

# A Fast Calculation of Metric Scores for Learning Bayesian Network \*

Qiang Lü<sup>1</sup>, Xiaoyan Xia<sup>2</sup>

<sup>1</sup>*School of Computer Science and Technology, Soochow University, Suzhou, P.R. China*

<sup>2</sup>*Provincial Key Lab for Computer Information Processing Technology, Suzhou, P.R. China*

E-mail: {qiang,xyxia}@suda.edu.cn

Received: Sep. 16, 2007.

## Abstract

Frequent counting is a very so often required operation in machine learning algorithms. A typical machine learning task, learning structure of Bayesian network (*BN* for short) based on metric scoring, is introduced as an example task which heavily relies on frequent counting. A fast calculation method for frequent counting enhanced with two cache layers is then presented for learning *BN*. We name this speedup technique RC solution. The main contribution of RC is to eliminate comparison operations for frequent counting by introducing a multi-radix number system calculation. Both mathematical analysis and empirical comparison between RC and state-of-the-art solution are conducted. The results show that RC is dominant prior over the current state-of-the-art solution at least in solving the problem of learning *BN*. Further discussions on how to extend RC to other similar learning tasks are also developed at the end of this paper.

**Keywords:** Artificial intelligence, learning Bayesian network, frequent counting, radix-based calculation, ADtree.

---

\*This work is supported by NSFC under Grant No. 60673041

# 1 Introduction

Inspired by the approach which decomposes a data mining algorithm into six components[1], we can also consider a machine learning algorithm based on datasets at least consisting of two operational components: one is to collect statistical information from the dataset, the other is to apply learning logics to these statistics. Many of such statistic operations need to do frequent counting on a specific (training) dataset.

Frequent counting is to calculate the repetition times (or the count) of which a specific query is suited on a given dataset. Let us firstly define what a query is. Given an attribute set  $\mathbf{X} = \{x_1, \dots, x_M\}$ , with  $V^i$  as the attribute value set from which  $x_i$  can take,  $|V^i| = r_i$  is called the arity of attribute  $x_i$ . Given a dataset  $\mathbf{D} = \{d_1, \dots, d_R\}$  where  $d_i$  is a possible assignment for each  $x_i \in \mathbf{X}$ , a query is defined as a sequence of  $(x_i = v_i^j)$  pairs in which  $v_i^j \in V^i \cup \{\phi\}$  and all the  $x_i$ es forms a subset of  $\mathbf{X}$  arranged in increasing order of index. Please note that the attribute value  $\phi$  means “any value” or “don’t care”. Therefore the count of a query is the number of records in  $\mathbf{D}$  matching all the pairs in the query.

Based on frequent counting, machine learning algorithms usually construct a little bit more complicate structures for their specific learning logics. The most useful such structures are contingency table (*ct* for short) and conditional contingency table (*cct* for short). A *ct* constructs a table of all the frequent counting on some interested attributes. And a *cct* is a *ct* filtered by another query (called conditional query of this *ct*).

To make the above description clearly understood, we take an example of learning structure of discrete Bayesian network (*BN* for short). A *BN* =  $(B_s, B_p)$  consists of two components.  $B_s$  describes the probabilistic relationships between  $M$  discrete random attributes in  $\mathbf{X}$ . Such relationships can be described by a directed acyclic graph (*DAG*) with nodes for all attributes in  $\mathbf{X}$  respectively, and each beginning node of a directed arc is called the parent of the ending node. Let  $\pi_i = (x_{i_1}, \dots, x_{i_n})$  be the parent nodes of  $x_i$ ,  $B_p$  denotes  $p(x_i|\pi_i), \forall x_i \in \mathbf{X}$ . Therefore the joint probability distribution of *BN* can be described as

$$p(\mathbf{X}) = \prod_{i=1}^n p(x_i|\pi_i) \quad (1)$$

The problem of learning *BN* (structure) is to find a  $B_s$  that best matches  $\mathbf{D}$ . Of course the point here is how you define the matching degree of a  $B_s$  under  $\mathbf{D}$ . Basically there are two different approaches to measure the match degree<sup>1</sup>, one based on conditional independence tests and the other based on scoring metrics. This paper is only focus on approaches based on scoring metrics.

---

<sup>1</sup>There are also other methods which combines the two basic approaches.

One of the popular used metrics for scoring  $B_s$  is  $K2$  metric[2] (log version):

$$f_{K2}(x_i, \pi_i) = \sum_{j=1}^{q_i} \left( \log \frac{(s_i - 1)!}{(N_{ij} + s_i - 1)!} + \sum_{k=1}^{s_i} \log(N_{ijk!}) \right) \quad (2)$$

where  $s_i$  is the number of possible values of the variable  $x_i$ ,  $q_i$  is the number of possible instantiations for the variables in  $\pi_i$ ,  $N_{ijk}$  is the number of cases in  $\mathbf{D}$  in which variable  $x_i$  has its  $k$ th value and  $\pi_i$  is instantiated to its  $j$ th value, and  $N_{ij} = \sum_{k=1}^{s_i} N_{ijk}$ . Figure 1 clearly shows how  $N_{ijk}$  and  $N_{ij}$  are constructed.

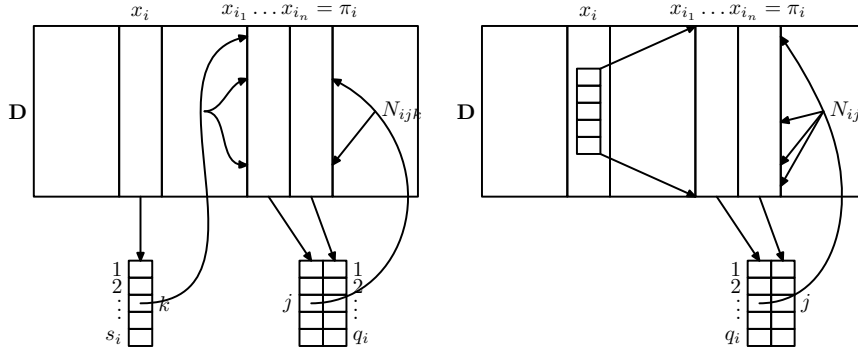


Figure 1:  $N_{ijk}$  description

This formula (2) measures how well  $x_i$  is under the parents  $\pi_i$ . Therefore the whole goodness of this  $BN$  is sum of all such scores. Learning  $BN$  by  $K2$  metric is to try every possible  $B_s$  to achieve  $B_s^* = \arg \max f_{K2}(B_s, \mathbf{D})$ .

There are also a lot of other similar metrics which measure the goodness in different views. It is clearly that the core of the computation in equation (2) is the calculation of  $N_{ijk}$  which can be mapped to each cell of a  $ct$ . For a typical learning algorithm the calculation of each evaluated metric score usually takes a huge time.

This paper presents a fast calculation solution for learning  $BN$ . We compare our solution with the state-of-the-art solution in both mathematical analysis and empirical comparison. And we achieve the dominant prior over the current state-of-the-art solution at least in solving the problem of learning  $BN$ . We also discuss how to extend our method to other similar learning tasks, although we only describe our solution in detail to learning  $BN$ .

The organization of this paper is as follows: In Section 1 we give the background of frequent counting and introduce a learning task which heavily relies on frequent counting. A brief introduction of popular solutions of frequent counting is reviewed in Section 2. For the purpose of speeding learning  $BN$ , two cache layers are designed in Section 3.1 and 3.3, and a fast calculation method of metric score is presented in Section 3.2. To evaluate our presentation, mathematical analysis and experimental analysis are presented in Section 4 and Section 5. In Section 6 we discussed some variants of RC for other learning tasks. Finally we conclude this paper in Section 7.

## 2 Related Work

Building a *ct* can be transformed to implementing a data cube problem[3]. A lattice framework was introduced to model such multidimensional analysis in their work, and they concluded that there was a trade-off for the fast generation of a new data cube: not all the cells in the computed data cube should be memorized

ADtree is a technique to re-organize the counting information of a training dataset into a tree (called ADtree) so that all possible *cts* or *ccts* can be fast extracted from the built tree[4]. ADtree was proved to be a super technique to accelerate Bayesian network learning task, rule learning task, feature selection task, and other machine learning tasks[5, 6, 7, 8, 9].

Unlike ADtree solution building a whole ADtree before a learning procedure starts, dynamic ADtree is an enhanced version of ADtree which incrementally builds a dynamic ADtree keeping pace with the client's on-line requirement[10]. In this way, dynamic ADtree improves memory performance of ADtree and couples more tightly with the client side.

The research group of ADtree also developed a new RADSEARCH method which extended ADtree to real-valued all-dimensions search[11].

KDtree, similar to ADtree, is an earlier tree structure to store counting information of the training data[12, 13]. As [4] stated, KDtree node splits on only one attribute instead of all attributes like ADtree does. This results in less memory but expensive counting.

Frequent set finder[14] was originally used with very large dataset of millions of records containing very sparse binary attributes. The founded frequent sets could be efficiently used to frequent counting[5].

In a summary, frequent counting techniques can be divided into two schemes: pre-counting and post-counting. By pre-counting, we mean that all the counting information is gathered from the training dataset before the learning algorithm is started. The features of pre-counting techniques are obvious: more memory to contain a transformed dataset, fast counting service and independent of the running task. ADtree<sup>2</sup>, KDtree and Frequent set finders are examples of pre-counting techniques.

By post-counting, we mean that only the necessary counting information required by one running of a learning algorithm is calculated. More strictly, the counting is done only upon the request from the running procedure of a learning algorithm. Of course post-counting may also need some

---

<sup>2</sup>Although ADtree will on-line build *ct* upon the frequent counting requirement, all the counting information is already embedded in the ADtree. Making *ct* is just to gather the counting information scattered in the ADtree. That's why we categorize ADtree into pre-counting techniques because the most counting operations are done before they are really needed.

pre-process to the training datasets and some special initialization. The most important feature of post-counting techniques is that the overall performance of the counting depends on a specific running. The possible factors affecting the performance for a running include the properties of the dataset, counting requirements of learning algorithm, and so on. Dynamic ADtree and the method introduced in this paper are examples of post-counting techniques.

In this paper, we propose a post-counting technique. It is a fast solution at least for learning *BN* algorithms based on metric score approach. As for this specific learning task, there are several important features which make our solution much workable:

1. Usually the maximum parents number  $n$  is restricted to like 4. This means that the size( $n+1$ ) of  $ct$  is not very big.
2. Once a  $ct$  has been built, different kinds of  $N_{ijk}$  can be derived for calculating  $f_{K2}$ , and such calculations will be required many times.
3. Special attribute value “ $\phi$ ” does not appear in the  $ct$  query.

## 3 Speeding up Techniques for Learning *BN*

### 3.1 Caching the Metric Scores

Since careful design of data structures and cautious implementation of correspond algorithms will greatly improve the performance of meta-heuristic algorithms[15, 16], we presents some speeding techniques in solving the problem of learning *BN*.<sup>3</sup>

It is obvious that calculating  $N_{ijk}$  and  $N_{ij}$  is a time-consuming job. Compared to the computing time, the space required for storing the dataset can be usually afforded easily. Please refer to Section 4 for the mathematical analysis of the space complexity for storing the dataset.

Please note that what we are proposing to cache are those computed metric scores, not individual  $N_{ijk}$  or  $N_{ij}$ . Each score is decided by a tuple  $(x_i, \pi_i)$  and of course the dataset. The current empirical studies show that the number of calculations of different tuples is around 45000 for ALARM dataset, 29000 for INSURANCE dataset and 13000 for BOBLO dataset[17, 18]. So basically for achieving avoidance of the recalculations, it is worth to maintain a hash mechanism to cache the metric scores.

We encode the hash key like this:

$$(x_i, \pi_i) = (x_i, (x_{i_1}, \dots, x_{i_n})) \mapsto (x_i, \{0, 1\}^M) \quad (3)$$

---

<sup>3</sup>Learning *BN* is NP-hard problem.

where  $M$  is the total number of attributes,  $\pi_i = (x_{i_1}, \dots, x_{i_n}), 1 \leq i_1 < i_2 < \dots < i_n \leq M$ .  $\pi_i$  is then encoded to a  $M$ -bit string  $b_1 \dots b_{i_1} \dots b_{i_n} \dots b_M$  where  $b_{i_1}, \dots, b_{i_n}$  are set to 1 and the other are set to 0. Considering that the *BN* benchmark problems have less than 64 nodes, a 64-bit integer is enough for encoding  $\pi_i$ . Note this also means that  $x_i$  will take less than 6 bits for encoding every node.

Since every node  $x_i$  is always used via the tuple  $(x_i, \pi_i)$ , the bits  $k_1 \dots k_6$  of  $x_i$  should be taken into the higher part of hash key. But because the first bit  $k_1$  of  $x_i$  does not contain much information as the other bits, we finally omit the first bit and take  $k_2 \dots k_6$  of  $x_i$  into the first part of hash key. So we take another 7 bits  $k'_1 k'_2 \dots k'_7$  from  $\pi_i$  into the lower part of hash key. We extract the bits from the  $\pi_i$  code at the position of 2, 5, 7, 11, 17, 19, 23. It is very obvious to notice that these positions are all at the prime number index order. Now we design a 12-bit hash key like this:  $k_2 \dots k_6 k'_1 k'_2 \dots k'_7$ .

### 3.2 Fast Calculation of $ct$

It is obvious that all the counting information for calculating  $f_{K2}(x_i, \pi_i)$  can be derived from  $ct(x_{i_0}, x_{i_1}, \dots, x_{i_n})$ . For the convenience, we denote  $x_i$  by  $x_{i_0}$ , and  $ct(x_i, \pi_i) = ct(x_{i_0}, x_{i_1}, \dots, x_{i_n})$  by  $ct(n+1)$ . Of course it is obvious that  $ct(n)$  denotes a  $ct$  with  $n$  attributes.

In fact a  $ct(n+1)$  can be considered as a hyper data cube

$$ct[0..r_{i_0} - 1][0..r_{i_1} - 1] \dots [0..r_{i_n} - 1]$$

Please recall that  $r_i$  is the arity of  $x_i$ . Therefore the formula (2) can be rewritten in terms of  $ct$  cells:

$$f_{K2}(x_i, \pi_i) = \sum_{i_1=0}^{r_{i_1}-1} \dots \sum_{i_n=0}^{r_{i_n}-1} \left( \log \frac{(r_{i_0} + 1)!}{(N_{ij} + r_{i_0} + 1)!} + \sum_{k=0}^{r_{i_0}-1} \log (ct[k][i_1] \dots [i_n])! \right) \quad (4)$$

where  $N_{ij} = \sum_{k=0}^{r_{i_0}-1} ct[k][i_1] \dots [i_n]$ .

Now we will show that we can calculate  $ct(n+1)$  data cube without any re-scanning or comparison operations. This speeding technique requires a preprocessing to the dataset which can be easily satisfied. Suppose that all the domains  $V^i$  of each random variable  $x_i$  can be expressed by a series of ordered sequential numbers. That is we can represent the actual value  $v_1^i, v_2^i, \dots, v_{r_i}^i$  with their corresponding index number  $0, 1, \dots, r_i - 1$ . This can be easily achieved by introducing a preprocessing to the dataset without losing any generalities of solving machine learning tasks.

The arity  $r_i$  of each  $x_i$  can be easily calculated when reading the dataset. Just remember the biggest value of each incoming  $x_i$ . Once the dataset is read, increase that biggest number by one to get  $r_i$ .

Now let's see how  $ct(n + 1)$  can be calculated in exactly  $R$  arithmetical addition operations. And during one pass of scanning of the dataset, for every row containing a configuration  $(v_{n'_0}^{i_0}, v_{n'_1}^{i_1}, \dots, v_{n'_n}^{i_n})$  of  $(x_{i_0}, x_{i_1}, \dots, x_{i_n})$ , just increase the counter of

$$ct[n'_{i_0}][n'_{i_1}] \dots [n'_{i_n}]$$

by one. Then after one pass of the scanning, a complete  $ct$  has been constructed. In fact,  $ct$  is a standard data hypercube whose grid contains the count information we are expecting. In this way we makes the counter calculation fast enough by eliminating the matching operation.

It is obvious that once  $ct(x_i, \pi_i)$  has been constructed, all  $ct(x_{i'}, \pi_{i'})$ s, where  $x_{i'} \in \{x_i\} \cup \pi_i$  and  $\pi_{i'} \subset \{x_i\} \cup \pi_i \setminus \{x_{i'}\}$ , can be gradually derived from  $ct(x_i, \pi_i)$  without the necessary to scan any passes of the dataset. We call such  $ct(x_{i'}, \pi_{i'})$  sub- $ct$  of  $ct(x_i, \pi_i)$ . For example,  $ct(n - 1) = ct(i_1, i_2, \dots, i_n \setminus i_k)$ , where  $1 \leq k \leq n$ , can be calculated from  $ct(n)$  like this:

$$ct(n - 1) = \underbrace{ct[i_1][i_2] \dots [i_n]}_{n - 1 \text{ items}} = \sum_{j=0}^{r_{i_k} - 1} \underbrace{ct[i_1][i_2] \dots [j] \dots [i_n]}_{n \text{ items}} \quad (5)$$

And again,  $ct(n - 2)$  can be calculated from  $ct(n - 1)$ , and so on. Therefore it is necessary to cache  $ct(n)$  for its possible future requirements from all other  $2^n - 1$  sub- $cts$ . Please notice that  $ct(0)$  is not necessary for our learning task.

### 3.3 Outlines of Fast Calculating Metric Scores

Similar to the process of cache design of metric score layer, we can also design a cache layer for storing computed  $cts$  and “building-ahead”  $cts$ .

Combined by two cache layers: metric score and  $ct$ , we name our calculation solution for learning  $BN$  radix-calculation (RC for short) . The general procedure of RC solution for calculating  $f(x_i, \pi_i)$  is summarized as follows:

1. Search  $f(x_i, \pi_i)$  in the score cache layer. If hit, return the cached score.
2. Create a  $f(x_i, \pi_i)$  node, and insert it into the score cache layer.
3. Search  $ct(x_i, \pi_i)$  in the  $ct$  cache layer. If hit, fetch it and calculate the above new  $f(x_i, \pi_i)$ . Return the fresh  $f(x_i, \pi_i)$  score.
4. Create a new  $ct$  node, compute and insert it into the  $ct$  cache layer. Calculate the above new  $f(x_i, \pi_i)$ . Return the fresh  $f(x_i, \pi_i)$  score.

Please notice that both the  $ct(n)$  and  $ct(n + 1)$  can be calculated in exactly  $R$  time cost. This implies that we can take such a “building-ahead” technique: when  $ct(n)$  is not cached, we build

$ct(n+1)$  in  $R$  time and spend another  $r^{n+1}$  time for calculating  $ct(n)$ . That is we cache two  $cts$  in not very much time, and all other  $2^{n+1}$  sub- $cts$  of  $ct(n+1)$  will not require scanning the dataset.

In this paper we just adopt a very simple strategy to decide the ahead attribute  $x_k$  of  $ct(\dots, x_i)$ :  $k$  is the next round closest to  $i$ .

## 4 Complexity Analysis and Comparison

The most common technique of speeding such calculations was ADtree solution proposed by [4]. Since the cache idea is the same for ADtree and RC, we just analyze the part of  $ct$  calculation.

Following [4]'s notation,  $R$  is the row number of the dataset,  $n$  is the dimension of  $ct$ ,<sup>4</sup>  $r$  is the arity of each attribute  $x_i$ , and  $M$  is the attribute number in  $\mathbf{X}$ .

### 4.1 Time Complexity of RC

Let  $T(n)$  be the time cost for RC to calculate  $ct(n)$ . It is clear that  $T(n) = R$ . And if  $ct(n)$  is cached, then from equation (5),

$$\begin{aligned} T(n-1) &= r^n \\ T(n-2) &= T(n-1) + r^{n-1} = r^n + r^{n-1} \\ &\dots \\ T(1) &= \sum_{j=2}^n r^j \end{aligned}$$

That means that the cost of  $T(n)$  will be shared by all its sub- $cts$ ' calculation. So the average computing time of all such  $cts$  is:

$$\frac{R + \binom{n}{n-1}T(n-1) + \binom{n}{n-2}T(n-2) + \dots + \binom{n}{1}T(1)}{2^n - 1} = \frac{R + \sum_{k=1}^{n-1} \left( \binom{n}{k} \sum_{j=k+1}^n r^j \right)}{2^n - 1} \quad (6)$$

We want to emphasize that the time metric of formula (6) is totally different from the regular sense of time cost in literature. In general, the time cost of generating such data cube was [4, 11]:

$$\mathcal{O}(nR + r^n) \quad (7)$$

In formula (6), the quantity is exactly for the arithmetical operation: addition. While in formula (7), the quantity is referring to comparison operations between attribute values. In another abstract word, such quantity is referring to searching or matching operations. We shall simply keep this in mind: the time cost of each operation in formula (6) is less than that in formula (7).

---

<sup>4</sup>In the context of learning  $BN$ , this means we want to calculate  $f_{K2}(x_i, \pi_i)$  where  $|\pi_i| = n - 1$ .

## 4.2 Space Complexity of RC

The first part of space cost of RC is  $R \prod_{i=1}^M \log_2 r_i \approx R(\log_2 r)^M$  for storing the dataset, where  $M$  is the number of variables, that is  $|\mathbf{X}|$ , and  $r_i$  is the arity of  $x_i$ . This cost will also be shared by a complete running. And it is easily afforded by a modern computer. For example, usually  $r_i < 256$ , we can take a byte enough for storing the value of each variable. Then the dataset occupies  $RM$  bytes. For a 20K-row and 40-variable dataset, it is 800KB.

The second part of space cost of RC is for caching  $ct(n)$  of  $r^n$  cells. And the really space cost will depend on the specific running behavior of the learning algorithm. But one thing is for sure that learning  $BN$  task will not require too much  $cts$  which the computer can not bear. Please check Section 5.1 for the evidence. Theoretically once all sub- $ct(n-1)$ s have been calculated/cached from  $ct(n)$ , and all  $n$  metric scores  $f(ct(n))$  have been calculated/cached, the space occupied by  $ct(n)$  should be released.

## 4.3 Comparison Between RC and ADtree

The following results about ADtree analysis are directly from [4]. If we take ADtree solution, a  $ct$  can be calculated in

$$(1 + n(r - 1))r^{n-1} \quad (8)$$

plus the one-off cost of building ADtree at the initial phase, which is bounded above by

$$\sum_{k=0}^{\lfloor \log_2 R \rfloor} \frac{R}{2^k} \binom{M}{k} \quad (9)$$

[4] gave a complicate space cost analysis of building ADtree under different situations, basically it is (considering the dataset of binary variables)

$$\sum_{k=0}^{\lfloor (\log_2 R)/(-\log_2 q) \rfloor} \binom{M}{k}$$

where  $q$  is related to the distribution of the values for variables. This space complexity is bounded above by

$$\mathcal{O} \left( \frac{M^{\lfloor (\log_2 R)/(-\log_2 q) \rfloor}}{(\lfloor (\log_2 R)/(-\log_2 q) \rfloor - 1)!} \right) \quad (10)$$

In fact, such ADtree is some kind of a re-organized dataset. The space cost will be reduced only when the original dataset has suited some compressing criteria.

Similarly the cost space of building a  $ct$  with  $n$  variables is bounded above by

$$\mathcal{O} \left( \frac{n^{\lfloor (\log_2 R)/(-\log_2 q) \rfloor}}{(\lfloor (\log_2 R)/(-\log_2 q) \rfloor - 1)!} \right) \quad (11)$$

It is obvious that the above two RC and ADtree methods beat the regular method which does not take cache mechanism. So we do not list the analysis results of the regular method. We summarize the above analysis results in Table 1 ( $m$  is the maximum number of different  $cts$  for a complete running):

		RC	ADtree
time	for init	$\approx 0$	formula (9)
complexity	for running	formula (6)	formula (8) + $\frac{1}{m}$ · formula (9)
space	for init	$RM$	formula (10)
complexity	for running	$\sum r^n$	formula (11)

We just skip the detail mathematical comparison of the complexity of the two methods in verbose symbolic formulas, because there does not seem to have a closed form result for such mathematical comparison. The more important reason why we do not compare the two RC and ADtree columns in Table 1 in mathematical way is that the results on ADtree are bounds which are on the worst situations. So we just list the computation results of two typical cases: for a3k case, let  $R_1 = 3000$  and  $m_1 = 50000$ ; for a20k case, let  $R_2 = 20000$  and  $m_2 = 60000$ . Other parameters are same:  $n = 5$ ,  $q = 0.5$  (although this is not favor for ADtree, for an unknown dataset, can we have other fair settings?),  $M = 37$ , and the average arity  $r = 2.8$ . Please note that these settings are all favor for ADtree method except for the noted  $q$  (because we have no choice.). Please check the example of learning ALARM network in Section 5.1 for the reasons why we have such settings. Now let's calculate Table 1 and present the results in Table 2:

		RC(a3k/a20k)	ADtree(a3k/a20k)
time	for init	0	3.86e+9/5.09e+10
complexity	for running	324.78/873.17	640+77200/1274+848333
space	for init	1.1e+5/7.4e+5	4.9e+10/1.4e+12
complexity	for running	$\sum r^n$	1.3/0.9

The  $\sum r^n$  part in Table 2 can not be calculated accurately because it depends on the specific running procedure. Please check Table 4 for the empirical values.

We find from Table 2 that RC method is overall prior to ADtree in the performance of both time and space.

## 5 Experimental Results

We evaluate RC solution in two case studies: the first is to compare RC with direct calculation solution, the second is to compare RC with ADtree solution. In both cases we target the learning task on learning the best structure of  $BN$  with the highest  $K2$  metric score.

### 5.1 Comparison between ACOB+RC and ACOB+Direct Calculation

Three applications have been evaluated in this section. ACOB is a heuristic approach to learning  $BN$ [17, 18]. Regarding the calculation of metric scores, it used cache technique to eliminate the recalculation of the metric scores. We list the results of this application in  $ACOB_c$  line of Table 3.

We presented a parallel implementation of ACOB[19]. Since our focus is on how the parallel behavior helps improving the diversification of learning process, we adopt nothing speeding techniques on calculating metric scores. That means once the request of scoring  $(x_i, \pi_i)$  is made, we scan the dataset and calculate it from scratch. We list the results of the sequential version of this application in  $ACOB_d$  line of Table 3.

In this paper, we reimplemented ACOB with RC calculation. We list our computing results in  $ACOB_{RC}$  line of Table 3.

Our running platform is IBM p550 4×Power5 1.5Ghz with 8MB memory and operating system IBM AIX5.3. The compiler is gcc 3.3.3. Since the three implementations are strictly following the same learning algorithm, we set the same parameter values for the comparison.

The training dataset is standard ALARM dataset[20] with the first 3000 rows. We call this training dataset a3k. It is easy to understand that by saying a20k we mean ALARM dataset with 20000 rows.

We give the average  $\mu$ , standard deviation  $\sigma$  and best value in brackets over 10 executions in Table 3, where **Diff score** column lists the number of different  $(x_i, \pi_i)$  scores to be calculated during one execution, and **All scores** column lists the number of all  $(x_i, \pi_i)$  scores to be calculated during one execution.

First of all, it is clear that  $ACOB_{RC}$  is much much faster than  $ACOB_d$ . The running time of  $ACOB_{RC}$  is several hundred seconds and that of  $ACOB_c$  is more than one hour. Although the running time is not comparable due to the different running platforms, we believe that the difference of the running time on order of magnitude has made sense. As for potential reference, one running of the deterministic greedy heuristic of K2SN[21] takes 9s on our platform involving 4060 calculations of different tuples and 22201 calculations of all tuples.

It is interesting to note that the **Diff scores** number of  $ACOB_{RC}$  is much bigger than that of

Table 3: Learning  $BN$  from a3k by ACOB-K2 with RC and without RC

Application	$K2$ score $\dagger$	Diff scores	All scores	T(h:mm)	Source
ACOB $_c$	-14401.83 $\pm$ 0.72 (-14401.29)	44693.20 $\pm$ 1208.56	75.48e05 $\pm$ 10.78e4	1:05 $\ddagger$	[17]
ACOB $_d$	(-14409.01)	not provide	1387503	4:36	[19]
ACOB $_{RC}$	-14401.78 $\pm$ 0.33 (-14401.29)	71854.2 $\pm$ 2360.88	75.27e05 $\pm$ 4.50e4	369.15s $\pm$ 11.45	This paper

$\dagger \mu \pm \sigma(\text{best})$

$\ddagger$ This computing time is not comparable with the other two because the source did not describe the running platform.

ACOB $_c$ , while the **All scores** number of ACOB $_{RC}$  is slightly smaller than that of ACOB $_c$ . This explains why the final result  $K2$  score of ACOB $_{RC}$  is a little bit better than that of ACOB $_c$  because ACOB $_{RC}$  explores more different pair scores. Considering all the parameter settings are exactly same, especially the iteration number is set to 100, it seems there are no reasons why ACOB $_{RC}$  explores more different pair scores even if it can get the pair score faster. The probable explanation would be the different running time of the local optimizer HCST[22]. Source [17] did not give the iteration number of HCST, while ACOB $_{RC}$  sets the iteration number of HCST to 20000. Again we believe that ACOB $_{RC}$  does not exceed ACOB $_c$  in execution time.

The last comment about Table 3 is that the ratio of **All scores** over **Diff scores** indicates the efficiency of score cache layer. It measures the contribution of score cache layer to score calculation. In Table 3 this ratio is about 104.76. This implies that the efficiency of score cache is very high.

How about the efficiency of  $ct$  cache layer? Table 4 lists the typical results of  $ct$  generating about running ACOB $_{RC}$  on a3k and a20k.

Table 4: Results of  $ct$  cache layer about running ACOB $_{RC}$ 

	Diff $ct$	Scan	ByS	FreeS	E	n=1,2,3,4,5	Memory
a3k	54202	33402	78096	11197	1.82	37,666,4852,17378,31262	6.56e+6
a20k	59159	36561	89194	11790	1.88	37,666,5083,18789,34577	7.22e+6

The explanation of Table 4 columns is as the following:

- **Diff  $ct$** : the number of different  $cts$  has been calculated. This indicates the scale which  $ct$  cache layer should manage. It will usually be much less than the **Diff scores** column in Table 3.

- **Scan**: the number of scanning dataset for generating all *cts*. Less is better. This item greatly determines the performance of RC.
- **ByS**: the number of *cts* used by score calculation.
- **FreeS**: the number of free *cts* which ONLY have been used as “building-ahead” *cts*. These *cts* have not been used by directly metric score calculation. Of course they are calculated by scanning the dataset. They are considered as of another cache for *ct* producing.
- $E(\text{fficiency}) = \frac{\text{ByS}}{\text{Diff}_{ct} - \text{FreeS}}$ , measures the contribution of *ct* cache layer to score calculation. Comparing to the efficiency of score caching, the efficiency of *ct* caching is much lower. But due to the huge time cost of building *ct*, the saved time of *ct* caching is still considerable.
- $n=1,2,3,4,5$ : the number of different  $ct(n)$ s with respect to  $n$ . Of course the sum should be equal to **Diff** *ct* value.
- **Memory**: considering  $ct(n)$  will take  $r^n$  space for storing, the whole memory taken by *ct* cache layer can be calculated. This column gives examples of  $\sum r^n$  part in Table 2.

Table 4 also explains why we set  $m_1 = 50000$  and  $m_2 = 60000$  for example evaluation in Section 4.

## 5.2 Comparison between GHC+RC and GHC+ADtree

An optimized implementation of Hill-Climbing (HC for short) for learning *BN* based on ADtree calculation method, enhanced with metric score caching, was presented in [23]. A similar earlier version of HC for learning *BN* based on ADtree was also used in [4] to demonstrate that ADtree could accelerate the task of learning *BN*.

The HC-ADtree implementation (appfbrun\_linux\_api package) on i386 platform can be downloaded from [24] upon request. In fact this package implements a new heuristic learning algorithm Optimal Reinsertion. HC is just a local search part of Optimal Reinsertion[23]. Since we are interested on the metric score calculation, we dis-enable the Optimal Reinsertion feature but let the HC part work alone. Actually HC-ADtree has two phases: the first greedy HC (we name it GHC-ADtree) phase and the second multi-restart HCST (10000 iteration per restart) phase. The first phase has to be finished even if the user does not give enough time, and the second phase will take over the remaining running time. For an empty *BN* graph, GHC will exactly require  $M$  scores like  $f_{K2}(x_i, \emptyset)$  and  $M(M-1)$  scores like  $f_{K2}(x_i, x_j)$  where  $i \neq j$ . This will result in all together  $M^2$  fresh scores but  $M + \frac{M(M-1)}{2}$  new *cts*.

Please note that the second phase of HC-ADtree will behave a slight differently due to the random number generator. Because we want to keep the exactly same learning logic when doing comparison so that we can compare the calculation efficiency, we also implement the same GHC but with our RC calculation underlined. So that we can compare the performance of RC and ADtree on calculation efficiency when learning *BN*.

The testing platform is Pentium D-core 3.0Ghz with 2GB memory and Redflag Linux 5 <sup>5</sup> The compiler is gcc 3.4.3. We tested GHC with RC and GHC with ADtree on all the published datasets[23]. Table 5 lists the comparative results,

Table 5: Compare results between GHC+RC and GHC+ADtree

Dataset	<i>R</i>	<i>M</i>	AA	RC	ADtree	Dataset source
adult	49K	15	7.7	1.64075	2+	UCI Rep.(R. Kohavi)
alarm	20K	37	2.8	4.31135	5+	[20]
covtype	150K	39	2.8	36.12651	39+	UCI Rep.(J. Blackard)
connect4	67K	43	3.0	20.17593	22+	UCI Rep.(J. Tromp)
edsgc	300K	24	2.0	failed	30+	[25]
synth2	25K	36	2.0	5.10623	6+	[23]
synth3	25K	36	2.0	5.08823	6+	[23]
synth4	25K	36	2.0	5.09523	6+	[23]
nursery	13K	9	3.6	0.15	<1	UCI Rep.(M. Bohanec et al.)
letters	20K	17	3.4	0.86387	1+	UCI Rep.(D. Slate)

where *R* is the record number of the dataset, *M* is the attribute number of the dataset, and AA is the average arity of the dataset.

In Table 5, **RC** column is the running seconds of GHC-RC, and **ADtree** column is that of GHC-ADtree. Since GHC-ADtree did not give the accurate time measure, we use “+” to indicate more than something over the integer seconds. The first comment is that we did not list the initial time for building ADtree. The second comment is that for a large dataset contained 300K rows, GHC-RC failed. For all the other datasets, GHC-RC is better than GHC-ADtree even if without considering the initial time, which is a huge mount of time for building ADtree.

<sup>5</sup>Since we did not get the source code of HC-ADtree, we have to test it with the provided execution code.

## 6 Further Discussion of RC

We shall mention that the current implementation of RC is for calculating metric score of learning *BN*. As for other general machine learning tasks, RC can be easily modified to different variants. For example, RC can be used to fold a complete *ct* to a two-dimensional array to implement a  $cct(\eta|\pi)$ , where  $\eta = (x_{i_1}, \dots, x_{i_n})$  and  $\pi = (x_{j_1}, \dots, x_{j_p})$ . In RC solution, we can fold a complete  $ct(x_{i_1}, \dots, x_{i_n}, x_{j_1}, \dots, x_{j_p})$  to a two dimensional array: one dimension index is numbered with the mixed-radix  $(r_{i_1}, \dots, r_{i_n})$  number system and the other dimension index with the mixed-radix  $(r_{j_1}, \dots, r_{j_n})$  number system. Figure 2 shows how this mapping works.

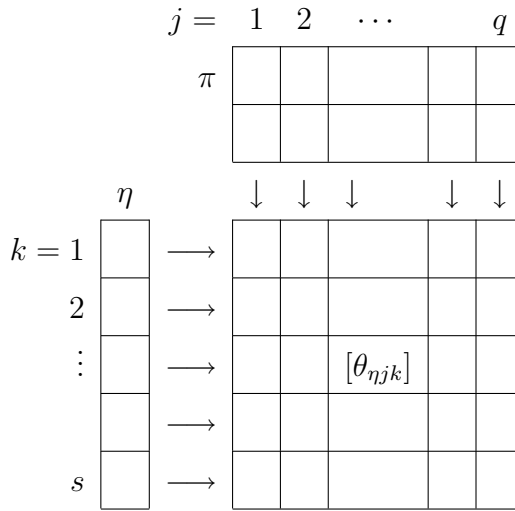


Figure 2: Convert a *ct* to a two dimensional *cct*

It is obvious that  $q = \prod_{j \in \pi} r_j$  and  $s = \prod_{j \in \eta} r_j$ . In fact Figure 1 is a special case of Figure 2 where  $\eta = x_i$ . We just rotate  $q_i$  part of Figure 1 to index the column of  $\theta_{\eta j k}$  block, and multiply  $s$  part, which indexes the row of  $\theta_{\eta j k}$  block, of Figure 1.

How to locate  $j$  and  $k$  of  $\theta_{\eta j k}$  block for each coming row of dataset? The key trick is to use a mixed-radix number system. Suppose a configuration of  $\pi = \{x_{j_1}, x_{j_2}, \dots, x_{j_n}\}$  is  $(n'_{j_1}, n'_{j_2}, \dots, n'_{j_n})$ . So  $j$  can be calculated like this:  $j = \sum_{p=j_1}^{j_n} n'_p d_p$ , where  $d_{j_n} = 1, d_{j_{n-1}} = r_{j_n}, d_{j_{n-2}} = r_{j_n} r_{j_{n-1}}, \dots, d_{j_1} = \prod_{p=2}^n r_{j_p}$ .

It is similar to locate  $k$  for a given  $\eta$ . Now for every row of dataset, locate  $k$  and  $j$  and therefore the  $\theta_{\eta j k}$ , simply increase the corresponding  $\theta_{\eta j k}$  by one. Once the dataset has been scanned one pass, a complete *cct* has been constructed.

In another common learning task: association rule mining[14], we need to estimate value like this:

$$p(x_{i_1} = n_{i_1}, \dots, x_{i_n} = n_{i_n} | x_{j_1} = n'_{j_1}, \dots, x_{j_p} = n'_{j_p}) \\ = \frac{ct(x_{i_1} = n_{i_1}, \dots, x_{i_n} = n_{i_n}, x_{j_1} = n'_{j_1}, \dots, x_{j_p} = n'_{j_p})}{ct(x_{j_1} = n'_{j_1}, \dots, x_{j_p} = n'_{j_p})} = \frac{ct[k][j]}{\sum_k ct[k][j]}$$

It is very clear that the above counting information could be directly read from Figure 2.

In fact another learning *BN* approach, conditional independence tests, is heavily relied on mutual information

$$I_{P_{B_s}}(x_i, x_j) = P_{B_s}(x_i, x_j) \log \frac{P_{B_s}(x_i, x_j)}{P_{B_s}(x_i)P_{B_s}(x_j)}$$

where  $P_{B_s}(\cdot)$  is the estimation distribution of  $\mathbf{X}$  under the *BN* structure  $B_s$ . And conditional mutual information  $I_{P_{B_s}}(\eta|\pi)$  is also needed in such approach. It is obvious that these calculation can also be easily calculated from *ct* and *cct*.

In the extreme case, if memory is enough, the biggest *ct*( $M$ ) can be calculated in  $R$  time, and any other sub-*ct* can be calculated in  $\mathcal{O}(r^M)$ . So the best average time of building any *ct* is

$$\frac{R + \sum_{k=1}^{M-1} \left( \binom{M}{k} \sum_{j=k+1}^M r^j \right)}{2^M - 1} \quad (12)$$

It is a little bit complicated to give the simple result comparison between (12) and  $\mathcal{O}(r^M)$ . Of course any *ct* can be calculated in  $R$  time cost. So the general time cost of building a *ct*, if we take no consideration of space requirement, is  $\mathcal{O}(\min(R, r^n))$  where  $n$  is the attribute number of the *ct*. This result is consistent with the one mentioned in [4].

But we leave a huge (and probably very sparse) data cube *ct*( $M$ ) there. In fact ADtree is the way to re-organize this biggest data cube to a tree-like structure. ADtree focuses on the organization of the grid for constructing data cube by squeezing the zeroes and MCV (most common value) but adding descriptions of the relationships. ADtree and its subsequent research, e.g. dynamic ADtree introduced in [10], did take the traditional calculation for each grid, while RC focuses on the fast calculation of each grid. Therefore ADtree and RC can be combined consistently: ADtree is focusing on the organization of computed *ct*, and RC is focusing on the calculation of *ct*.

Of course there is no “silver bullet” for universal learning cases. But in general, more general suitable for learning tasks, more waste for a specific learning task. Vice versa, more efficient for a specific learning task, more lose of the generality.

In summary, we conclude that RC has also the same general coverage of helping machine learning algorithms as the ADtree approach, but with less complexity of time and space, and with more robustness and stability. Especially for learning *BN* structure with the limitation of parents number, RC solution is the best solution for calculating metric score by now.

## 7 Conclusion and Future Work

Firstly we summarize two requirements for which RC method can be applied: the dataset as a whole can be loaded into memory; and each attribute domain  $V^i$  is enumerable. As we see in the

description of Section 4, these two requirements are easily satisfied by introducing a preprocessing without losing any generalities of RC method applied to other machine learning tasks.

The key contribution of RC method is to eliminate comparison operations of frequent counting with number-system calculations. Mathematical analysis and empirical studies show that RC method is prior to ADtree at least in solving the problem of learning  $BN$ . RC method is robust because it has less parameters than ADtree solution. RC is suitable for small and medium scale datasets with larger arities, while ADtree for larger datasets with smaller arities. As for what kind of scale can be judged as small, medium or large scale, it is hard to give a clear line to distinguish. It is actually depending on at least two factors: one is dynamic behaviors when running on specific dataset via specific learning logics, the other is the ratio of cost of arithmetical addition over that of attribute value matching operation.

The performance of RC will also depend on the “building-ahead” strategy which will greatly save time for deriving sub- $cts$  from the mother  $ct$ . This is the very important feature we will study in the future agenda.

The performance analysis of the cache layer is another independent topic which has not been covered in this paper although we have also collected the data during our testing. Without tightly coupled with the learning task a general discussion of cache performance does not make any sense. It is obvious that cache performance is related to the procedure of requesting calculation of a tuple sequence  $(x_i, \pi_i)$ . So we need more tests on different datasets to evaluate the cache performance. This is also covered in our future research agenda.

## Acknowledgments

The authors greatly thank Jisi Pan for his parallel implementation of ACOB, Xiaojuan Wu for her mathematical validation in Section 4 and preprocessing of the datasets, Xuejun Liao for his computing experimental comparison between RC and HC-ADtree.

## References

- [1] David J. Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*, chapter 5 A Systematic Overview of Data Mining Algorithms. The MIT Press, 2001.
- [2] G.F. Cooper and E. Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–348, 1992.

- [3] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD '96*, pages 205–216, Montreal, June 1996.
- [4] A. Moore and M. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.
- [5] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In E. Simoudis, J. Han, and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996.
- [6] Y. Tsin, Y. Liu, and V. Ramesh. Texture replacement in real images. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR '01)*. IEEE Computer Society Press, Los Alamitos, CA, 2001.
- [7] Qin Ding, Qiang Ding, and William Perrizo. Association rule mining on remotely sensed images using p-trees. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 66–79, 2002.
- [8] A. Dobra, A. Karr, and A. Sanil. Preserving confidentiality of high-dimensional tabular data: Statistical and computational issues. *Statist. and Computing*, 13(4):363–370, 2003.
- [9] S. Sanghai, P. Domingos, and D. Weld. Dynamic probabilistic relational models. In *IJCAI-03*, 2003.
- [10] Paul Komarek and Andrew Moore. A dynamic adaptation of ad-trees for efficient machine learning on large data sets. In *Proceedings of the 17th International Conference on Machine Learning*, 2000.
- [11] Andrew Moore and Jeff Schneider. Real-valued all-dimensions search: Low-overhead rapid searching over subsets of attributes. In Adnan Darwiche & Nir Friedman, editor, *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, pages 360–369, 340 Pine Street, 6th Fl., San Francisco, CA 94104, July 2002. Morgan Kaufmann Publishers, SF CA.
- [12] S. M. Omohundro. Efficient algorithms with neural network behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987.
- [13] A. W. Moore, J. Schneider, and K. Deng. Efficient locally weighted polynomial regression predictions. In D. Fisher, editor, *Proceedings of the 1997 International Machine Learning Conference*, 1997.
- [14] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. *Advances in Knowledge discovery and Data Mining*, chapter Fast discovery of association rules. AAAI Press, 1996.

- [15] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann Publishers, 2004.
- [16] Roberto Battiti and Marco Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, 2001.
- [17] Luis M. de Campos, Juan M. Fernandez-Luna, Jose A. Gámez, and J. Miguel Puerta. Ant colony optimization for learning bayesian networks. *International Journal of Approximate Reasoning*, 31(3):291–311, 2002.
- [18] Luis M. de Campos, Jose A. Gámez, and J. Miguel Puerta. Learning bayesian networks by ant colony optimization: Searching in the space of orderings. *Mathware and Soft Computing*, 9(2-3):251–268, 2002.
- [19] Jisi Pan, Qiang Lv, and Hongling Wang. A parallel ant colonies approach to learning bayesian network. *Journal of Chinese Computer system*, 28(4):652–655, 2007. (in Chinese).
- [20] I. A. Beinlich, H.J. Suermondt, R. M. Chavez, and G. F. Cooper. The alarm monitoring system: A case study with two probabilistic inference techniques for belief networks. In *Proceedings of Second European Conference on AI and Medicine*, pages 247–256, Berlin, 1989. Springer-Verlag.
- [21] Luis M. de Campos and Jose Miguel Puerta. Stochastic local algorithms for learning belief networks: Searching in the space of the orderings. In *ECSQARU*, pages 228–239, 2001.
- [22] David Heckerman, Dan Geiger, and David Maxwell Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.
- [23] Andrew Moore and Weng-Keen Wong. Optimal reinsertion: A new search operator for accelerated and more accurate bayesian network structure learning. In T. Fawcett and N. Mishra, editors, *Proceedings of the 20th International Conference on Machine Learning (ICML '03)*, pages 552–559, Menlo Park, California, August 2003. AAAI Press.
- [24] Auton lab: <http://www.autonlab.com/autonweb/10530.html?branch=1&language=2g>. URL, 2007.
- [25] R. C. Nichol, C. A. Collins, and S. L. Lumsden. The edinburgh/durham southern galaxy catalogue - ix. the galaxy catalogue. <http://xxx.lanl.gov/abs/astroph/0008184>, 2000.