International Journal of Grid and Distributed Computing
Vol. 11, No. 4 (2018), pp.113-126
http://dx.doi.org/10.14257/ijgdc.2018.11.4.10

Distributed Bayesian Network Learning Algorithm using Storm Topology

Fei Ding[1] and Yi Zhuang[2]

Nanjing University of Aeronautics and Astronautics, Nanjing, 210016, China
[1]felixding@nuaa.edu.cn, [2]zy16@nuaa.edu.cn

Abstract

The use of cloud computing enables access to high-capacity computing resources, which exploits parallel resources using parallel-machine learning algorithms such as Bayesian network structural learning algorithms. In this paper, we propose the distributed Bayesian network learning (DBNL) algorithm other than the MapReduce-based methods based on the Storm topology framework. By using the calculation for one DAG or CPDAG as the basic computation unit, we divide the searching space to the minimum scale to store B-spaces or E-spaces in parallel. In order to detect loops in the search space, we design the mechanics that ensures a state being sent to the same computing node if it has already been previously scored. We also use the hash code of each state as the key of a tuple in the grouping stage to assign processing tasks efficiently. We present the DBNL algorithm in detail and demonstrate its flexibility by giving four adapted variants based on hill-climbing, K2, TPDA and GES algorithms. We perform experiments that approximate the linear speedup. We conduct experiments using four real-life datasets to test the performance of DBNL in setups with different data size, numbers of processing instances and scoring criterion. The experimental results obtained indicate that the percentage of parallel parts is up to 93.28%, and the performance improvements are more significant with larger datasets.

Keywords: *Machine learning; Cloud computing; Bayesian networks learning; Storm topology*

1. Introduction

One of the fundamental issues to be addressed in data mining and machine learning involves discovering a model that explains a given set of data. The high complexity of many algorithms makes it challenging for us to learn models effectively. The structural learning of the Bayesian network, which is one of the most popular models used for inferring uncertain knowledge, is a specific example which has been proven to be NP-hard **Error! Reference source not found.**.

Despite the computationally difficult tasks, there have been several studies focused on speeding up algorithms, such as Bayesian network learning using parallel-learning methods. Sahin and Devasia applied the particle-swarm optimisation technique to parallelize the Bayesian learning algorithm by capitalising on its innately parallel behaviour **Error! Reference source not found.**. Yu *et al.*, demonstrated the PL-SEM algorithm **Error! Reference source not found.**, which exploits parallel resources by parallelizing an expectation step (E-step) inside the Expectation Maximisation (EM) part of the structural EM algorithm (SEM). Gou et al. proposed a parallel three-phase dependency analysis (P-TPDA) **Error! Reference source not found.** using conditional independency (CI) testing to combine local structures. Madsen *et al.*, proposed a parallel approach **Error! Reference source not found.** considering both horizontal and vertical

Received (January 7, 2018), Review Result (March 6, 2018), Accepted (March 12, 2018)

ISSN: 2005-4262 IJGDC

parallelization to speed up conditional independence testing during the Bayesian network learning process.

However, there is the potential for very large data sets to result in performance bottlenecks. In recent years, the rapid growth in the capacity of cloud computing has led to new opportunities for speeding up using high-level distributed programming paradigms **Error! Reference source not found.Error! Reference source not found.**. Yue *et al.*, used the MapReduce paradigm on Apache Hadoop to extend the classical score-and-search algorithm for learning Bayesian network from dynamically changing large scale data **Error! Reference source not found.**. Basak *et al.*, focused on applying the MapReduce framework to the Bayesian network parameter-learning algorithm, specifically the EM algorithm **Error! Reference source not found.Error! Reference source not found.**. Fang *et al.*, presented a distributed version of the traditional score-and-search algorithm using Hadoop MapReduce **Error! Reference source not found.**. Furthermore, Yue *et al.*, applied the MapReduce-based Bayesian network learning method discovering user similarities upon the massive social behavioral interactions **Error! Reference source not found.**. Arias *et al.*, proposed several MapReduce-based Bayesian network classifiers using the Apache Spark framework **Error! Reference source not found.**, aiming to minimise the data transfer through the cluster.

Most of the existing related works, including the works mentioned above, use the MapReduce paradigm by performing the MapReduce process iteratively. The intermediate results have to be saved in either distributed file system (such as Hadoop) or memory (such as Spark). It brings significant burden to the performance of algorithms with massive iterations. Besides, many previous works have parallelized existing algorithms based on a specific learning method at a relatively high level without giving much consideration to the issue of storage. The use of multiple computing resources could be further exploited by redesigning a storage method which is suitable for learning the Bayesian network in parallel, and it is suitable for storing massive data. Furthermore, some existing approaches do not parallelize completely, and are therefore limited to particular parameters, such as the number of attributes in a Bayesian network. The researches for solving these problems are highly valuable to the field of cluster computing and machine learning.

The commonly used Bayesian network learning algorithms could not be straightforwardly adapted to the parallel versions due to the challenges of parallel storage for massive data, loop detection in state space and parallel task assignment. In this paper, we propose the distributed Bayesian network learning (DBNL) algorithm, which is a non-serial and parallel algorithm (NSPA) for searching the B-space and E-space. The novelty of this study lies in the new approach of parallelizing Bayesian network learning other than the MapReduce-based methods with high speedup. In order to exploit the advantages of parallelizing iterative algorithms using cloud computing framework, we use the generation of candidates states and computation of the score for only one Bayesian network, rather than a subset of states in searching space, as the basic calculation units to maximise parallelism. By dividing learning processes into fine-grained tasks utilising a state-of-the-art elastic-stream processing framework such as Apache Storm **Error! Reference source not found.**, the capacity of DBNL for massive sample data is enhanced so that it could achieve the approximation of linear speedup. We also focus on the flexibility of the DBNL algorithm to enable ease of adaptation to various Bayesian learning algorithms and score criterion to a distributed version.

The contents of the article are as follows. First, we present an overview of the architecture and notations of the DBNL algorithm in Section 2. Afterwards, we discuss the illustration of the initial task, searching task, scoring task and output task. More specifically, in Section 2.4, we explore in detail the issues related to parallel storage for massive data, loop detection in state space and task assignment. In Section 4, we demonstrate a comprehensive experimental setup, and show results

obtained for four large-scale real-life datasets as well as up to six computing nodes. Finally, in Section 5, we summarise the DBNL algorithm.

2. Distributed Bayesian Network Learning Algorithm

In this section, we introduce the distributed Bayesian network learning algorithm (DBNL) by illustrating the algorithm architecture, initial task, searching task, scoring task and output task, respectively. The challenges of parallel storage for massive data, loop detection in state space and task assignment are discussed in detail in Section 2.4. Then, we analyse the parallelism and speedup of the DBNL algorithm.

2.1. Algorithm Architecture

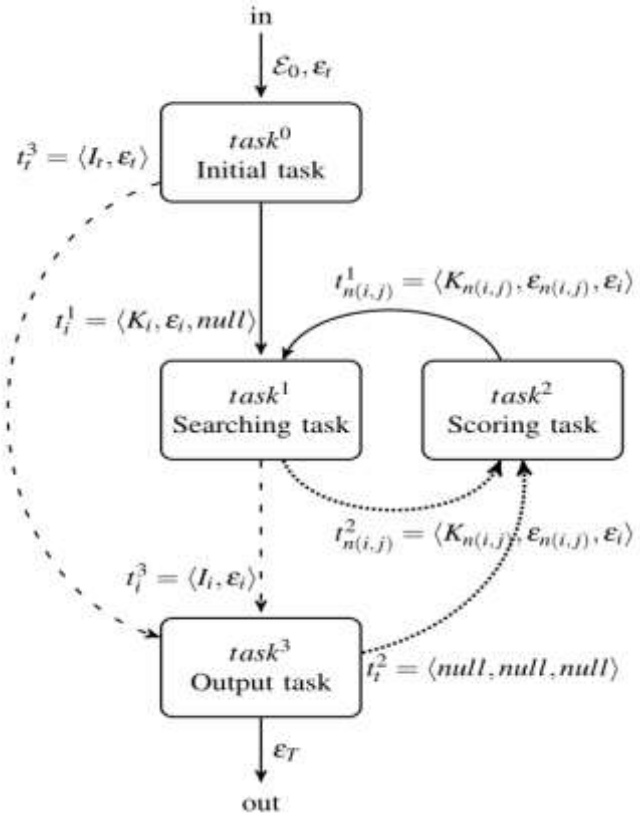

Figure 1. Tasks and Streams of the DBNL Algorithm

The DBNL algorithm is an NSPA that is employed to search graph spaces, particularly for learning Bayesian network structures. In order to maximise the speedup of the process of learning Bayesian networks, we divided the procedure into four tasks so that they can be executed simultaneously. The four tasks comprising the DBNL algorithm are the initial task, searching task, scoring task and output task, and these are denoted by $task^0$, $task^1$, $task^2$ and $task^3$, respectively. Each of the four types of tasks is a procedure that can be executed in a distributive manner using several processing instances in a cluster. The initial task initialises the tuples of the initial states of the state space. The searching task produces a sequence of operators which the algorithm should perform to attain a certain state. The scoring task calculates the score of a certain state according to a scoring criterion. The output task collects the scored states in the state space before determining

and terminating the algorithm. As illustrated in

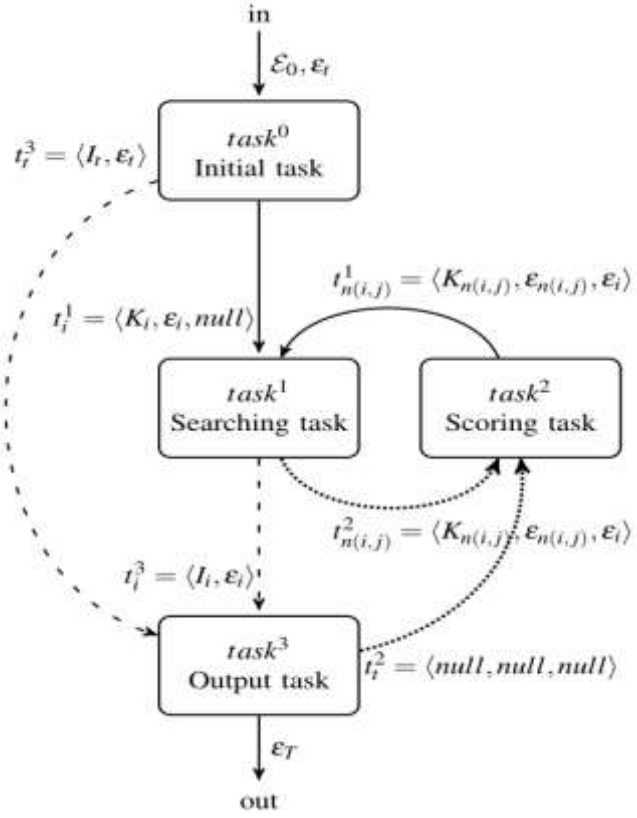

Figure **1.** , each task accepts the stream consisting of a particular type of tuple of data from another task.

Table 1. Description of Notations

Notations	Description
ε_i	the i_{th} state in E, ε_i =<G_i/$P^C{}_i$, $I_{p(i)}$, s_i>
t_i^k	the tuple corresponding to ε_i which is sent to $task^k$
$\gamma_{i,j}$	the operator that transfers ε_i to $\varepsilon_{n(i,j)}$
$\varepsilon_{n(i,j)}$	the j_{th} neighbour state of ε_i
$\varepsilon_{p(i)}$	the parent state of ε_i
ε_t	the state used for determining algorithm termination
K_i	the hash key of ε_i
I_i	the unique identifier of t_i^1 and t_i^2 corresponding to ε_i

As shown in Table 1, the tuple corresponding to ε_i in the stream flowing to $task^k$ is denoted by t_i^k, where ε_i is the i_{th} state in the set of all possible states E. The sequence of initial states from which the searching algorithm starts is denoted by E_0. A state ε is a triple of three elements: the Bayesian network model, such as a DAG G or a CPDAG P^C, the identifier of the parent state of ε and the score of ε according to a certain criterion. The operator $\gamma_{i,j}$ can be performed on ε_i to transfer to the j_{th} neighbour state of ε_i, which is denoted by $\varepsilon_{n(i,j)}$. Operators are different for DAG and CPDAG. For DAGs, the commonly used operators are adding, deleting and reversing a directed edge **Error! Reference source not found.**. For an equivalence class, the six permitted operators are InsertU, DeleteU, InsertD, DeleteD, ReverseD and MakeV **Error! Reference source not found.**. Although a state may be transferred from more than one state when exploring the search

space, only one of its parent states is recorded in the searching tree. The exclusive parent state of ε_i is denoted by $\varepsilon_{p(i)}$. Note that there is one of the neighbour states $\varepsilon_{n(i,j)}$ which is equal to the parent state $\varepsilon_{p(i)}$, namely one which contains the same Bayesian network as that of $\varepsilon_{p(i)}$, if and only if operators $\gamma_{i,j}$ and $\gamma_{j,i}$ both exist. For each of the state ε_i, the algorithm generates a hash key, which is denoted by K_i, to utilise the stream-grouping mechanism of cloud-computing platforms. For each of the tuples corresponding to ε_i, which is sent to a searching task or scoring task, the algorithm creates a unique identifier, which is denoted by I_i. Tuples t_i^1 and t_i^2 contain the same states ε_i, and hence, they use the same identifier. In practice, the identifier may be implemented as an integer or a string. The functions of generating hash keys and identifiers will be explained in more detail later in Section 3.3.

2.2. Initial Task

The initial task shown in Algorithm 1 is the first procedure in the algorithm which accepts the set of initial states, E_0, from which the searching starts, and the termination state ε_t is used for determining when to terminate the algorithm. According to different applied problems, the set of initial states E_0 differs. Because the termination condition of the algorithm varies, the termination state ε_t is optional. The degree of parallelism (DOP) of $task^0$ is one. In other words, there is only one processing instance in the cluster that executes the initial task.

Algorithm 1. Initial task

cons I_t: a constant for identifying the termination state
procedure INIT(ε_0,ε_t)
  for ε_i in E_0 **do**
    $\varepsilon_i.parent$:= *null*
    $\varepsilon_i.score$:= -∞
    send tuple t_i^1 = < K_i, ε_i, *null* > to searching task
  end for
  broadcast tuple t_t^1 = <*null*, ε_t, *null* > to scoring task
  send tuple t_t^3 = < I_t, ε_t> to output task
end procedure

First, the initial task assigns a null value and negative infinity to the parent identifier and score of each state in E_0, respectively. Second, for each initial state, a tuple is constructed and sent to the searching task. The first element in a tuple is the hash key of ε_i, and the second elements represent the currently processed Bayesian network. The third element in t_i^1 represents the parent state of ε_i, namely $\varepsilon_{p(i)}$. Because no states precede the initial states in E_0, the third elements sent in the initial task are all null values. Finally, the constant identifier of the termination state I_t maintained by the initial task is sent with the termination state ε_t to the output task to determine when to terminate the algorithm according to a certain convergence criterion. The constants in the algorithm are determined and coded as constant variables. Therefore, the processing instances do not really store the constants.

2.3. Searching Task

To explore the B-space or E-space, the searching task (shown in Algorithm 2) determines which Bayesian network structures should be modified and scored in the next iteration. The search task is initiated once it receives a tuple from the initial task. The searching task also receives streams from the scoring task with the same format. The DOP of the searching task could be set to be one or more accordingly. The assignment strategy of the searching task is the same as that of the scoring task, which will be explained later in Section 2.4.

Algorithm 2. Searching task

var $task^1.counter$: an integer counting the searched states
procedure SEARCH($t_i^1 = <K_i, \varepsilon_i, \varepsilon_{p(i)}>$)
  $task^1.counter$ increases by 1
  generate the identifier I_i according to $task^1.counter$
  send tuple $t_i^3 = <I_i, \varepsilon_i>$ to output task
  for $\gamma_{i,j}$ in $Operator(\varepsilon_i)$ $(1 \leq j \leq |Operator(\varepsilon_i)|)$ **do**
    $\varepsilon_{n(i,j)} := Transition(\varepsilon_i, \varepsilon_{n(i,j)})$
    $\varepsilon_{n(i,j)}.parent := I_i$
    if $\varepsilon_{p(i)} = null$ or $\varepsilon_{p(i)} \neq \varepsilon_{n(i,j)}$ **then**
    send tuple $t_{n(i,j)}^2 = <K_{n(i,j)}, \varepsilon_{n(i,j)}, \varepsilon_i>$ to scoring task
    end if
  end for
end procedure

First, the variable $task^1.counter$, which is maintained by each of the computing nodes which runs the searching task, is increased by one. The initial value of $task^1.counter$ is zero. $task^1.counter$ is used for generating identifiers I_i for subsequent tuples. The identifier is composed of the id of the processing instance and the number of states of the current instance which are already searched. The identifier is not generated immediately after a tuple is created, but it is calculated after the tuple is received by a searching task. Third, the key-value pair $<I_i, \varepsilon_i>$ is sent to the output task to determine whether to terminate the algorithm. Then, the function *Operator* is performed on the state ε_i. The function *Operator* establishes the operators which are valid to perform on a certain state. It returns an ordered set of operators $(\varepsilon_{n(i,1)}, \ldots, \varepsilon_{n(i,m)})$. The order of operators determines which heuristics strategy is used by the DBNL algorithm, and this will be explained later in Section 4. The searching task then generates new states using the function *Transition*, which returns the neighbour state after performing operator $\gamma_{i,j}$ on ε_i. The parent's id in each $\varepsilon_{n(i,j)}$ is assigned with the identifier I_i. The searching task sends only the states which are not initial states and not equal to ε_i's parent state.

2.4. Scoring Task

Before illustrating the scoring task, it is helpful to discuss three challenging issues in distributed learning Bayesian networks. The first one involves how to store and maintain a data structure for state space in an effectively manner. The second issue involves the detection of loops in the state space in a distributed scenario. The third problem involves assigning tasks to computing nodes appropriately.

One of the most straightforward methods employed to storing state space in a distributive manner is to maintain the whole searching tree in each of the computing nodes. However, there are obvious drawbacks in that it requires substantial memory to save states and computing time to synchronise the searching trees. The DBNL algorithm saves only a subset of the state space in a list (denoted $task^2.L$) on each computing node which runs scoring tasks. The whole state space is partitioned into x subsets, where x is the number of instances which run the scoring task. $task^2.L$ saves only those states which are already scored by the current processing instance. The concatenation of all lists contains information of the whole searched state space. By using this approach, the searching tree is stored and maintained in a distributive manner.

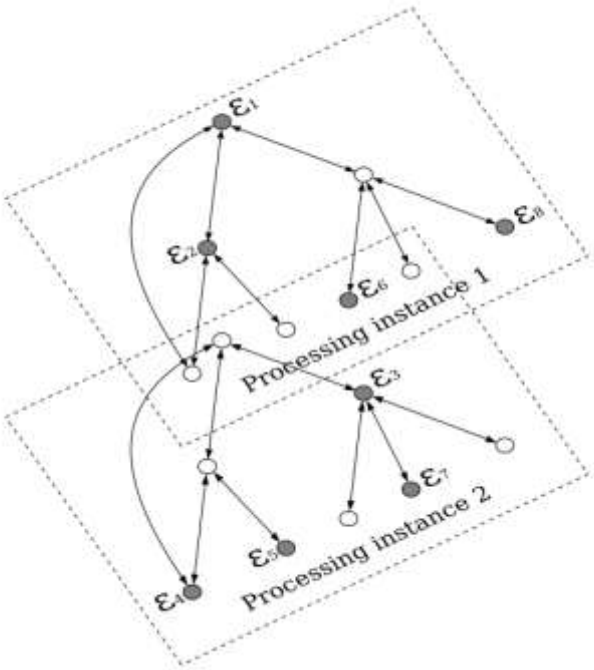

Figure 2. Save State Space Distributively in Two Processing Instances

Figure 2 shows an example of learning Bayesian networks using two computing nodes. Both of the processing instances score the same state space. Nonetheless, the nodes save only those states which are scored by themselves (shown as grey points). Figure 3 shows the states and the identifiers of ε_i and $\varepsilon_{p(i)}$ in $task^2.L$ of processing instances 1 and 2. The first node saves states ε_1, ε_2, ε_6 and ε_8. The second node saves states ε_3, ε_4, ε_5 and ε_7. The edges of the searching tree are represented as links to parent states indicated by identifiers. Suppose that the fourth searching instance receives the tuple $<K_3, \varepsilon_3, \varepsilon_1>$ from the second scoring instance after calculating the score of ε_3. In addition, we assume that t_3^1 is the first tuple which is searched by the fourth searching instance. Hence, the value of identifier I_3 is "4-1" because the id of the searching instance which receives t_3^1 is "4", and $task^1.counter$ is "1".

I_i	ε_i	$I_{p(i)}$
I_1	ε_1	*null*
I_2	ε_2	I_1
I_6	ε_6	I_3
I_8	ε_8	I_3

(a)

I_i	ε_i	$I_{p(i)}$
I_3	ε_3	I_1
I_4	ε_4	I_2
I_5	ε_5	I_2
I_7	ε_7	I_3

(b)

Figure 3. State List $Task^2.L$ of Processing Instance 1 (a) and Processing Instance 2 (b) in Figure 2

Because a state space is a directed graph which may contain loops, it is possible to reach a state which is already searched and scored. Storing the searching tree in a distributive manner enables each processing instance to obtain access to the information of only a part of the state space. In other words, a computing node cannot determine whether a new Bayesian network is already explored by traversing the whole searching tree.

To solve this problem, the DBNL algorithm utilises the hash code K_i of a tuple t_i^2 to assign searching tasks. The hash code is generated by using the adjacent matrix representing the structure of a DAG or CPDAG. The hash function is called an initial task and a searching task when tuple t_i^1 is constructed. Two states may not be equal if they have the same hash code because the hash function does not consider the information of the scores and the parent states of these two states. If two states contain a Bayesian network with the same structure, namely the same adjacent matrices, they must have the same hash code. One of the straightforward approaches to assigning scoring tasks in a distributive manner is to averagely partition new generated Bayesian networks to processing instances according to the number of instances in the cluster. Consider a cluster with x computing nodes. After the searching task generates a set of allowable states, by wrapping the hash code with the simple mod function, state ε_i is sent to the $((K_i \bmod x)+1)_{th}$ processing instance. This ensures that a state will be sent to the same scoring node again if it has already been previously scored. Hence, the scoring instance can check through $task^2.L$ to determine whether it has already been scored. In other words, if the hash code of a state does not exist in the local state list of a particular scoring instance which receives this tuple, the state cannot exist in any other state lists of other scoring instances.

However, the capacity of computing nodes varies in practice. By strictly dividing average searching and scoring tasks, low-performance computing nodes may lead to bottlenecks in the cluster. The DBNL algorithm utilises the grouping mechanics of cloud-computing platforms, such as the shuffle and sort functions in Hadoop MapReduce and the fields grouping mechanics in Storm. Instead of using the simple mod function, the DBNL algorithm uses K_i directly as the key of a tuple in the grouping stage. For the same purpose of mod function, tuples with the same keys are transferred to the same executors, even if they are created by a different executor at different times. Moreover, tuples are sent to a scoring instance once the instance calculation is complete. The faster a computing node performs a scoring task, the more tuples it calculates. The loads of nodes are balanced by utilising the grouping mechanics of cloud-computing platforms. This assignment strategy is used on both searching tasks and scoring tasks.

Note that the identifier I_i of a tuple is different from that of the hash code K_i of a state. In the DBNL algorithm, we say two states are equals, and are denoted by $\varepsilon_i = \varepsilon_j$, if they have the same Bayesian network models, even if they may have different score values or parent's state. The hash codes K_i and K_j have the same values for two equal states ε_i and ε_j. Furthermore, two tuples, t_i^1 and t_j^1, which contain the same elements, namely $K_i = K_j$, $\varepsilon_i = \varepsilon_j$ and $\varepsilon_{p(i)} = \varepsilon_{p(j)}$, still have different identifiers because the Bayesian networks in them are searched separately.

Algorithm 3. Scoring task

var $task^2.L$: a list storing scored states
procedure SCORE($t_{n(i,j)}^2 = <K_{n(i,j)}, \varepsilon_{n(i,j)}, \varepsilon_i>$)
  if $K_{n(i,j)} = null$ **then**
    terminate current scoring task
  else
    if $\forall s'$ in $L : s' \neq \varepsilon_{n(i,j)}$ **then**
      add the state $\varepsilon_{n(i,j)}$ to $task^2.L$
      $\varepsilon_{n(i,j)}.score := Score(\varepsilon_{n(i,j)})$
      send tuple $t_{n(i,j)}^1 = < K_{n(i,j)}, \varepsilon_{n(i,j)}, \varepsilon_i >$ to searching task
    end if
  end if
end procedure

By resolving the three problems explained above, the scoring task (shown in Algorithm 3) distributively calculates the scores of Bayesian structures, either DAGs or CPDAGs.

The DOP of the scoring task could be set to one or more accordingly to share the computation.

The scoring task terminates after receiving a tuple $t_{n(i,j)}^2$ with three *null* elements from the output task. For the tuples sent from searching tasks, the scoring task first examines whether $\varepsilon_{n(i,j)}$ in $t_{n(i,j)}^2$ exists in $task^2.L$. The scoring task adds $\varepsilon_{n(i,j)}$, which has not been scored to the $task^2.L$. After assigning $\varepsilon_{n(i,j)}.score$ a score according to $Score(\varepsilon_i, \varepsilon_{p(i)})$ function, scoring task sends tuple $t_{n(i,j)}^1$ with the same value of $K_{n(i,j)}$ and ε_i in $t_{n(i,j)}^2$ to a searching task to generate identifier $I_{n(i,j)}$ and child states of $\varepsilon_{n(i,j)}$.

2.5. Output Task

Algorithm 4. Output task

```
cons I_t: a constant for identifying the goal state
var task^3.ε_t : the state used for determining termination
procedure OUTPUT(t_i^3 = <I_i, ε_i>)
    if I_i = I_t then
        task^3.ε_t := ε_i
    else
        if Sufficiency(ε_i, task^3.ε_t)
            broadcast t_t^2 = <null, null, null> to all scoring tasks
            return the state containing trained Bayesian network ε_T
        end if
    end if
end procedure
```

There is one instance involving execution of the output task (as shown in Algorithm 4) in a cluster. The output task saves a constant variable I_t, which has the same value of the I_t in the initial task, in order to identify the tuple containing termination state ε_t. ε_t is saved to $task^3.\varepsilon_t$. An identifier that is not equal to I_t indicates that the tuple t_i^3 is scored and sent from the searching task. The output task uses the function $Sufficiency(\varepsilon_i, task^3.\varepsilon_t)$ to determine whether the ε_i meets the convergence criterion. The *Sufficiency* function differs according to the adapted Bayesian learning algorithms, which will be explained in more detail in Section 3. When the Bayesian network structure contained in ε_i meets the criterion, the output task broadcasts a tuple consisting of three *null* elements to every scoring task in order to terminate them. Finally, the output task returns ε_i as the trained Bayesian network ε_t.

3. Adapted Variants

The DBNL algorithm can be adapted to the distributed versions of structural learning algorithms for either DAG-based Bayesian networks or Markov equivalence classes. There are four adaptable functions and variables which are not specified in the description of the DBNL algorithm in Section 2. They are *Transition*, *Operator*, *Sufficiency* and E_0. We refer to a set of the four functions and variables as a variant configuration. By specifying the variant configuration, as presented in Table 2, the primary algorithm proposed in Section 2 is modified based on the principles of hill-climbing, K2, TPDA and the GES algorithm.

Table 2 Four Adapted Versions of DBNL Algorithm based on Hill-Climbing, K2, TPDA and GES Algorithm

Variant algorithm	Variant configuration			
	E_0	Γ for *Transition*(ε_i, $\gamma_{i,j}$)	*Operator*	*Sufficiency*
Hill-climbing DBNL	$\{\varepsilon_0\}$	$\{\gamma_{i,j}^{+D}, \gamma_{i,j}^{-D}, \gamma_{i,j}^{\neg D}\}$	Sorted by score	Local maximum

K2 DBNL	$\{\varepsilon_1, \ldots, \varepsilon_M\}$	$\{\gamma_{i,n}{}^{+D}\}$	Sorted by K2 score	Local maximum
TPDA DBNL	$\{\varepsilon_1, \ldots, \varepsilon_{C(M,2)}\}$	$\{\gamma_{i,j}{}^{+D}\}$, $\{\gamma_{i,j}{}^{+D}\}$, $\{\gamma_{i,j}{}^{-D}\}$	Sorted by MI	EdgeNeeded_H, EdgeNeeded
GES DBNL	$\{\varepsilon_0\}$	$\{\gamma_{i,j}{}^{+D}\}$, $\{\gamma_{i,j}{}^{-D}\}$	Unsorted	Local maximum

Similar to the centralised hill-climbing greedy search algorithm, the hill-climbing-based DBNL algorithm starts from an empty DAG. In other words, E_0 equals to $\{\varepsilon_0\}$, which contains only a state whose DAG has no edges. The *Transition*(ε_i, $\gamma_{i,j}$) function in the hill-climbing-based DBNL algorithm returns ε_j by performing operator $\gamma_{i,j}{}^{+D}$, $\gamma_{i,j}{}^{-D}$ or $\gamma_{i,j}{}^{\neg D}$ to ε_j, namely by adding, deleting or reversing the directed edges of DAG in ε_j. The *Operator* function returns a sequence of operators $\gamma_{i,j}$ in γ_{DAG} ($1 \leq m,n \leq M$) which are sorted by score in descending order. Note that the *Operator* function is called by searching for a task in several computing nodes. The strategy of giving the sequence of the operators determines the rate of convergence. In practice, we use a sorted list, rather than a ring buffer, to implement the *Disruptor Queue* of Apache Storm. The *Sufficiency* function gives a true value when the DAG score received by the output task is less than the previous one.

To adapt the DBNL algorithm based on the K2 algorithm, the notation of ε in Table 1 is modified to having one more element ρ, which represents the preceding node to be operated on.

That is, $\varepsilon_i^{\rho} = <G_i, I_{p(i)}, s_i, \rho>$. The K2-based DBNL algorithm has M initial states corresponding to M nodes in a Bayesian network which all contain an empty DAG. The only difference between the M initial states is the value of ρ, which equals i in ε_i^{ρ}. During the first execution of the searching task, edges are added to the DAGs in M initial states. In other words, the search space is partitioned into M parts so that their searching and scoring tasks can be executed simultaneously. As noted in Section 2.4, the centralised K2 algorithm keeps adding parent nodes to a particular node until the local maximum is reached. Accordingly, the set of allowable operators γ for *Transition*(ε_j, $\gamma_{i,j}$) contains only $\gamma_{i,n}{}^{+D}$, where n represents the current node. Besides, *Sufficiency* uses the same criterion as the hill-climbing algorithm.

The TPDA-based DBNL algorithm has the same three phases as TPDA. In each phase, the relevant CI tests are conducted in parallel. Because the mutual information of DAGs with each possible edge is calculated in the "drafting" phase, there is a total of C_M^2 initial states corresponding to each edge. Each initial state in E_0 contains a DAG with only one edge. The operators used in the first two phases are both $\gamma_{i,j}{}^{+D}$ because they increase the mutual information score by adding new edges. On the contrary, the searching task removes edges in the "thinning" phase. The output task uses EdgeNeeded_H and EdgeNeeded functions **Error! Reference source not found.** to determine when to terminate the "thickening" and "thinning" phase, respectively.

The major difference between the GES-based DBNL algorithm with the three DAG-based variants described above is that it performs operator $\gamma_{i,j}{}^{+D}$ and $\gamma_{i,j}{}^{-D}$ on states containing CPDAGs. Moreover, the sequence of operators performed on the states is unsorted because the scores of all CPDAGs must be calculated to obtain the local maximum.

4. Empirical Experiments

We implemented the hill-climbing version of the DBNL algorithm based on our custom Apache Storm **Error! Reference source not found.Error! Reference source not found.** framework in Java. We modified the *Disruptor Queue* of the Apache Storm project to support heuristic searching based on various score-and-search algorithms.

In the Bayesian network learning scenario, the factors that may effect the performance of algorithm includes the number of data entries, the number of attributes, the arities of attributes, and the computing nodes used for execution. To comprehensively test the performance of the algorithms proposed in this paper, we conducted experiments using four real-life datasets from different areas to test the effects of the sample data size, number of computing nodes, number of attributes in the dataset and different scoring criterion on the execution time of the DBNL algorithm. The four datasets are the Asia network dataset **Error! Reference source not found.**, Fisher's Iris dataset **Error! Reference source not found.**, StumbleUpon dataset **Error! Reference source not found.** and the Data Mining and Knowledge Discovery Cup 1999 competition (KDD1999) dataset **Error! Reference source not found.**.

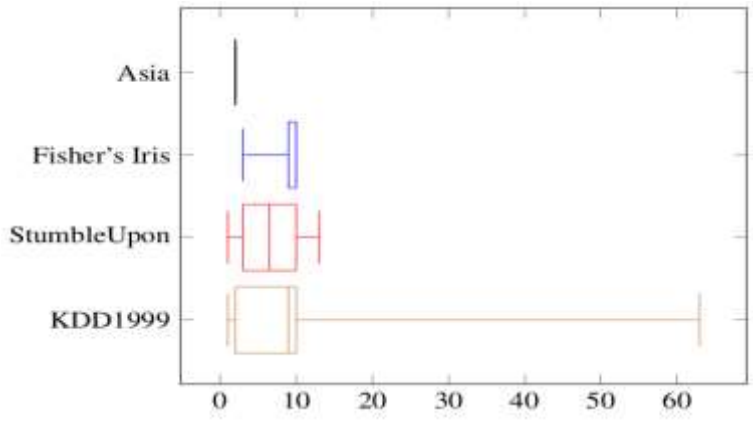

Figure 4. The Statistics of Attributes Arities for Four Datasets

The statistics of arities for four datasets are given in Figure 4. The Asia network dataset for a hypothetical medical domain is widely used in the area of Bayesian networks. It contains 100,000 entries with eight binary attributes indicating whether a patient has bronchitis, lung cancer and so on. The Fisher's Iris dataset is a dataset which contains 150 entries with attributes that record information of flower species used for discriminant analysis. The StumbleUpon dataset has 7365 entries containing information for web pages, such as its category and the average number of words in each link. The KDD1999 dataset contains more than 5 million entries with 42 attributes which describe network connections focusing on network intrusion detection. In some experiments, we normalised the size of the sample data to 100,000 by generating entries according to the probabilities of values in the original sample data. We also discretized the continuous value in the Fisher's Iris dataset, StumbleUpon dataset and the KDD1999 dataset.

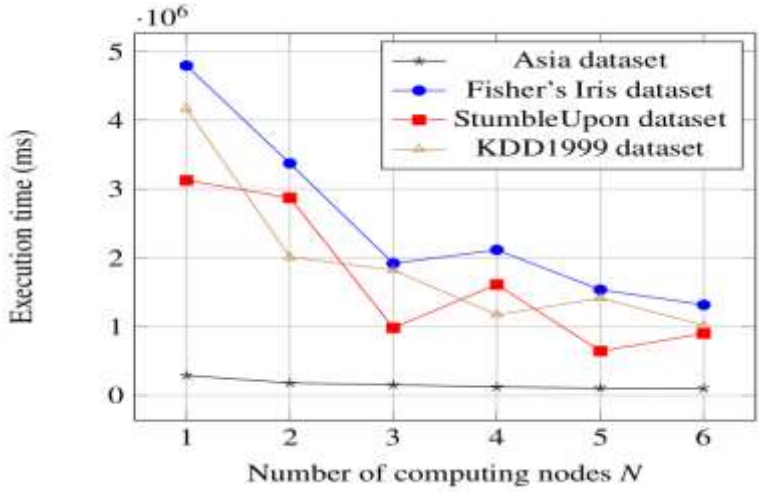

Figure 5. Total Execution Time for 1-6 Computing Nodes for Four Datasets

First, we tested the DBNL algorithm on 16 computing nodes to execute the scoring task on the four datasets. Then, we normalised the sizes of the sample data to 100,000. The MDL function is used as the scoring criterion. To compare the execution time, we set the Bayesian network learning process using the same iterations so that they have the same workloads. For each configuration in the experiment, we obtained the execution time of DBNL for 20000 iterations. The experimental results shown in Figure 5 illustrate that the execution takes less time when using more computing nodes in the cluster. The learning process on the Asia dataset takes significantly less time than the other three datasets owing to its binary attributes as noted in Figure 4. That is, the number of values that attributes can take, as well as the number of attributes, is positive correlated with the execution time of learning algorithm.

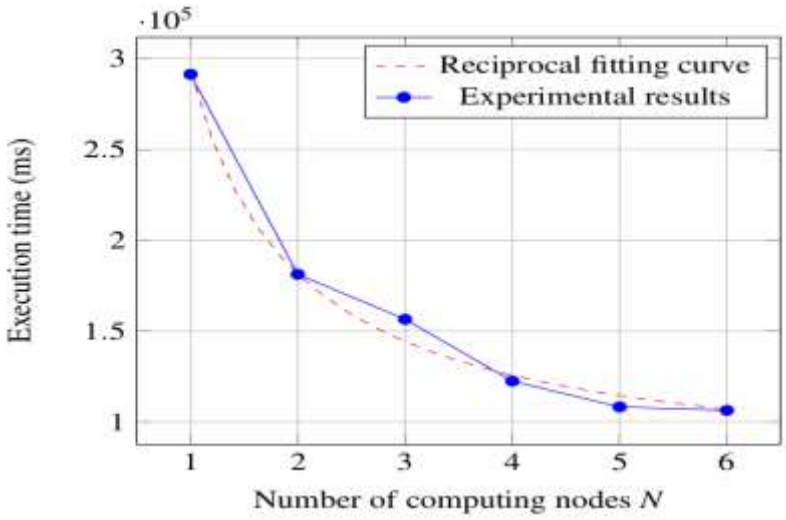

Figure 6 Reciprocal Fitting Curve for the Execution Time and Different Number of Computing Nodes for the Asia Dataset

Furthermore, Figure 5 and Figure 6 show that the execution time and number of computing nodes follow reciprocal curves. We calculated *p*, the percentage of parallel parts in DBNL and T(1), which is the sequential execution time, by using the experimental results according to the Amdahl law. As an example of the

StumbleUpon dataset, T(1) is 3125860(ms) and p is 93.28%. The speedup in this example is 14.88 because the ideal speedup $S'(N) = \lim_{N\to\infty} S(N) = 1/(1-p)$. In summary, the results indicate that the DBNL algorithm could achieve high speedup close to the linear speedup.

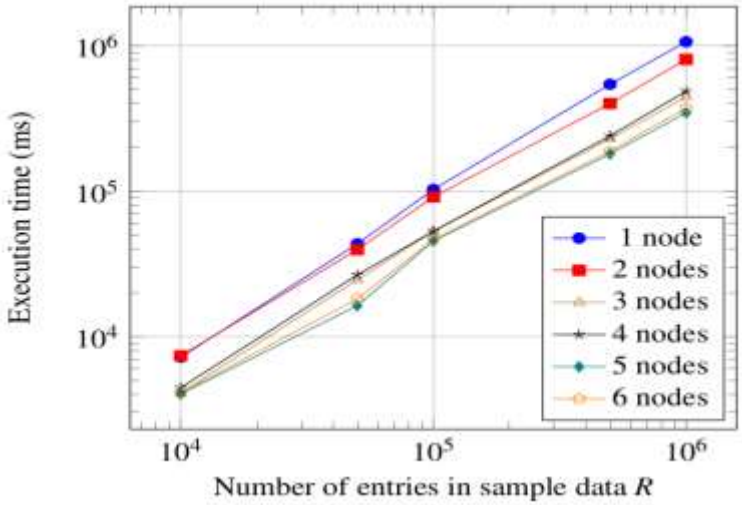

Figure 7. Total Execution Time of DBNL when Calculating *R* entries of KDD1999 Dataset

Second, we tested the proposed algorithm for a large dataset with varying sizes. Figure 7 show the execution time for each possible situation from 16 executors and 10^4 to 10^6 KDD1999 data entries. The scoring criterion is the MDL function. The execution time of the whole DBNL algorithm on the Storm cluster all show linear curves as shown in Figure 7. In other words, the execution time increases linearly with the size of the sample data. The reason we did not use the storage space of datasets to measure the time is that datasets have different numbers of attributes. As the number of attributes possessed by a dataset increases, the larger will be the storage space used by an entry, whether in memory or HDFS. By using the number of entries, we make the comparison more reasonable than using storage space. Besides, note that Figure 7 uses the logarithmic scale on both the x-axis and y-axis so that it does not change the linear relation between two variables.

Table 3. Execution Time and Number of Execution of each Tasks

p	Execution time(ms) [Number of execution]				
	Initial task	Searching task	Scoring task	Output task	Total time
1	9.00 [1]	1,052,140.65 [2]	1,060,930.00 [1000]	8.33 [3]	1,061,362.67
2	21.67 [1]	800,844.99 [3]	782,485.67 [1001]	10.33 [4]	803,892.00
3	9.00 [1]	442,258.74 [3]	441,334.44 [1002]	20.33 [4]	444,363.33
4	9.33 [1]	483,083.37 [3]	477,215.00 [1003]	8.33 [4]	486,011.67
5	9.67 [1]	345,091.79 [4]	343,943.73 [1004]	20.33 [5]	346,496.33
6	9.00 [1]	371,743.35 [4]	364,470.33 [1005]	11.67 [5]	372,064.33

Moreover, we demonstrate the execution time of each tasks and the total time in Table 3 when using 1 to 6 computing nodes. Because the four types of tasks run simultaneously, the total execution time is not the summation of execution time of all tasks. The results indicate that the searching task and scoring task occupy the most proportion of total execution time.

Table 4. Percentages of Parallel Tasks and Speedup for Different Sizes of KDD1999 Sample Data

R($\cdot 10^4$)	1	5	10	50	100
T(1) ($\cdot 10^4$)	0.78	4.67	10.86	56.03	110.31
p	55.21%	67.41%	67.45%	79.63%	79.30%
S'(N)	2.23	3.07	3.07	4.91	4.83

By calculating the proportion of the parallel parts and the sequential execution time, we obtain the speedup for different sizes of KDD1999 sample data (shown in Table 4). From the results, we determined that the execution time of sequential tasks is longer for learning Bayesian networks on a larger dataset. However, the percentage *p* and the speedup S'(N) are much larger with a larger dataset.

5. Conclusions

The large storage capacity afforded by cloud computing enhances the structural learning algorithms of Bayesian networks. In order to maximise parallelism, we proposed the distributed Bayesian network learning (DBNL) algorithm. Specifically, we solved the problems of saving large-sized sample data, detecting loops in search space and assigning computation tasks. We also demonstrated the flexibility of DBNL by illustrating four adapted variants based on hill-climbing, K2, TPDA and GES algorithms. Then, we performed a theoretical analysis of the speedup of DBNL in order to approximate the linear speedup. The experimental results support the analysis of the speedup, and indicate that the advantage of parallelism is more significant for large datasets.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (General Program) under Grant No. 61572253, "13th Five-Year Plan" Equipment Pre-Research Projects Fund under Grant No. 61402420101HK02001, Aviation Science Fund under No. 2016ZC52030.

References

[1] D. M. Chickering, D. Heckerman and C. Meek, "Large-sample learning of bayesian networks is np-hard", Journal of Machine Learning Research, vol. 10, no. 5, **(2004)**, pp. 1287-1330.
[2] F. Sahin and A. Devasia, "Distributed particle swarm optimization for structural Bayesian network learning", INTECH Open Access Publisher, **(2007)**.
[3] K. Yu, H. Wang and X. Wu, "A parallel algorithm for learning bayesian networks", In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, **(2007)**, pp. 1055-1063.
[4] K. X. Gou, G. X. Jun and Z. Zhao, "Learning bayesian network structure from distributed homogeneous data", Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, IEEE, vol. 3, **(2007)**, pp. 250-254.
[5] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth and T. D. Nielsen, "A parallel algorithm for bayesian network structure learning from large data sets", Knowledge-Based Systems, vol. 117, **(2017)**, pp. 46-55.
[6] L. Han and H. Y. Ong, "Parallel data intensive applications using mapreduce: a data mining case study in biomedical sciences", Cluster Computing, vol. 18, no. 1, **(2015)**, pp. 403-418.
[7] L. Li, J. Ye, F. Deng, S. Xiong and L. Zhong, "A comparison study of clustering algorithms for microblog posts", Cluster Computing, **(2016)**, pp. 1-13.
[8] K. Yue, Q. Fang, X. Wang, J. Li and W. Liu, "A parallel and incremental approach for data-intensive learning of bayesian networks", IEEE transactions on cybernetics, vol. 45, no. 12, **(2015)**, pp. 2890-2904.
[9] A. Basak, I. Brinster, X. Ma and O. J. Mengshoel, "Accelerating bayesian network parameter learning using hadoop and mapreduce", Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, ACM, **(2012)**, pp. 101-108.
[10] A. Basak, I. Brinster and O. J. Mengshoel, "Mapreduce for bayesian network parameter learning using the em algorithm", Proceedings of Big Learning: Algorithms, Systems and Tools, **(2012)**.

[11] Q. Fang, K. Yue, X. Fu, H. Wu and W. Liu, "A mapreduce-based method for learning bayesian network from massive data", Asia-Pacific Web Conference, Springer, **(2013)**, pp. 697-708.
[12] K. Yue, H. Wu, X. Fu, J. Xu, Z. Yin and W. Liu, "A data-intensive approach for discovering user similarities in social behavioral interactions based on the bayesian network", Neurocomputing, vol. 219, **(2017)**, pp. 364-375.
[13] J. Arias, J. A. Gamez and J. M. Puerta, "Learning distributed discrete bayesian network classifiers under mapreduce with apache spark", Knowledge-Based Systems, vol. 117, **(2017)**, pp. 16-26.
[14] W. Hummer, B. Satzger and S. Dustdar, "Elastic stream processing in the cloud", Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 3, no. 5, **(2013)**, pp. 333-345.
[15] D. M. Chickering, "Learning equivalence classes of bayesian-network structures", The Journal of Machine Learning Research, vol. 2, **(2002)**, pp. 445-498.
[16] D. M. Chickering, "Optimal structure identification with greedy search", The Journal of Machine Learning Research, vol. 3, **(2002)**, pp. 507-554.
[17] J. Cheng, G. Grainer, J. Kelly, D. Bell and W. Lius, "Learning bayesian networks from data: An information-theory based approach", Artificial Intelligence, vol. 137, no. 1-2, **(2002)**, pp. 43-90.
[18] "Apache storm", **(2015)**, URL http://storm.apache.org/. Accessed: 2018-03-02.
[19] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth and T. D. Nielsen, "A parallel algorithm for bayesian network structure learning from large data sets", Knowledge-Based Systems, vol. 117, **(2017)**, pp. 46-55.
[20] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems", Journal of the Royal Statistical Society, Series B (Methodological), **(1988)**, pp. 157-224.
[21] R. A. Fisher, "The use of multiple measurements in taxonomic problems", Annals of eugenics, vol. 7, no. 2, **(1936)**, pp. 179-188.
[22] "Stumbleupon evergreen classification challenge", **(2013)**, URL http://www.kaggle.com/c/stumbleupon. Accessed: 2018-03-02.
[23] "Kdd cup 1999: Computer network intrusion detection", **(2014)**, URL http://www.sigkdd.org/kdd-cup-1999-computer-network-intrusion-detection. Accessed: 2016-07-13.

Authors

Fei Ding, received his B. S. Degree in computer science and technology from Nanjing University of Aeronautics and Astronautics in 2011, and received M.S. degree in computing from University of York, UK in 2013. Currently, he is a Ph.D. candidate of the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics in China. His research interests include machine learning and distributed computing.

Yi Zhuang, was born in 1956. She graduated from the Department of Computer Science, Nanjing University of Aeronautics and Astronautics in 1981.Now she is a professor and Ph. D. supervisor of the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. Her research interests include network and distributed computing, information security and dependable computing.

