score $ls(i, \pi_i)$ is obtained by applying the $findIndex$ procedure shown in Algorithm 1 on the node index i and the parent set π_i. Compared with the previous strategy that uses a hash function to find the position [7], [8], our method significantly reduces the time to calculate the position, as will be demonstrated in Section 8.1.

In the previous works [7], [8], bit vectors were used to generate every parent set consistent with a given order. However, our experimental results indicated that using bit vector representation is too slow: it is not scalable for networks with more than 30 nodes. This is because for the last node in an order, each of the $n-1$ nodes preceding it could be its parent. Therefore, even if we limit the maximal size of the parent sets to a constant, it is necessary to consider a total of 2^{n-1} bit vectors to filter out the parent sets of size larger than the limit. In our implementation, we generate the combination directly when the maximal size of a parent set is limited to a constant $s \ll n$. As a result, we only need to consider $\sum_{j=0}^{s} \binom{n-1}{j}$ potential parent sets for the last node, which is much smaller than 2^{n-1}.

5.2 Scoring

The scoring part is a major subroutine of our algorithm, which scores a given order. To effectively measure an order, we introduce a new scoring function, which is an optimization of the one proposed by Friedman and Koller [24]. It is well known that given an arbitrary topological order, there exist many graphs that are consistent with the order. We define the score of an order $\prec$ to be proportional to the maximal score over all the graphs that are consistent with the order, i.e.,

$$P(\prec, D) \propto \max_{G \in \prec} P(G, D). \quad (5)$$

Since the score $P(\prec, D)$ is very small, we perform the computation in the log-space. We apply logarithm on both sides of Equation (5) and obtain

$$\log P(\prec, D) \propto \max_{G \in \prec} (\log\, P(G, D)).$$

In what follows, we refer to $\log\, P(\prec, D)$ as the *score* of the order $\prec$. Based on Equation (2) and the relation that $ls(i, \pi_i) = \log\, P(v_i, \pi_i; D)$, we further have

$$\log\, P(\prec, D) \propto \max_{G \in \prec} \sum_{i=1}^{n} ls(i, \pi_i).$$

The computation of the right-hand side of the above equation is time-consuming, since we need to enumerate all the graphs that are consistent with the order. A much faster way to compute the right-hand side is based on the following theorem.

Theorem 1.

$$\max_{G \in \prec} \sum_{i=1}^{n} ls(i, \pi_i) = \sum_{i=1}^{n} \max_{\pi_i \in \Pi_\prec} ls(i, \pi_i),$$

where $\Pi_\prec$ is the set of all possible parent sets of the node v_i that are consistent with the order $\prec$.

Proof. Assume that a graph G_m consistent with the order $\prec$ gives the maximal value $\max_{G \in \prec} \sum_{i=1}^{n} ls(i, \pi_i)$. In other words,

$$G_m = \arg\max_{G \in \prec} \sum_{i=1}^{n} ls(i, \pi_i).$$

We use the notation $ls(i, \pi_i; G_m)$ to represent the local score for each node i in the graph G_m. It is clear that

$$\sum_{i=1}^{n} ls(i, \pi_i; G_m) \le \sum_{i=1}^{n} \max_{\pi_i \in \Pi_\prec} ls(i, \pi_i)$$

because each $ls(i, \pi_i; G_m)$ must be no greater than $\max_{\pi_i \in \Pi_\prec} ls(i, \pi_i)$.

Now assume that $\sum_{i=1}^{n} ls(i, \pi_i; G_m)$ is strictly less than $\sum_{i=1}^{n} \max_{\pi_i \in \Pi_\prec} ls(i, \pi_i)$. Then we can construct a new graph G'_m, where the parent set of node i is the one gives the value $\max_{\pi_i \in \Pi_\prec} ls(i, \pi_i)$. Then $\sum_{i=1}^{n} ls(i, \pi_i; G'_m)$ is greater than $\sum_{i=1}^{n} ls(i, \pi_i; G_m)$. This contradicts with our assumption that G_m is the graph with the highest $\sum_{i=1}^{n} ls(i, \pi_i)$. Thus, we have

$$\sum_{i=1}^{n} ls(i, \pi_i; G_m) = \sum_{i=1}^{n} \max_{\pi_i \in \Pi_\prec} ls(i, \pi_i).$$

Since G_m gives the maximal value $\max_{G \in \prec} \sum_{i=1}^{n} ls(i, \pi_i)$, we can conclude that

$$\max_{G \in \prec} \sum_{i=1}^{n} ls(i, \pi_i) = \sum_{i=1}^{n} \max_{\pi_i \in \Pi_\prec} ls(i, \pi_i).$$

□

Based on Theorem 1, we obtain an efficient way to compute $\log\, P(\prec, D)$ as follows

$$\log\, P(\prec, D) \propto \sum_{i=1}^{n} \max_{\pi_i \in \Pi_\prec} ls(i, \pi_i). \quad (6)$$

The scoring subroutine is shown at Lines $9 \sim 20$ in Algorithm 3. We refer to our scoring function as the *max-based scoring function*, and the scoring function shown in Equation (4) as the *sum-based scoring function*. We notice that a similar function to ours was previously proposed by Cooper and Herskovits [29]. However, it is only used in the post-processing part to find the best graph from the best order.

Our max-based scoring function is better than the sum-based scoring function in the following two aspects:

- Our scoring function only needs comparison and assignment operations, avoiding the time-consuming exponentiation and logarithm operations required by the sum-based scoring function. The runtime comparison in Section 8.1 confirms the shorter runtime of our max-based scoring function.
- Using the max-based scoring function, we do not need any post-processing to construct the best graph from the best order. However, we need this step if using the sum-based scoring function.

Our max-based scoring function produces similar results as the sum-based scoring function, which will be demonstrated in Section 8.2.

5.3 Order Generation and Metropolis-Hastings Comparison

At the beginning of each iteration, we generate a new order by randomly selecting two nodes v_i and v_j in the previous order and swapping them, i.e., changing the order $(v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n)$ to the order $(v_1, \ldots, v_j, \ldots, v_i, \ldots, v_n)$. Only the scores of those nodes that are between the node v_i and the node v_j, inclusively, need to be recalculated.

At the end of each iteration, we apply the Metropolis-Hastings rule to decide whether to accept the new order or not, as Linderman et al. did in [8]. This rule is the core of the MCMC method, which ensures the convergence and traversability of the MCMC method. By this rule, a new order is accepted with probability

$$p = \min\left\{1, \frac{P(\prec_{new}, D)}{P(\prec_{old}, D)}\right\},$$

where $P(\prec_{new}, D)$ and $P(\prec_{old}, D)$ are the scores of the new order and the previous order, respectively.

Since we actually compute the scores in the log space, the new order is accepted if

$$\log(u) < \log P(\prec_{new}, D) - \log P(\prec_{old}, D),$$

where u is a random number generated uniformly from the unit interval $[0, 1]$.

6 Interventions for Characterizing Pairwise Relationship

In this section, we present a way of adding interventions to characterize the prior knowledge on the dependency between a pair of nodes. In this paper, we use the phrases artificial interventions and prior knowledge interchangeably.

Assume that a function $p(i, m)$ represents the confidence of the prior knowledge on the existence of an edge from v_m to v_i, the joint probability of a graph becomes

$$P(G, D) = \prod_{i=1}^{n} \gamma^{|\pi_i|} \prod_{m \in \pi_i} p(i, m) \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ik})}{\Gamma(\alpha_{ik} + N_{ik})} \times \prod_{j=1}^{|v_i|} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}. \quad (7)$$

From the above equation, we can see that if the confidence of the prior knowledge $p(i, m)$ on the existence of an edge in the Bayesian network is large, then the probabilities of those graphs containing that edge will increase, and hence those graphs are more likely to be sampled. In log space, Equation (7) becomes

$$\log P(G, D) \propto \sum_{i=1}^{n} \left[ls(i, \pi_i) + \sum_{m \in \pi_i} \log p(i, m) \right], \quad (8)$$

where $ls(i, \pi_i)$ is the local score defined in Section 3.3. We call $\log p(i, m)$ the *pairwise prior function* (PPF) for the nodes v_i and v_m. It is also denoted as $PPF(i, m)$. Thus, Equation (8) becomes

$$\log P(G, D) \propto \sum_{i=1}^{n} \left[ls(i, \pi_i) + \sum_{m \in \pi_i} PPF(i, m) \right]. \quad (9)$$

With this general form of adding pairwise prior knowledge, we can meet different needs by applying different PPFs.

In our design, we provide an interface for users. It is an $n \times n$ matrix R, where n is the number of nodes in the graph. Each entry in matrix R is between zero and one. A value of $R(i, m)$ between 0 and 0.5 means that it is unlikely that there exists an edge from the node v_m to the node v_i; a value of $R(i, m)$ between 0.5 and 1 means that it is likely that there exists an edge from the node v_m to the node v_i; a value of $R(i, m)$ equal to 0.5 means that there is no bias on whether or not there exists such an edge from the node v_m to the node v_i. This interface provides a convenient way to specify the pairwise interventions. The actual PPF is a function on the values in the matrix R. Our experiment results show that

- when $R[i, m]$ approaches 1, a good choice of $PPF(i, m)$ is around 10;
- when $R[i, m]$ approaches 0.5, $PPF(i, m)$ should approach 0;
- when $R[i, m]$ approaches 0, a good choice of $PPF(i, m)$ is around -10.

Here 10 and -10 are chosen empirically to have a significant impact on the ultimate score of a graph.

Based on the above-mentioned requirements, we propose the following cubic polynomial to transform the value in the interface matrix R into the PPF:

$$PPF(i, m) = 100(R[i, m] - 0.5)^3. \quad (10)$$

7 GPU Implementation of the BN Learning Algorithm

In this section, we discuss the implementation of the proposed BN learning algorithm on a GPU.

7.1 The Architecture of the GPU

Fig. 2 shows the architecture of a typical GPU. The host, a CPU, assigns tasks to and collects results from the GPU. The GPU contains a number of blocks connected in the form of a grid. Each block supports $0 \sim 512$ threads. Each thread

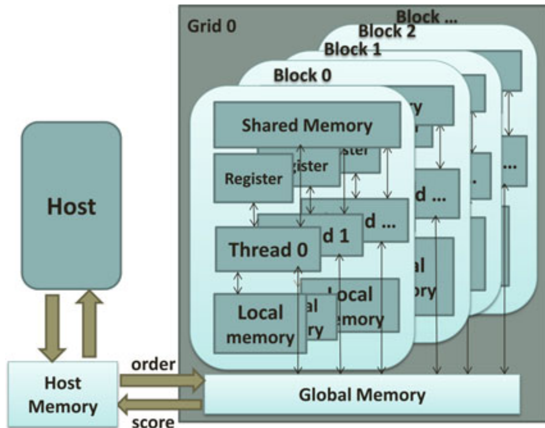


Fig. 2. The architecture of the GPU and its communication with the host CPU.

has a number of registers and a local memory. All the threads within a block can access the shared memory of that block. All the threads can also access the global memory of the GPU. As we will show in Section 7.2, the GPU implements the pre-processing and the scoring parts of our algorithm, since these parts can be parallelized across all the nodes (see Equation (6)). The remaining parts of our learning algorithm are handled by the CPU: it passes a new order to the GPU and gets the best graph and its score from the GPU, as shown in Fig. 2.

7.2 Parallel Implementation

The GPU executes the pre-processing and the scoring parts of the proposed BN learning algorithm. The mapping algorithms described in Section 4 play an important role here, which allow us to implement the BN learning algorithm on the GPU efficiently.

7.2.1 Parallel Pre-Processing

In the pre-processing subroutine, local scores of every possible parent set for each node should be calculated from data. It is time-consuming. Therefore, acceleration by GPUs is essential.

Since each thread has a thread ID and a block ID in the CUDA programming environment, we can assign a specific task to a thread based on its ID. The problem is how to let a thread know the parent sets that it needs to handle. It is equivalent to a combination indexing problem: given a set of n elements, we want to index all the combinations of the elements with *at most* s elements in a lexicographic order, so that given an arbitrary valid index we can easily get the corresponding combination. In other words, we need to index the collection of all 0-combinations, all 1-combinations, ..., and all s-combinations. The mapping algorithm described in Section 4.2 has already given us a way to index all k-combinations for a specific value k. We could apply that algorithm to index the collection of all k-combinations for all $k \leq s$. For convenience, we index these combinations in a descending order on the value k. Algorithm 4 shows the procedure that obtains a parent set corresponding to a given index. It applies the procedure $findComb$ shown in Algorithm 2.

Algorithm 4. Algorithm for each thread in the pre-processing subroutine.

1: {Given tid, bid, n, and $numParentSet$, where tid is the thread ID within the block, bid is the block's ID within the grid, n is the size of the Bayesian network, and $numParentSet$ is the total number of possible parent sets for each node.}
2: $id \Leftarrow bid * blocksize + tid$;
3: **if** $id < numParentSet$ **then**
4: $\quad comb \Leftarrow findComb(id, n-1)$;
5: $\quad$**for** $node = 1$ to n **do**
6: $\quad\quad parentSet \Leftarrow recover(comb, node)$;
7: $\quad\quad localScore[numParentSet * node + id]$
8: $\quad\quad\quad \Leftarrow calScore(parentSet, node)$;
9: $\quad$**end for**
10: **end if**

In Algorithm 4, $localScore$ stores all local scores keyed by the combination of a node index and a parent set. $numParentSet$ refers to the total number of possible parent sets of each node. A parent set of a node v_i is a subset of the set $\{v_1, \ldots v_{i-1}, v_{i+1}, \ldots, v_n\}$. Given that we limit the parent set size to s, we have

$$numParentSet = \sum_{j=0}^{s} \binom{n-1}{j}.$$

The algorithm first calculates a thread's ID in a grid (Line 2). Only those threads with ID less than $numParentSet$ are used to calculate the local score (Line 3). It then calls $findComb$ to get the combination corresponding to the thread ID (Line 4). Given a node $1 \leq node \leq n$, its parent set needs to be $recover$ed from the combination (Line 6), because $comb$ only points to the indices of the nodes in the set of potential parents of a node. The recovered parent set depends on both the combination and the node. For example, suppose that a graph has 6 nodes $v_1, v_2, \ldots, v_6$. Let us consider the parent set for node v_4. It is a subset of the set $\{v_1, v_2, v_3, v_5, v_6\}$. If $comb[] = \{1, 4, 5\}$, then it refers to the first, fourth, and fifth node in the set $\{v_1, v_2, v_3, v_5, v_6\}$. The corresponding parent set should be $recover$ed as $\{v_1, v_5, v_6\}$. Finally, the local score is computed according to Equation (3) (Lines 7 and 8).

7.2.2 Parallel Scoring

In the scoring subroutine, threads have to get the local scores from the local score table and return the best local score. The procedure for each thread is shown in Algorithm 5.

In this algorithm, we first call the function $findComb$ to obtain the indices combination $comb$ that the current thread is in charge of. Then, $comb$ is $recover$ed into the actual parent set $parentset$. Further, $parentset$ is $transform$ed into another indices combination $comb2$, which is the indices combination of the $parentset$ in the set that includes all the nodes except $node$ itself. From $comb2$, the function $findIndex$ produces the index of the local score corresponding to the parent set and the node, which is further fetched into the shared memory.

As an example, consider a graph of 10 nodes $v_1, \ldots, v_{10}$. Assume that in a specific order we are sampling, three

nodes v_2, v_6, v_9 proceed the node v_5. We want to fetch each local score corresponding to the node v_5 and a compatible parent set. By calling $findComb()$, the fifth thread gets $comb[] = \{1, 2\}$, which is $recover$ed into the parent set $\{v_2, v_6\}$ because nodes v_2 and v_6 are the first and second node proceeding the node v_5 in the order, respectively. The question is where in the local score table the thread should fetch the local score corresponding to the node v_5 and the parent set $\{v_2, v_6\}$. Note that the local score table stores all the possible parent sets of v_5, which are subsets of the set $A = \{v_1, v_2, v_3, v_4, v_6, v_7, v_8, v_9, v_{10}\}$. By $transforming$ the parent set $\{v_2, v_6\}$ into the indices in the set A, we get $\{2, 5\}$ because nodes v_2 and v_6 are the second and the fifth node in the set A, respectively. After passing $\{2, 5\}$ to function $findIndex()$, the algorithm returns the index of the local score the thread should fetch.

Algorithm 5. Algorithm for each thread in the scoring subroutine.

1: {Given $sharem$, which is the shared memory within each block, and $posN$, which is the number of nodes preceding $node$.}
2: $id \Leftarrow bid * blocksize + tid$;
3: **if** $id < numParentSet$ **then**
4: $comb \Leftarrow findComb(id, posN)$;
5: $parentset \Leftarrow recover(comb, node)$;
6: $comb2 \Leftarrow transform(parentset)$;
7: $index \Leftarrow findIndex(comb2, k)$;
8: $sharem[tid] \Leftarrow localScore[index]$;
9: **end if**
10: Find the best score and the best parent set within a block using a reduction algorithm;

One useful feature of GPUs is coalescent memory accesses. If consecutive threads access the memory with consecutive addresses, the accesses can be coalesced into a single access. The feature saves a lot of bandwidth and makes programs more parallelizable. It is obvious that Algorithm 4 can be perfectly coalesced because the consecutive threads write to the consecutive addresses in the memory (Line 7 in Algorithm 4). However, it is not the case for Algorithm 5, because the parent set each thread accesses is pretty random.

7.2.3 Reduction Algorithm

After getting the local score, each thread stores the parent set in local memory within the thread and the local score in a shared memory within the block. We further need to find the best score and the best parent set among all choices stored in the shared memory. In order to do this efficiently, we modified a reducing algorithm mentioned in [30]. Of course, using larger shared memory, traditional reduction algorithm can work, too. Storing all parent set and the local scores requires s times larger shared memory (s is the size limit of the parent set, in our case, $s = 4$). However, shared memory for each block cannot satisfy such a demand. So we need other solutions without larger requirement of shared memory.

Each thread has kept its local best parent set and the corresponding local best score. The problem is to pick the highest score among all the local best scores as well as its corresponding parent set. We have to recover its original position during a highly dynamic process. Our solution is described as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-3	-5	-9	-1	-11	-25	-4	-7	-35	-2	-21	-11	-7	-9	-2	-10
-3	-2	-9	-1	-7	-9	-2	-7	0	9	10	3	12	13	14	7
-3	-2	-2	-1	0	1	14	3								
-2	-1	14	3												
-1	3														

Fig. 3. An illustration of the reduction algorithm to find the highest score in the shared memory.

Before explaining the solution, we have to define an effective part (i) of a shared memory is the only part of memory ($0 \sim i$) that have to be written. For every reduction, higher scores are moved to the left half of the effective part. At start, the effective part is all of the shared memory ($i = block_size$). It reduces by half at the end of each reduction ($i/ = 2$). In the mean time, the original position of higher scores are stored in the right half of the effective part. Repeating this procedure on and on until the effective part converges into 1 ($i == 1$). The best score is moved to the leftmost of the shared memory.

We then have to keep track of the ID of the original thread that gives better value. That is, recovering the original position of the best score. Since we store the positions of the higher scores on the left half of the effective part at each reduction stage, the original position can be back traced accordingly.

An illustration of the algorithm is shown in Fig. 3. Assume that a shared memory has 16 entries. We want to move the highest score to the entry 0 of the array and record the ID of the thread that gives the highest score in entry 1. In the first reduction, thread 0 compares its value with the value in entry 8. Then, thread 0 assigns entry 0 of the shared memory with value -3 and entry 8 with 0, which is the ID of the thread giving the larger value -3.

In the second reduction, entry 2 of the shared memory has to be compared with entry 6 of the shared memory. Since -2 is larger than -9, -2 is stored in entry 2. Note that -2 is now from entry 6. However, the ID of the original thread that gives the value -2 is store in entry $6 + 8 = 14$, where 8 is the current number of threads involved. Then, we update entry 6 with the original thread ID by copying the value of entry 14 to entry 6. The total number of iterations required to get the best score among a total of n scores is $\log_2 n$. After obtaining the ID of the original thread that gives the highest score, we can fetch the best parent set from that thread.

8 Experimental Results

In this section, we show the experimental results. We performed experiments on both a general-purpose processor and a GPU. The GPP we used is a 2.00 GHz Intel Xeon E5-2620 processor. The GPU we used is an NVIDIA Tesla M2090 GPU with 6 GB GDDR5 RAM. The version of our NVIDIA CUDA compilation tool is V6.0.1. Note that our serial and parallel optimizations can be applied to any platform of CPUs and GPUs. Our GPU-based implementation is described in Section 7, with the pre-processing and the

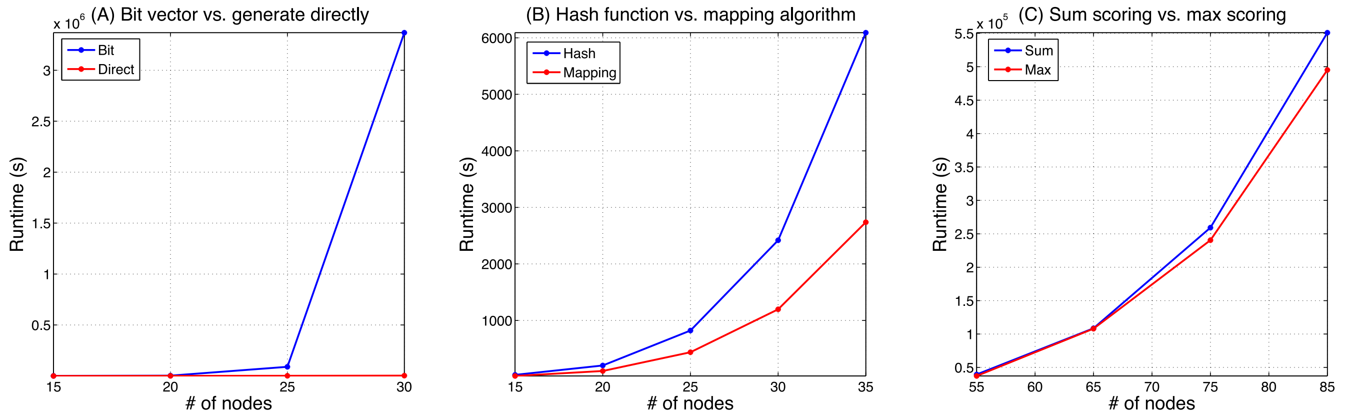


Fig. 4. (A) The runtimes for the implementation that uses bit vector to represent the parent sets and the implementation that generates valid parent sets directly. (B) The runtimes for the implementation that uses a hash function and the implementation that uses the mapping algorithm proposed in Section 4. (C) The runtimes for the implementation that uses the sum-based scoring function and the implementation that uses the max-based scoring function.

scoring parts of the BN learning algorithm performed on the GPU and the remaining parts of the algorithm performed on the CPU.

8.1 Speedup Study

In this set of experiments, we studied the speedup of our GPU implementation with the proposed algorithmic improvement. We set the maximal size of the parent set as 4, i.e., $s = 4$. The number of data is 600. The number of MCMC iterations is $10{,}000$. Note that once the number of MCMC iterations and the size of the graph are fixed, the runtime of our algorithm does not depend on the detailed value of the data set. Thus, in the experiments, we use randomly generated graphs of various sizes as the targets to be learned and generate the input data sets from these graphs.

We first studied the effect of our algorithmic improvement. Our algorithm is an MCMC-based algorithm for learning Bayesian networks. We compared it with the state-of-the-art MCMC-based algorithm for learning Bayesian networks described in [8], which applies bit vector to represent parent set, uses a hash table to store the local scores, and uses a sum-based function for calculating scores. Our proposed algorithm produces valid parent sets directly, uses the mapping algorithm proposed in Section 4.1 to obtain the local scores, and uses a max-based function for calculating scores. Both our algorithm and the algorithm from [8] are implemented on a general-purpose processor.

Fig. 4A compares the implementation that uses bit vector to represent the parent sets with the implementation that generates valid parent sets directly. Both of the implementations limit the size of parent sets. Both of them use a hash table to store the local scores and a sum-based function to calculate scores. However, as mentioned in Section 5.1, conventional bit vector-based method requires generating all possible parent sets in order to filter out the parent sets that are inconsistent with the limitation on the parent set size. We evaluated these two implementations on GPP using randomly generated graphs with $15 \sim 30$ nodes. For graphs with more than 25 nodes, the runtime of the conventional bit vector-based implementation increases dramatically. Our experiment of a 35-node graph has been running for eight days without even finishing preprocessing. Therefore, we have to limit our comparison to graphs with no more than 30 nodes. From the figure, we can see that the runtime of the bit vector-based implementation grows exponentially as the graph size increases. The largest speedup by generating valid parent sets directly is $1{,}393$.

Figs. 4B and C are the results of the algorithmic improvements. In Fig. 4B, we compared the GPP implementation that uses a hash table with the implementation that uses our mapping algorithm proposed in Section 4.1. Both of the two implementations generate valid parent sets directly and use the sum-based function for scoring. We used five randomly generated graphs with $15 \sim 35$ nodes. The result is shown in Fig. 4B. From the figure we can see that using the proposed mapping algorithm instead of a hash function reduces the total runtime. The largest acceleration rate is 2.22.

Fig. 4C compares the implementation that uses the sum-based scoring function (Equation (4)) with the implementation that uses the max-based scoring function (Equation (6)), using four randomly generated graphs with $55 \sim 85$ nodes. Both of the two implementations generate valid parent sets directly and use our proposed mapping algorithm to store local scores. Since the exponentiation and the logarithm operations consume much more time than the comparison and the assignment operations, the implementation that uses the max-based scoring function is faster than the implementation that uses the sum-based scoring function. The largest acceleration rate is 1.11.

Note that the speedup shown in Fig. 4A is gained from a trivial implementation improvement, while the speedups shown in Figs. 4B andC are gained from more sophisticated algorithmic improvements. We now analyze the overall speedup due to the algorithmic improvements. It is the product of the speedup using the mapping algorithm and the speedup using the max-based scoring function. From the trend shown in Figs. 4B and C, the acceleration rates due to the two algorithmic improvements do not decrease with the graph size. We argue that for graphs with over 100 nodes, the speedup achieved from each algorithmic improvement is no less than the largest speedup shown in the figure. Thus, the speedup using the mapping algorithm and the speedup using the max-based scoring function are at least 2.22 and 1.11, respectively, for graphs with over 100 nodes. The overall speedup due to algorithmic improvements is at least $2.22 \times 1.11 = 2.46$.

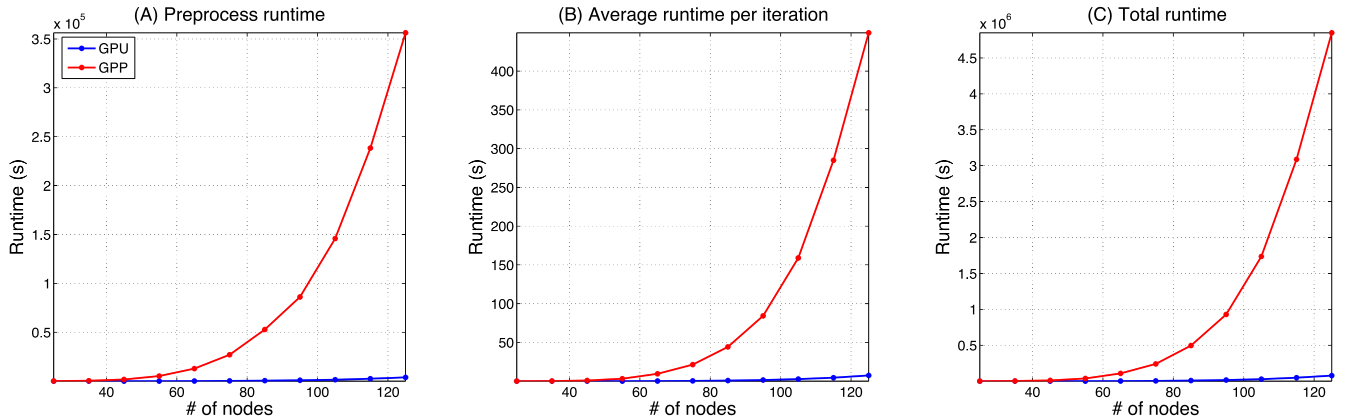


Fig. 5. The pre-processing runtimes, the average runtimes per iteration, and the total runtimes of both the GPP-based implementation and the GPU-based implementation.

We now study the speedup when further implementing the optimized algorithm using a GPU. Fig. 5 shows the result. The optimized algorithm includes all the techniques we have proposed, i.e., generating the valid parent sets directly, using the proposed mapping method to store the scores, and using the max-based scoring function. In the figure, we compare the runtimes of both the GPP-based implementation and the GPU-based implementation of the optimized algorithm for different network sizes. Fig. 5A plots the pre-processing runtimes, Fig. 5B plots the average runtimes per iteration, and Fig. 5C plots the total runtimes, which includes both the preprocessing runtime and the MCMC iteration runtime. The figure indicates that we have achieved a significant speedup using a GPU.

The detailed total runtimes of both the GPP-based implementation and the GPU-based implementation of the optimized algorithm, together with the acceleration rates of the latter implementation over the former implementation, are listed in Table 2. We achieved at least $58\times$ speedup for graphs with over 100 nodes. For graphs with 125 nodes, the GPP-based implementation needs approximately eight weeks while the GPU-based implementation only needs less than one day.

As shown in Table 2, when the optimized algorithm is implemented on the GPU, we obtained an additional speedup of at least 58 for graphs with over 100 nodes. In summary, we have achieved a speedup of at least $2.46 \times 58 = 143$ for graphs with over 100 nodes through both algorithm and GPU enhancements.

TABLE 2
The Runtimes of the GPP-Based Implementation and the GPU-Based Implementation and the Speedups of the GPU-Based Implementation Over the GPP-Based Implementation

# of Nodes	GPP runtime (sec.)	GPU runtime (sec.)	Speedup
25	354.59	113.84	3.11
35	2,577.83	207.28	12.43
45	10,601.43	417.08	25.42
55	37,358.70	917.77	40.70
65	108,062.71	1,988.31	54.34
75	240,399.92	4,224.38	56.91
85	495,218.78	8,492.83	58.31
95	928,907.06	15,876.35	58.50
105	1,735,958.87	27,948.19	62.11
115	3,088,128.50	47,766.58	64.65
125	4,850,759.00	77,311.83	62.74

8.2 Accuracy Study

We also empirically study the accuracy of our algorithm. In order to quantify the accuracy, we used measurements called "specificity" (S_p) and "sensitivity" (S_n) described by Tamada et al. [10]. It is defined as

$$Sp = \frac{TP}{TP + FP}, Sn = \frac{TP}{TP + FN},$$

where TP (true positive) is the number of edges estimated correctly, FP (false positive) is the number of estimated edges that are not in the actual network, and FN (false negative) is the number of edges in the actual network but not estimated [31]. Based on the definition of specificity and sensitivity, it is clear that the more the number of edges estimated correctly and the fewer the number of edges estimated incorrectly, the higher the specificity and sensitivity are and the more accurate the final result is.

We performed experiments on a randomly generated graph with 25 nodes. The number of MCMC iterations is $10{,}000$. The number of experimental data is 600. The specificities and sensitivities for these two runs are shown in

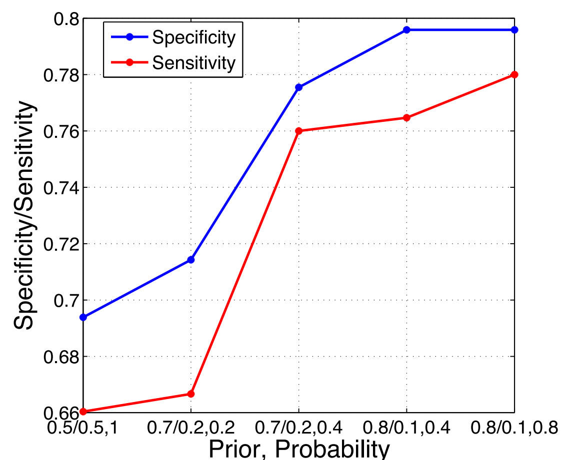


Fig. 6. The specificity and sensitivity of the result for learning a 25-node graph from 600 observed data. We ran our algorithm with 10,000 MCMC iterations.

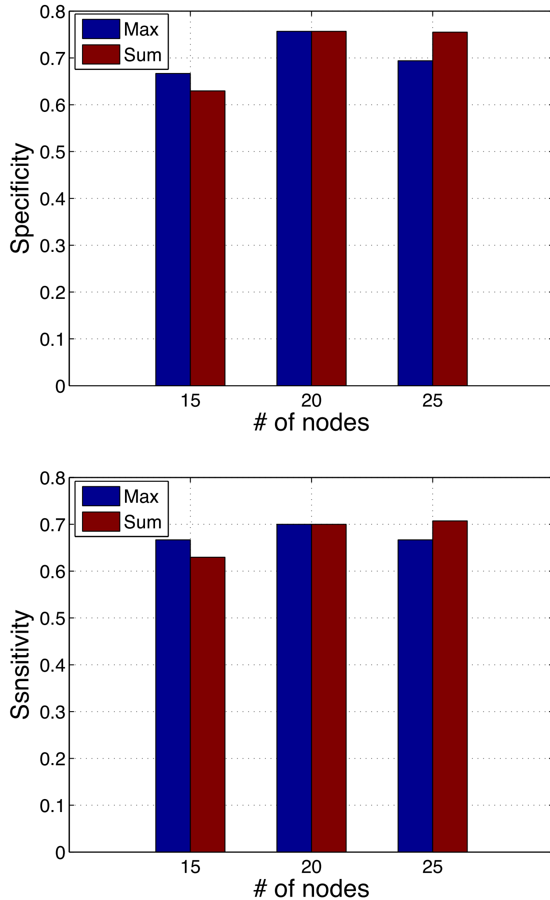


Fig. 7. The specificity and sensitivity comparison of sum-based scoring and max-based scoring.

Fig. 6. In the figure, the points from the left to the right are generated as follows: the first point is obtained without adding any prior knowledge on edges; the second point is obtained by assigning "interface" R value (refer to Section 6) 0.7 (0.2) with a probability of 0.2 to edges which are mistakenly removed (added) when learned without any prior knowledge; the third point is obtained by adding the same artificial interventions used in generating the second point but with a probability of 0.4; the fourth point is obtained by assigning "interface" R value 0.8 (0.1) with a probability of 0.4 to edges which are mistakenly removed (added) when learned without any prior knowledge; the fifth point is obtained by adding the same artificial interventions used in generating the fourth point but with a probability of 0.8. Note that the confidence on the priors knowledge added becomes stronger as we generate the points from the first to the last. The higher specificity and sensitivity with stronger prior suggests that our way of adding artificial interventions based on prior knowledge of the existence of edges is able to improve the accuracy of the final results.

We also compared the accuracy of our max-based scoring function with the traditional sum-based function. We performed experiments on three randomly generated graphs with 15, 20, and 25 nodes, respectively. The number of MCMC iterations is $10{,}000$. The number of experimental data is 600. Fig. 7 shows the specificities and sensitivities of the two scoring functions. The difference between two scoring functions is not significant. Both of the two scoring functions are able to generate high quality result networks. Taken into consideration that our max-based function is faster, we suggest the users to use the max-based scoring function instead of the sum-based scoring function.

TABLE 3
The Runtime Comparison between Algorithm 4 and the One-Score-Per-Thread Algorithm in Performing Pre-Processing

# Nodes	Algorithm 4 (sec.)	One-Score-Per-Thread Algorithm (sec.)
25	0.84	0.83
35	5.28	4.48
45	19.08	18.24

8.3 Study of the GPU Thread-Level Parallelism

As we discussed in Section 7.2.1, to perform pre-processing in parallel, we apply Algorithm 4 which lets each thread of GPU compute n local scores (Line 5 of Algorithm 4), where n is the number of the nodes in the network. The pre-processing subroutine can be further parallelized by letting each thread compute only one local score. We call this algorithm the one-score-per-thread algorithm. In this section, we experimentally compare the performances of Algorithm 4 and the one-score-per-thread algorithm.

The setup of this experimental study is the same as that of the experiment in Section 8.1: we choose the maximal parent set size as 4 (i.e., $s = 4$) and the data set size as 600. The only difference between the one-score-per-thread algorithm and Algorithm 4 is that the former uses one thread to compute one local score while the latter uses one thread to compute n local scores.

The runtimes of both algorithms for performing pre-processing are shown in Table 3. From the table, we can see that the runtimes of both implementations are very close. For the network with 45 nodes, the runtime of Algorithm 4 is only 4.6 percent slower than that of the one-score-per-thread algorithm. This means that Algorithm 4 has already fully utilized the GPU.

Further, the one-score-per-thread algorithm does not work for graphs over 45 nodes in our experiments. This is because in the CUDA programming model, the maximal number of blocks within a grid is fixed and the limitation is 65,535 blocks per grid for the GPU we use. As a result, the one-score-per-thread algorithm is not scalable for larger networks.

Given the above two facts, we use Algorithm 4 instead of the one-score-per-thread algorithm in our implementation of the parallel pre-processing.

8.4 Study of the Scalability over the Parent Set Size

As an important parameter in our algorithm, parent set size s affects the runtime of the algorithm. We study the scalability of our algorithm over the parent set size by measuring the runtime of learning a randomly generated 25-node graph using 10,000 MCMC iterations on GPP. The parent set size s ranges from 2 to 6. The implementation contains all the proposed algorithm optimizations. Fig. 8 shows how runtime scales over the parent set size. We notice that the

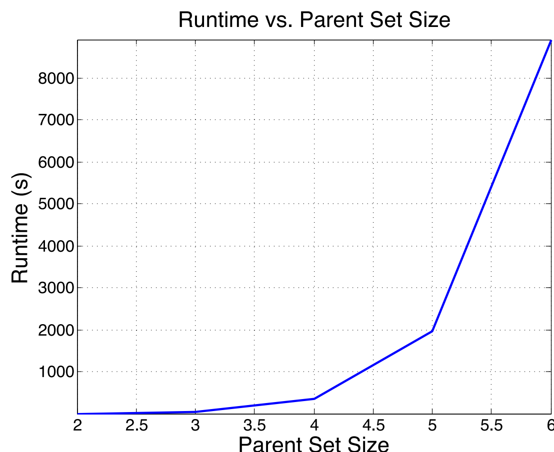


Fig. 8. How the runtime scales over the parent set size.

runtime increases quickly with parent size over 4. As a result, the algorithm is more suitable for sparse networks. Since our algorithm assumes a limit on the parent size, in its practical use, it is better that the user has an estimation of how sparse the network is and then choose a proper parent set size. In cases where the sparsity is not easy to estimate, one can also sweep the parent set size and compare the scores of the resultant graphs, because the scores are proportional to the likelihood of the graphs.

9 CONCLUSION

Learning Bayesian networks from data is a challenging computational problem. In this work, we have made two major improvements to the basic MCMC-based BN learning algorithm based on the work of Linderman et al. [8], which leads to a 2.46-fold acceleration. By further implementing our algorithm on a GPU, we achieved an additional 58-fold acceleration. In total, we have accelerated the algorithm by $143\times$. As a result, our system is applicable to learning BNs with up to 125 nodes, which is large enough for many biological applications. Besides, we add artificial interventions for characterizing the prior knowledge on pairwise interactions between nodes, which helps increase the accuracy of the learning results. Therefore, our system provides a very efficient tool for biologists. In our future work, we plan to expand the optimized algorithm to a GPU cluster. The complex operation of enumerating parent sets could be further accelerated on those more powerful hardware platforms.

ACKNOWLEDGMENTS

The first author Y. Wang finished the major part of this work during her study in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. The authors would like to thank Bob Adolf, David Brooks, Tianqi Chen, Glenn Holloway, Junjun Hu, Haopeng Liu, Yang Lu, Ye Pan, Jianghong Shi, Xiaoxing Wu, Xinan Wang, Song Xu, Ruoshi Yuan, and Tao Yin for their help. This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61472243, No. 61202026 and No. 61332001. The authors would like to thank the anonymous reviewers for their constructive comments which greatly improved the quality of this work. B. Yuan is the corresponding author.

REFERENCES

[1] A. Brazma, H. Parkinson, U. Sarkans, M. Shojatalab, J. Vilo, N. Abeygunawardena, E. Holloway, M. Kapushesky, P. Kemmeren, G. G. Lara, and A. Oezcimen, P. Rocca-Serra, S. A. Sansone, "Arrayexpress: A public repository for microarray gene expression data at the EBI," *Nucleic Acids Res.*, vol. 31, no. 1, pp. 68–71, 2003.

[2] P. Zimmermann, M. Hirsch-Hoffmann, L. Hennig, and W. Gruissem, "Genevestigator. arabidopsis microarray database and analysis toolbox," *Plant Phys.*, vol. 136, no. 1, pp. 2621–2632, 2004.

[3] D. J. Craigon, N. James, J. Okyere, J. Higgins, J. Jotham, and S. May, "Nascarrays: A repository for microarray data generated by NASC's transcriptomics service," *Nucleic Acids Res.*, vol. 32, no. suppl 1, pp. D575–D577, 2004.

[4] J. M. Irish, N. Kotecha, and G. P. Nolan, "Mapping normal and cancer cell signalling networks: Towards single-cell proteomics," *Nature Rev. Cancer*, vol. 6, no. 2, pp. 146–155, 2006.

[5] D. M. Chickering, "Learning Bayesian networks is NP-complete," in *Learning from Data*. New York, NY, USA: Springer, 1996, pp. 121–130.

[6] B. Ellis and W. Wong, "Learning causal Bayesian network structures from experimental data," *J. Am. Statist. Assoc.*, vol. 103, no. 482, pp. 778–789, 2008.

[7] N. Asadi, T. Meng, and W. Wong, "Reconfigurable computing for learning Bayesian networks," in *Proc. 16th Int. ACM/SIGDA Symp. Field Program. Gate Arrays*, 2008, pp. 203–211.

[8] M. Linderman, R. Bruggner, V. Athalye, T. Meng, N. B. Asadi, and G. Nolan, "High-throughput Bayesian network learning using heterogeneous multicore computers," in *Proc. 24th ACM Int. Conf. Supercomput.*, 2010, pp. 95–104.

[9] N. B. Asadi, C. Fletcher, G. Gibeling, E. Glass, K. Sachs, D. Burke, Z. Zhou, J. Wawrzynek, W. Wong, and G. Nolan, "Paralearn: A massively parallel, scalable system for learning interaction networks on FPGAs," in *Proc. 24th ACM Int. Conf. Supercomput.*, 2010, pp. 83–94.

[10] Y. Tamada, S. Imoto, H. Araki, M. Nagasaki, D. S. Charnock-Jones, S. Miyano, "Estimating genome-wide gene networks using nonparametric Bayesian network models on massively parallel computers," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 8, no. 3, pp. 683–697, May 2011.

[11] Y. Tamada, S. Imoto, and S. Miyano, "Parallel algorithm for learning optimal Bayesian network structure," *J. Mach. Learn. Res.*, vol. 12, pp. 2437–2459, 2011.

[12] O. Nikolova and S. Aluru, "Parallel Bayesian network structure learning with application to gene networks," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–9.

[13] O. Nikolova, J. Zola, and S. Aluru, "Parallel globally optimal structure learning of Bayesian networks," *J. Parallel Distrib. Comput.*, vol. 73, no. 8, pp. 1039–1048, 2013.

[14] P. Ao, D. Galas, L. Hood, and X. Zhu, "Cancer as robust intrinsic state of endogenous molecular-cellular network shaped by evolution," *Med. Hypotheses*, vol. 70, no. 3, pp. 678–684, 2008.

[15] G. Wang, X. Zhu, L. Hood, and P. Ao, "From phage lambda to human cancer: Endogenous molecular-cellular network hypothesis," *Quantitative Biol.*, vol. 1, no. 1, pp. 32–49, 2013.

[16] D. M. Chickering, "Optimal structure identification with greedy search," *J. Mach. Learn. Res.*, vol. 3, pp. 507–554, 2003.

[17] D. Heckerman, D. Geiger, and D. Chickering, "Learning Bayesian networks: The combination of knowledge and statistical data," *Mach. Learn.*, vol. 20, no. 3, pp. 197–243, 1995.

[18] A. Moore and W.-K. Wong, "Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning," in *Proc. 20th Int. Conf. Mach. Learn.*, 2003, vol. 3, pp. 552–559.

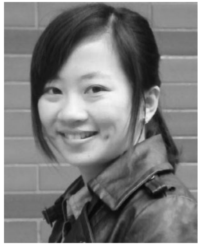
[19] G. Elidan and N. Friedman, "Learning hidden variable networks: The information bottleneck approach," *J. Mach. Learn. Res.*, vol. 6, pp. 81–127, 2005.

[20] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*. Cambridge, MA, USA: MIT Press, 2000, vol. 81.

[21] X.-W. Chen, G. Anantha, and X. Wang, "An effective structure learning method for constructing gene networks," *Bioinformatics*, vol. 22, no. 11, pp. 1367–1374, 2006.

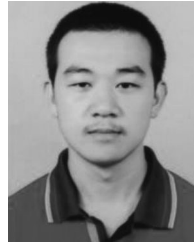
[22] M. Wang, Z. Chen, and S. Cloutier, "A hybrid Bayesian network learning method for constructing gene networks," *Comput. Biol. Chemistry*, vol. 31, no. 5, pp. 361–372, 2007.

[23] J. S. Liu, *Monte Carlo Strategies in Scientific Computing*. New York, NY, USA: Springer-Verlag, 2008.
[24] N. Friedman and D. Koller, "Being Bayesian about network structure: A Bayesian approach to structure discovery in Bayesian networks," *Mach. Learn.*, vol. 50, no. 1, pp. 95–125, 2003.
[25] W. Rudin, *Principles of Mathematical Analysis*. New York, NY, USA: McGraw-Hill, vol. 3, 1964.
[26] C. P. de Campos and Q. Ji, "Efficient structure learning of Bayesian networks using constraints," *J. Mach. Learn. Res.*, vol. 12, no. 3, pp. 663–689, 2011.
[27] *IEEE standard for software test documentation*, IEEE Std 829, 1998.
[28] D. E. Knuth, "The art of computer programming," *Sorting Searching*, vol. 3, pp. 426–458, 1999.
[29] G. Cooper and E. Herskovits, "A Bayesian method for the induction of probabilistic networks from data," *Mach. Learn.*, vol. 9, no. 4, pp. 309–347, 1992.
[30] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology.*, vol. 2, no. 4, 2007.
[31] T. Fawcett, "Roc graphs: Notes and practical considerations for researchers," *Mach. Learn.*, vol. 31, pp. 1–38, 2004.



Yu Wang received the BS degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China, and is working towards the PhD degree in the School of Engineering and Applied Sciences at Harvard University. Her current interests include accelerators, memory systems, and machine learning. She is the student member of the IEEE.

Weikang Qian received the BEng degree in automation at Tsinghua University, China in 2006, and the PhD degree in electrical engineering at the University of Minnesota in 2011. He is an assistant professor in the University of Michigan-Shanghai Jiao Tong University Joint Institute at Shanghai Jiao Tong University. His main research interests include electronic design automation and digital design for emerging technologies. In recognition of his doctoral research, he received the Doctoral Dissertation Fellowship at the University of Minnesota. One of his papers was nominated for the William J. McCalla Best Paper Award at the 2009 International Conference on Computer-Aided Design (ICCAD). He is the member of the IEEE.



Shuchang Zhang recieved the BS degree from the Department of Computer Science and Engineering, and is working towards the master's degree in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. He is interested in the geometric aspects of Markov Chain Monte Carlo algorithms.

Xiaoyao Liang received the PhD degree from Harvard University. He has ample industry experience working as a senior architect or IC designer at companies like NVIDIA, Intel, and IBM. He is a professor and the associate dean in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. His research interests include computer processor and system architectures, energy efficient and resilient microprocessor design, high throughput and parallel computing, general purpose graphic processing unit (GPGPU) and hardware/software co-design for the cloud and mobile computing. He is also interested in IC design, VLSI methodology, FPGA innovations and compiler technology. He is the member of the IEEE.

Bo Yuan received the bachelor degree from Peking University Medical School in 1983, China, the MS degree in biochemistry, and the PhD degree in molecular genetics from the University of Louisville, KY, in 1990 and 1995, respectively. He is currently a professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University (SJTU). Before joining SJTU in 2006, he was a tenure-track assistant professor with the Ohio State University (OSU), while serving as a co-director for the OSU Program in Pharmacogenomics. At OSU, he was the founding director for the OSUs genome initiative during the early 2000s, leading one of the only three independent efforts in the world (besides the Human Genome Project and the Celera company), having assembled and deciphered the entire human and mouse genomes. At SJTU, his research interests focus on biological networks, network evolution, stochastic process, biologically inspired computing, and bioinformatics, particularly on how these frameworks might impact the development of intelligent algorithms and systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.