A Condition-Enumeration Tree Method for Mining Biclusters from DNA Microarray Data Sets

Jiun-Rung Chen, Ye-In Chang [*]

Dept. of Computer Science and Engineering, National Sun Yat-Sen University

No. 70, Lienhai Rd., Kaohsiung 80424, Taiwan, R.O.C.

Abstract

Biclustering, which performs simulataneous clustering of rows (*e.g.*, genes) and columns (*e.g.*, conditions), has proved of great value for finding interesting patterns from microarray data. To find biclusters, a model called *pCluster* was proposed. A pCluster consists of a set of genes and a set of conditions, where the expression levels of these genes have a similar variation under these conditions. Based on this model, most of the previous methods need to compute *MDSs* (Maximum Dimension Sets) for every two genes in the microarray data. Since the number of genes is far larger than the number of conditions, this step is inefficient. Another method called MicroCluster was proposed. This method does not compute MDSs for every two genes, and transforms the problem into a graph problem. However, it needs to solve the *Maximal Clique* problem, which is NP-Complete. To avoid the above disadvantages, in this paper, we propose a new method, *CE-Tree* (Condition-Enumeration Tree), for finding pClusters. Instead of generating MDSs for every two genes, we generate only MDSs for every two conditions. Then, based only on these MDSs, we expand the CE-Tree in a special *local breadth-first within global depth-first* manner to efficiently find all pClusters. We also utilize the idea of the traditional hash join approach to efficiently support the CE-Tree. From the simulation results, we show that the CE-Tree method could find pClusters more efficiently than those previous methods.

Key words: Bicluster, bioinformatics, expression level, microarray, pCluster

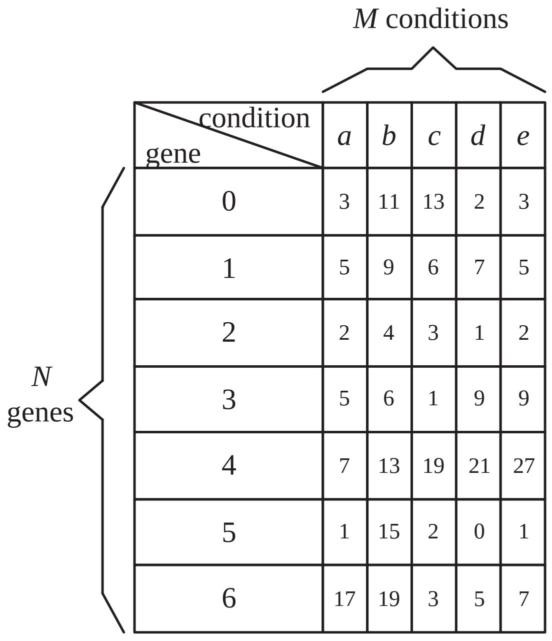
1 Introduction

Microarrays are one of the breakthroughs in experimental molecular biology, providing a powerful tool by which the expression patterns of thousands of genes can be monitored simultaneously and are already producing huge amount of valuable data (Yang et al., 2005). The gene expression data generated from microarrays are organized as matrices, where rows represent genes, columns represent various samples such as tissues or experimental conditions (one microarray per column), and values in each cell characterize the expression level of the particular gene in the particular sample (Yang et al., 2005). Figure 1 shows an example of a gene expression matrix, where the number of genes, N, is usually from 10^3 to 10^4, and the number of conditions, M, is usually less than 10^2. Due to the large number of genes and the complexity of the underlying biological networks, extracting information from microarray data is a formidable task (Tan et al., 2008). Analysis of such data is becoming one of the major bottlenecks in the utilization of the microarray technology (Yang et al., 2005).

* Corresponding author. Tel.: +886-7-5252000 (ext. 4334); Fax: +886-7-5254301.
Email addresses: `jiunrung@gmail.com` (Jiun-Rung Chen),
`changyi@cse.nsysu.edu.tw` (Ye-In Chang).

Since the microarray data which come from biological experiments are usually large, the techniques of data mining could be applied to analyze the data efficiently (Merz, 2003). As a data mining task, *clustering* groups objects into classes of similar objects according to some distance measurement, in a large, multidimensional data set (Chen et al., 1996). Clustering of microarray data can lead to molecular classification of disease states, identification of co-fluctuation of functionally related genes, functional groupings of genes and logical descriptions of gene regulation, among others (Aguilar-Ruiz, 2005). Discovery of such clusters is essential in revealing the significant connections in gene regulatory networks (Yang et al., 2005).

Traditional clustering methods work in the full dimensional space, which consider the value of each object in all the dimensions and try to group the similar objects together (Zhao and Zaki, 2005). Moreover, they can only be applied to either the rows or the columns of the data matrix, separately (Aguilar-Ruiz, 2005). Biclustering (Cheng and Church, 2000), however, does not have such a strict requirement. It performs simultaneous clustering of rows and columns. If some objects are similar in several dimensions (a subspace), they will be clustered together in that subspace. This is very useful, especially for clustering in a high dimensional space where often only some dimensions are meaningful for some subsets of objects. Moreover, investigations show that more often than not, several genes contribute to the same pathway, which motivates researchers to identify a subset of genes whose expression levels rise and fall coherently under a subset of conditions; that is, they exhibit the fluctuation of a similar shape when conditions change (Yang et al., 2005). Therefore, biclustering has proved of great value for finding the interesting patterns in microarray data (Zhao and Zaki, 2005).


M conditions
N genes

gene \ condition	*a*	*b*	*c*	*d*	*e*
0	3	11	13	2	3
1	5	9	6	7	5
2	2	4	3	1	2
3	5	6	1	9	9
4	7	13	19	21	27
5	1	15	2	0	1
6	17	19	3	5	7

Fig. 1. A gene expression matrix

There have been several types of biclusters proposed (Madeira and Oliveira, 2004). Among these types, biclusters with *coherent values* define a bicluster as a subset of genes and a subset of conditions of the microarray data which have coherent values of expression levels on both rows and columns (Madeira and Oliveira, 2004). Figure 2 shows an example of biclusters with coherent values in the microarray data shown in Figure 1. Gene set $\{1, 2, 5\}$ under condition set $\{a, c, e\}$ forms a bicluster, as shown in Figure 2-(a), while gene set $\{0, 2, 5\}$ under condition set $\{a, d, e\}$ forms another bicluster, as shown in Figure 2-(b). Both of these two biclusters consist only a subset of all genes and a subset of all conditions. Moreover, their gene sets have an overlap, so do their condition sets. Furthermore, in Figure 2-(a), gene 1 is not close to other genes, if measured by distance functions such as Euclidean, Manhattan, or Cosine often used in traditional clustering methods. However, we could see that in Figure 2-(a), the expression levels of these three genes exhibit the fluctuation of a similar shape, which shows an interesting cluster (*i.e.*,

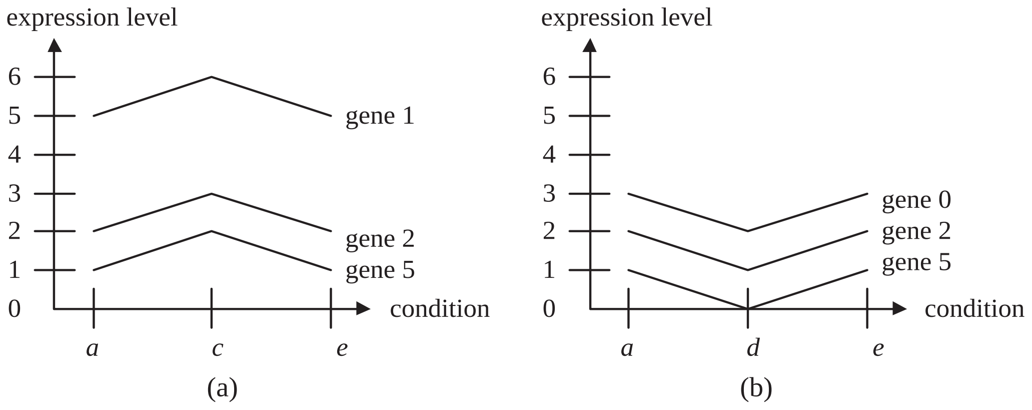


Fig. 2. An example of biclusters: (a) a bicluster consisting of gene set $\{1, 2, 5\}$ and condition set $\{a, c, e\}$; (b) another bicluster consisting of gene set $\{0, 2, 5\}$ and condition set $\{a, d, e\}$.

a *correlation cluster*). Due to the above properties, it is difficult to apply traditional clustering methods to the problem of biclustering.

Therefore, to mine biclusters with coherent values, there have been many methods proposed. Most of these methods are based on one of the following two models: *δ-bicluster* and *pCluster*. Figure 3 shows the methods based on these two models. The δ-bicluster model was proposed by Cheng and Church (Cheng and Church, 2000). They proposed a term, the *mean squared residue*, to define a δ-bicluster. The mean squared residue is evaluated as a score for a submatrix of the microarray data. If the score of one submatrix is below threshold δ, this submatrix forms a δ-bicluster. The FLOC method (Yang et al., 2002, 2005) also follows this model. However, this model has several limitations (Wang et al., 2002). First, a submatrix of one δ-bicluster is not necessarily a δ-bicluster. This creates difficulties in designing efficient methods. Moreover, many found δ-biclusters may contain some outliers. Furthermore, these methods apply a greedy and random manner to find δ-biclusters, which may miss some biclusters. Therefore, another model, *pCluster*, was proposed

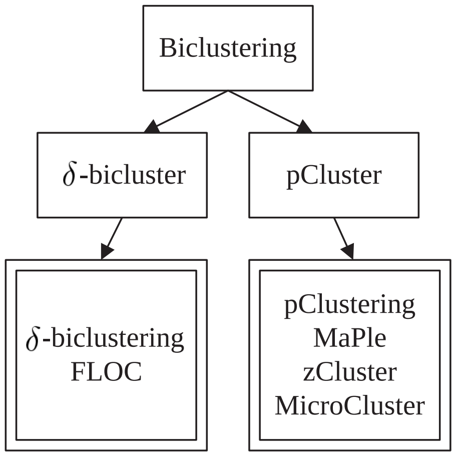


Fig. 3. The biclustering methods

(Wang et al., 2002). The pCluster model determines whether a submatrix of the microarray data could be a pCluster according to the *pScore* of this submatrix. As compared to the δ-bicluster model, the pCluster model has two advantages. First, a submatrix of one pCluster must also be a pCluster. Second, the pClusters found will not contain outliers. Therefore, in this paper, we will focus on the design of a new method based on the pCluster model.

Many methods based on the pCluster model have been proposed. Most of the methods, *e.g.*, pClustering (Wang et al., 2002), MaPle (Pei et al., 2003), and zCluster (Yoon et al., 2005), need to compute *MDSs* (Maximum Dimension Sets) for both of every gene-pair and every condition-pair in the gene expression matrix. An MDS is formally defined as follows (Wang et al., 2002): assuming $c = (O, T)$ is a pCluster, column set T (or row set O) is an MDS of c if there does not exist $T' \supset T$ (or $O' \supset O$) such that (O, T') (or (O', T)) is also a pCluster. Note that a gene-pair MDS (consisting of a pair of genes and T) indicates under which conditions two genes have similar behavior, and a condition-pair MDS (consisting of a pair of conditions and O) indicates which genes have similar behavior under two conditions. Since the number of genes (usually from 10^3 to 10^4) is far larger than the number of conditions (usually less than 10^2) in the microarray data, computing the MDSs for every two genes should be time-consuming. Moreover, to find pClusters, these methods utilize the gene-pair MDSs to construct a prefix tree, and perform a complex duplicating process at each node of the prefix tree. The time complexity of this duplicating process seems to be exponential to the number of conditions, which reduces the efficiency of these methods. Although the MicroCluster method (Zhao and Zaki, 2005) computes only condition-pair MDSs and utilizes a multigraph to find pClusters, it needs to solve the Maximal Clique problem, which is a well-known NP-Complete problem and can not be solved in polynomial time.

Therefore, in this paper, we propose a new method, *CE-Tree* (Condition-Enumeration Tree), to find pClusters from DNA microarray data sets. In the proposed method, first, instead of generating the gene-pair MDSs, we generate only the condition-pair MDSs. Then, we utilize the Condition-Enumeration Tree (CE-Tree) to find the maximal pClusters. The CE-Tree will enumerate the possible combinations of conditions, and derive the corresponding set of genes at each node, even without the help of those gene-pair MDSs. Moreover, at each node of the CE-Tree, unlike the prefix tree used in those previous proposed methods, we do not need to perform the complex duplicating process. Furthermore, we expand the CE-Tree in a special manner, *the local breadth-first within the global depth-first*. We make use of many properties of this expanding manner to develop three bounding techniques, which could help us avoid exhaustively expanding the CE-Tree. We also utilize the idea of the traditional hash join approach to propose a new data structure, *signature tables*, which could significantly improve the efficiency of the joining process performed at each node of the CE-Tree when deriving the corresponding set of genes. From the simulation and experiment results, we show that the proposed CE-Tree is more efficient than those previous proposed methods (Wang et al., 2002; Yoon et al., 2005; Zhao and Zaki, 2005).

The rest of this paper is organized as follows. In Section 2, we give a survey of two biclustering methods. Section 3 presents the proposed CE-Tree method. In Section 4, we study the performance of the CE-Tree method and make a comparison with those previous proposed methods. Finally, we make a conclusion in Section 5.

2 Related Work

In this section, we describe two biclustering methods, including pClustering (Wang et al., 2002) and zCluster (Yoon et al., 2005).

2.1 The pClustering Method

In (Wang et al., 2002), Wang *et al.* noticed several limitations of the δ-bicluster model for biclustering, and proposed a new model, *pCluster*. Let D be a set of objects (genes) in the microarray data, and A be a set of attributes (conditions) of objects in D. Let O be a subset of D ($O \subseteq D$) and T be a subset of A ($T \subseteq A$). Pair (O, T) specifies a submatrix of the microarray data. Given

$x, y \in O$, and $a, b \in T$, we define the *pScore* of the 2×2 matrix as:

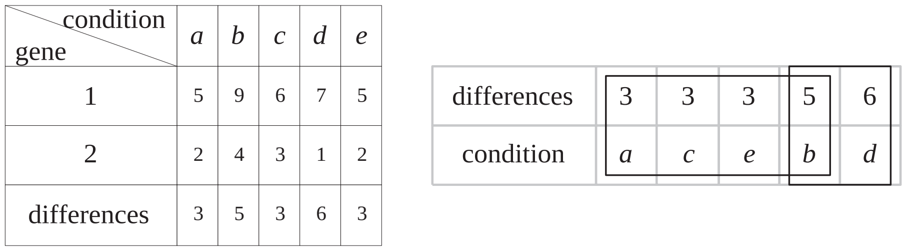
$$pScore(\begin{bmatrix} d_{xa} & d_{xb} \\ d_{ya} & d_{yb} \end{bmatrix}) = |(d_{xa} - d_{xb}) - (d_{ya} - d_{yb})|,$$

where d_{uv} is the value (the expression level) of object u on attribute v. Pair (O, T) forms a pCluster if for any 2×2 submatrix X in (O, T), we have $pScore(X) \leq \delta$ for some $\delta \geq 0$. Intuitively, $pScore(X) \leq \delta$ means that the change of values on the two attributes between the two objects in X is confined by δ, a user-specified threshold. If such confines apply to every pair of objects in O and every pair of attributes in T, we have found a pCluster. Formally, the generation of a pCluster needs to consider two more thresholds, NR and NC, where NR and NC defines the minimal numbers of objects and attributes, respectively. That is, for pCluster (O, T), $|O| \geq NR$ and $|T| \geq NC$.

The pClustering method contains 3 main steps for generating pClusters as follows.

Step 1: Pairwise Clustering

Since the *pScore* of any 2×2 submatrix of one pCluster must be less than threshold δ, the pClustering method generates MDSs for any two genes (gene-pair) and any two conditions (condition-pair). These MDSs will be combined into a pCluster consisting more than two genes and more than two conditions later. Figure 4 shows an example of the generation of gene-pair MDSs of the pClustering method, where the raw data are from the microarray data shown in Figure 1. To generate gene-pair MDSs, first, the pClustering method computes the differences between two genes under all conditions, as shown in Figure 4-(a). Next, these differences will be sorted in ascending order, as shown



condition / gene	a	b	c	d	e
1	5	9	6	7	5
2	2	4	3	1	2
differences	3	5	3	6	3

(a)

differences	3	3	3	5	6
condition	a	c	e	b	d

(b)

Fig. 4. An example of generating gene-pair MDSs: (a) the differences between gene 1 and gene 2; (b) the sorted differences.

in Figure 4-(b). Then, this method groups those close differences together. For each group, the difference between the largest one and the least one must be less than or equal to the threshold, δ. Therefore, if δ is 2, in the example shown in Figure 4-(b), there exist two groups, $\{a, c, e, b\}$ and $\{b, d\}$. That is, for genes 1 and 2, the corresponding MDS is $\{\{a, c, e, b\}, \{b, d\}\}$. This means that genes 1 and 2 have similar behavior under not only conditions $\{a, c, e, b\}$ but also conditions $\{b, d\}$. If one MDS contains less than NC conditions, it is pruned. A similar process is used to generate the condition-pair MDSs.

Step 2: MDS Pruning

The number of pairwise MDSs depends on the clustering threshold δ and the user-specified minimum numbers of genes and conditions, *i.e.*, NR and NC. However, only some of these pairwise MDSs are valid. That is, there exist only some of these MDSs actually occurring in pClusters with size larger than $NR \times NC$. In Step 2, the pClustering method tries to prune these MDSs. It counts the number of times for each condition of a gene-pair MDS occurring in condition-pair MDSs, and the number of times for each gene of a condition-pair MDS occurring in gene-pair MDSs. If the number of occurrence is not large enough, that condition/gene is removed from that MDS.

Step 3: Tree Constructing and Traversing

To generate pClusters, the pClustering method utilizes gene-pair MDSs to construct a prefix tree. Each edge of the prefix tree corresponds to one condition of a gene-pair MDS. At the left node along one path, it records the two genes of one gene-pair MDS. Figure 5-(a) shows an example of the prefix tree for two gene-pair MDSs $(\{1, 2\}, \{a, b, c, d\})$ and $(\{1, 3\}, \{b, e\})$.

Then, the pClustering method performs a post-order traversal of the prefix tree. For each traversing node, it first detects the pClusters contained within. Then, it duplicates the gene pairs at the current node to those nodes which represent subsets with size $(k - 1)$ of the k conditions of the current node. Figure 5-(b) shows an example of duplicating the gene-pairs of the leaf node under path $\{a, b, c, d\}$. Gene pair $\{1, 2\}$ will be duplicated to those nodes under pathes $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, d\}$, and $\{b, c, d\}$. When traversing to the next node, *i.e.* the node under path $\{a, b, c\}$ in this example, this duplicating process will be performed again, until the depth of the traversing node equals to NC. This process, in fact, duplicates the gene information at one node to all nodes which represent the power set of conditions of this node. This seems to be inefficient and time-consuming.

2.2 The zCluster Method

Yoon *et al.* (Yoon et al., 2005) proposed a method to improve the performance of the pClustering method by exploiting *Zero-suppressed Binary Decision Diagrams* (ZBDDs). We call this method *zCluster* in this paper. ZBDDs are a compact data structure which provides an efficient representation for ma-

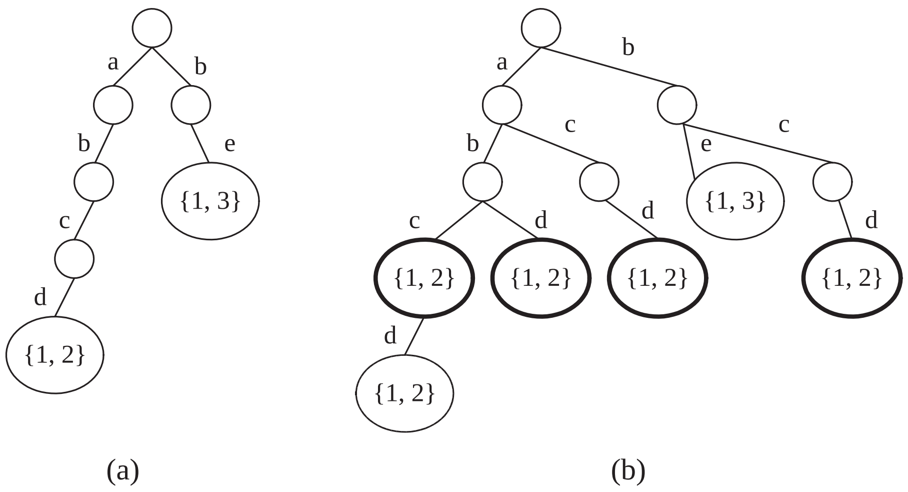


Fig. 5. An example of the prefix tree used in the pClustering method: (a) the prefix tree; (b) duplicating the gene information.

nipulating large-scale sets of combinations. The zCluster method utilizes the ZBDDs to store the set of conditions in each gene-pair MDS. This ZBDD-based representation is crucial to keeping the entire method computationally manageable.

Most steps of the zCluster method are similar to the pClustering method. These two methods differ in the last step, which is to generate pClusters by traversing a prefix tree. The zCluster method also constructs a prefix tree by using gene-pair MDSs. However, after the prefix tree is constructed, the zCluster method traverses this tree in a depth-first manner. At each node of the prefix tree, the zCluster method derives the set of related genes corresponding to the conditions along the path from the root node to the current traversing node. Figure 6 shows an example of the prefix tree used in the zCluster method. Assume that GS_{xy} is the set of related genes for conditions x and y. When traversing to the node under path $\{a, b, c\}$, GS_{abc} is derived by $GS_{ab} \otimes GS_{ac} \otimes GS_{bc}$, where $A \otimes B = \{I | I = a \cap b, \forall a \in A \text{ and } \forall b \in B\}$. Similarly, GS_{abcd} is derived by $GS_{abc} \otimes GS_{ad} \otimes GS_{bd} \otimes GS_{cd}$. These intersection operations are

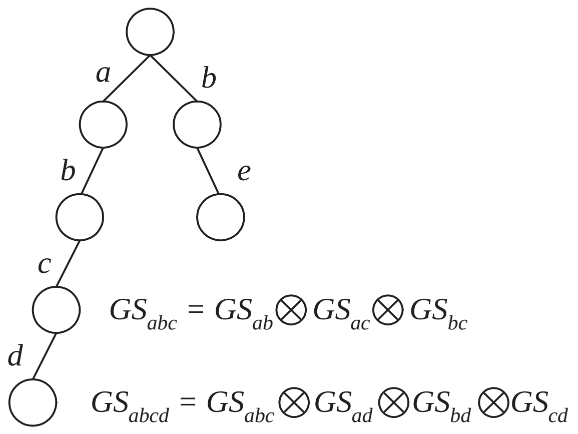

a
b
b
e
c
d
$GS_{abc} = GS_{ab} \otimes GS_{ac} \otimes GS_{bc}$
$GS_{abcd} = GS_{abc} \otimes GS_{ad} \otimes GS_{bd} \otimes GS_{cd}$

Fig. 6. An example of the prefix tree used in the zCluster method

implicitly performed on ZBDDs, thus resulting in high efficiency.

3 The Proposed Method

In this section, we present our method, *CE-Tree* (Condition-Enumeration Tree), for finding pClusters from DNA microarray data. Table 1 shows the variables used in the proposed method, and Figure 7 shows the proposed method. The proposed method has three main steps: (1) generating condition-pair MDSs, (2) the pruning step, and (3) the joining step. We will describe these three steps in the following three subsections, respectively.

3.1 Step 1: Generating Condition-Pair MDSs

In the CE-Tree method, first, we generate condition-pair MDSs from the given microarray data. The purpose of this step is to find which genes have similar behavior under the same two of these conditions. Basically, the process of this step is similar to that used in the pClustering method (Wang et al., 2002). That is, for every two conditions of the microarray data, we evaluate

Table 1

Variables used in the proposed method

Variable	Description
MA	A two-dimensional gene expression matrix
$GenS$	The set of genes of the matrix
$CondS$	The set of conditions of the matrix
δ	The tolerable deviation of a pCluster
NR	The minimal number of genes (rows) of a pCluster
NC	The minimal number of conditions (columns) of a pCluster
PC	The set of pClusters in the matrix
$CountT$	A table consists of $\|CondS\|$ tuples, where each tuple contains two fields: (1) *Count* which counts the number of occurrence of each condition, and (2) *CandList* which records a candidate list of conditions

```
Function CE-Tree($MA, \delta, NR, NC$): a set of pClusters;
begin
   // Step 1: generating condition-pair MDSs
   foreach $cond1, cond2 \in CondS, cond1 < cond2$ do
      $GenCondMDS(cond1, cond2)$;
   // Step 2: the pruning step
   foreach tuple $t \in CountT$ do
      if $(t.Count < (NC - 1))$ then
         remove $t.Cond$ from $CountT$ wherever it occurs;
   // Step 3: the joining step
   foreach tuple $t \in CountT$ do
      ExpandCE-Tree($t.Cond, t.CandList$);
   return $PC$;
end;
```

Fig. 7. The CE-Tree method

the difference between these two conditions for each gene. Figure 8 shows an example of the microarray data, and Figure 9-(a) shows these differences of genes between conditions a and b. Next, we sort these differences in an ascending order, as shown in Figure 9-(b). Then, we scan these sorted values to determine the groups of genes which have the *similar behavior*. That is, for any two genes in one group, the difference of values between them is not larger

gene \ condition	*a*	*b*	*c*	*d*	*e*
0	1	2	3	1	4
1	10	20	30	47	40
2	3	4	5	4	7
3	30	40	50	68	61
4	5	6	8	50	9
5	50	60	71	51	81
6	7	8	100	200	10

Fig. 8. An example microarray data set

than δ. This is the definition of a condition-pair MDS, which means the genes in the same group have similar changes of expression levels under the current two conditions. In the previous example, for those sorted differences shown in Figure 9-(b), there are two groups generated with $\delta = 1$, *i.e.*, {0, 2, 4, 6} and {1, 3, 5}. We also check whether the number of genes in each group is less than NR, where NR defines the minimal number of genes of a pCluster. If the number of genes of one group is less than NR, this group is pruned. In the previous example, assuming that NR is 3, both of {0, 2, 4, 6} and {1, 3, 5} are not pruned. Therefore, we generate a condition-pair MDS, ($\{a, b\}$, {{0, 2, 4, 6}, {1, 3, 5}}). This MDS indicates that under conditions a and b, genes $\{0, 2, 4, 6\}$ have the similar behavior, so do genes $\{1, 3, 5\}$.

However, to efficiently support Steps 2 and 3 later, we have the following two modifications:

(1) For each gene set in the generated MDS, we use a bit string, $b_0 b_1 ... b_{|GenS|-1}$, to record those genes in this set, where b_i is turned on if gene i is in this set. For example, genes $\{0, 2, 4, 6\}$ is represented as 1010101. Therefore,

condition / gene	*a*	*b*	*b* - *a*
0	1	2	1
1	10	20	10
2	3	4	1
3	30	40	10
4	5	6	1
5	50	60	10
6	7	8	1

(a)

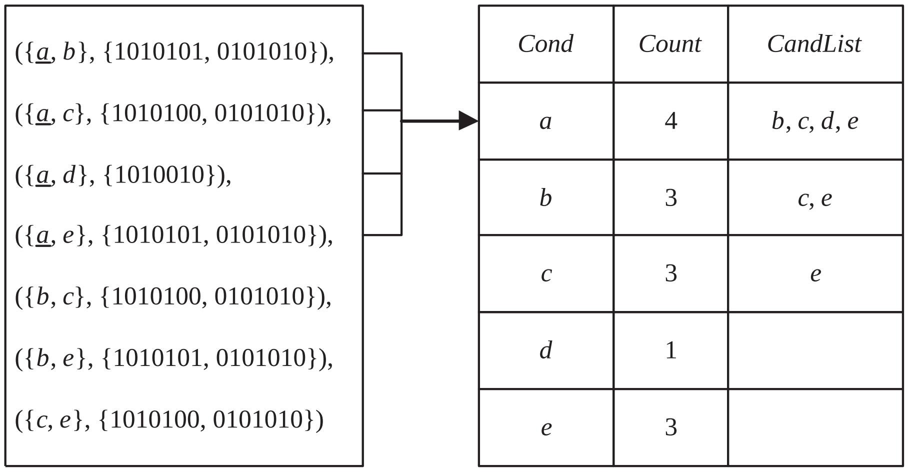
b - *a*	1	1	1	1	10	10	10
gene	0	2	4	6	1	3	5

(b)

Fig. 9. An example of generating condition-pair MDSs: (a) the differences of expression values between conditions a and b; (b) the sorted differences.

in the previous example, the generated MDS is represented as $(\{a, b\}$, $\{1010101, 0101010\})$. (In fact, we also propose a new data structure, the *signature table*, to store these bit strings, which will be described later.)

(2) For each condition pair which appears in the generated MDSs, we utilize a counting table, $CountT$, to count the number of occurrence of each condition. The number of occurrence could help us prune those conditions which are impossible to appear within a pCluster in Step 2. Each tuple of $CountT$ records the information for one condition, x, consisting of two parts: (a) $Count$, which records the number of occurrence of condition x, and (b) $CandList$, which records those conditions related to condition x ($CandList$ will help us expand the enumeration tree in Step 3). Figure 10-(a) shows all the condition-pair MDSs generated from the microarray data shown in Figure 8, and the corresponding counting table is shown in Figure 10-(b). In Figure 10-(b), the value of $Count$ for condition a is 4, since there are 4 condition pairs related to condition a in Figure 10-(a).



({a, b}, {1010101, 0101010}),

({a, c}, {1010100, 0101010}),

({a, d}, {1010010}),

({a, e}, {1010101, 0101010}),

({b, c}, {1010100, 0101010}),

({b, e}, {1010101, 0101010}),

({c, e}, {1010100, 0101010})

(a)

Cond	*Count*	*CandList*
a	4	*b, c, d, e*
b	3	*c, e*
c	3	*e*
d	1	
e	3	

(b)

Fig. 10. The result of Step 1: (a) all condition-pair MDSs; (b) the counting table, $CountT$.

$CandList$ for condition a is a set of all conditions y where $\{a, y\}$ occurs in the generated MDSs. In this example, $CandList$ for condition a is $\{b, c, d, e\}$, since there are 4 condition pairs, *i.e.*, $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, and $\{a, e\}$, occurring in the condition-pair MDSs.

3.2 Step 2: The Pruning Step

In Step 2, we utilize table $CountT$ generated in Step 1 to prune those conditions which are impossible to appear in a pCluster. Figure 11 shows the illustration of this pruning. A pCluster must consist of at least NR genes and at least NC conditions, where NR and NC are determined by users. If there exists one pCluster containing at least NC conditions, all pairs of any combination of these conditions must have been generated in Step 1. Therefore, the number of occurrence of each condition must be not less than $(NC - 1)$, as shown in Figure 11. In the previous example of table $CountT$ shown in Figure 10-(b), assume that NC is 3. Then, condition d will be pruned, since its value

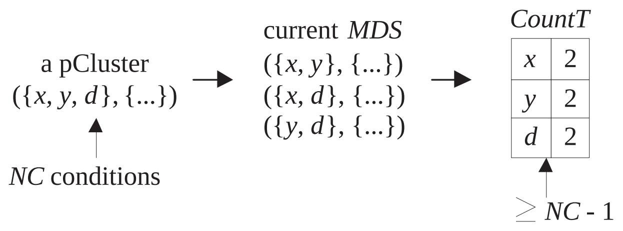
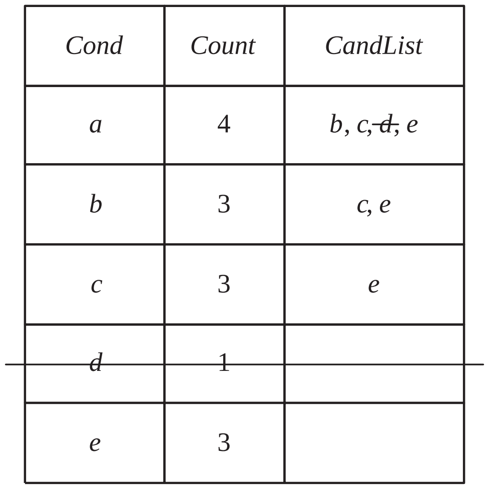


Fig. 11. Pruning of a pCluster based on $CountT$



Cond	*Count*	*CandList*
a	4	*b*, *c*, ~~*d*~~, *e*
b	3	*c*, *e*
c	3	*e*
~~*d*~~	~~1~~	
e	3	

Fig. 12. The result of table $CountT$ after the pruning step

of $Count$ is less than $(NC - 1) = 2$. We will remove condition d wherever it occurs in table $CountT$, as shown in Figure 12.

3.3 Step 3: The Joining Step

In this step, we generate pClusters based on those condition-pair MDSs (as shown in Figure 10-(a)) and table $CountT$ (as shown in Figure 12) generated from the previous 2 steps. We apply the idea of the Condition-Enumeration Tree (*CE-Tree*) for generating pClusters. A pCluster consists of two parts: a condition set and a set of sets of genes (*i.e.*, a set of bit strings in our method). In the proposed method, a condition set is enumerated by the CE-Tree, and its set of bit strings is derived by applying the $\otimes$ operations.

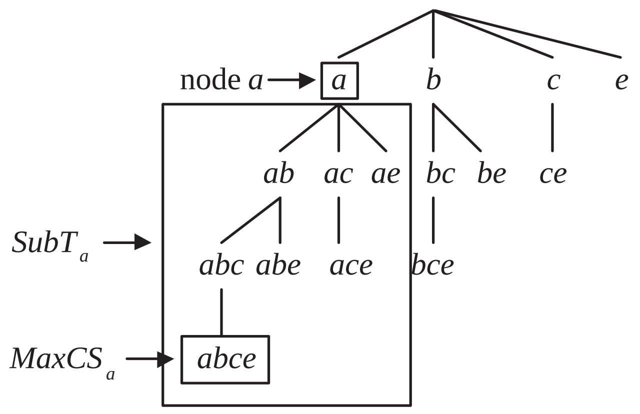


Fig. 13. The CE-Tree for table *CountT* shown in Figure 12

3.3.1 The Condition-Enumeration Tree

The CE-Tree can enumerate all possible combinations of conditions, where each node in this tree shows a possible combination of the conditions (*i.e.*, a condition set). (Note that we enumerate possible combinations of conditions, instead of possible combinations of genes, since the number of the former one is much less than that of the later one.) Figure 13 shows the CE-Tree for those conditions recorded in table *CountT* shown in Figure 12. The subtree under node a, for example, is expanded by enumerating the possible combinations of $Cond$ ($= \{a\}$) and $CandList$ ($= \{b, c, e\}$) from the first tuple of table $CountT$ shown in Figure 12. In the CE-Tree, for node x, we denote its condition set as CS_x, its set of bit strings as GS_x, the subtree under node x as $SubT_x$, and the maximal condition set in $SubT_x$ as $MaxCS_x$, *i.e.*, the condition set in the leaf node which can not be expanded anymore, as shown in Figure 13.

We expand the CE-Tree in a special manner. We call this manner, the *Local Breadth-first within the Global Depth-first* (*LBGD*). (This manner was also applied to mine closed itemsets in (Chi et al., 2004).) Figure 14 shows how the CE-Tree of the previous example shown in Figure 13 is expanded based on the LBGD manner. At level 1 of the CE-Tree, all the condition sets with one

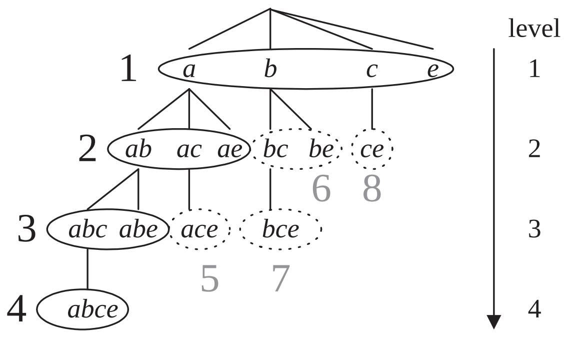


Fig. 14. The LBGD manner

single condition will be expanded according to the alphabetical order. Next, globally, the tree is expanded in the depth-first manner. Therefore, $SubT_a$ will be expanded before expanding $SubT_b$, since node a is expanded before node b. However, in $SubT_a$, all of the child nodes of node a, *i.e.*, ab, ac, and ae, which have a common prefix, a, will be expanded before expanding $SubT_{ab}$. That is, we first expand those nodes $i, (i+1), ..., (i+j)$ at level k of the same subtree under one node. Then, we will sequentially expand $SubT_i, SubT_{i+1}, ..., SubT_{i+j}$, as shown in Figure 14. This traversal order could be done by sequentially selecting a condition set with k conditions at level k as the *join-base*. Then, we try to join other condition sets at the same level k with this join-base, where these condition sets have a common prefix of length $(k-1)$ as the join-base.

Take the CE-Tree shown in Figure 14 as an example. Figure 15 shows the selecting order of the join-bases for this example. First, we have four nodes, a, b, c, and e. Next, we select CS_a as the join-base, and join CS_a with CS_b, CS_c, and CS_e, respectively, to form CS_{ab}, CS_{ac}, and CS_{ae}. (Note that the corresponding information is recorded in the first tuple of table $CountT$ shown in Figure 12.) Then, we select CS_{ab} $(k = 2)$ as the join-base. We will join $CS_{\underline{a}b}$ with $CS_{\underline{a}c}$ and $CS_{\underline{a}e}$, respectively, which have a common prefix of length

$(k-1)$, a, as the join-base, to form $CS_{\underline{a}bc}$ and $CS_{\underline{a}be}$. Then, we select $CS_{\underline{ab}c}$ $(k = 3)$ as the join-base, and try to join it with other condition sets whose prefix is ab, *i.e.*, $CS_{\underline{ab}e}$. After $SubT_{ab}$ is expanded, we select CS_{ac} as the join-base, and join $CS_{\underline{a}c}$ with $CS_{\underline{a}e}$ to expand $SubT_{ac}$. After $SubT_{ac}$ is expanded, originally, CS_{ae} should be the next join-base. However, since there exists no other condition set which could be joined with CS_{ae} at level 2, there exists no $SubT_{ae}$. Therefore, since $SubT_a$ has been expanded completely, we next select CS_b as the join-base and continue to expand $SubT_b$. With the similar process, we can expand the entire CE-Tree.

In fact, the process of the LBGD manner for those nodes whose size of condition sets is grater than two is similar to the idea of the generation of C_{k+1} based on L_k as proposed in the Apriori algorithm for mining association rules (Agrawal and Srikant, 1994). The main property is that two large k-itemsets in L_k could be joined together to form a candidate $(k+1)$-itemsets, if they have a common prefix of length $(k-1)$. Moreover, all subsets of this candidate itemset must also be large itemsets; otherwise, this candidate itemset is impossible to be a large itemset and should be pruned.

Based on the LBGD manner, the CE-Tree has two properties for conditions. In the previous example, for one subset of CS_{abce}, whose length is k $(2 < k < 4)$, its node is expanded before expanding node *abce*, if it has the same prefix of length $(k-1)$ as CS_{abce}. Otherwise, its node will be expanded after node *abce*. We denote this property as the *CE-Tree_cond* property. For example, both of nodes *abc* and *abe* (*i.e.*, $k = 3$) are expanded before node *abce*, since the prefix of CS_{abc} and CS_{abc} of length $(3-1)$, *i.e.*, *ab*, is the same as the prefix of CS_{abce}. However, for any subset of CS_{abce}, whose prefix of length $(k-1)$ is different from the prefix of CS_{abce}, *e.g.*, CS_{ace} or CS_{bce}, its node

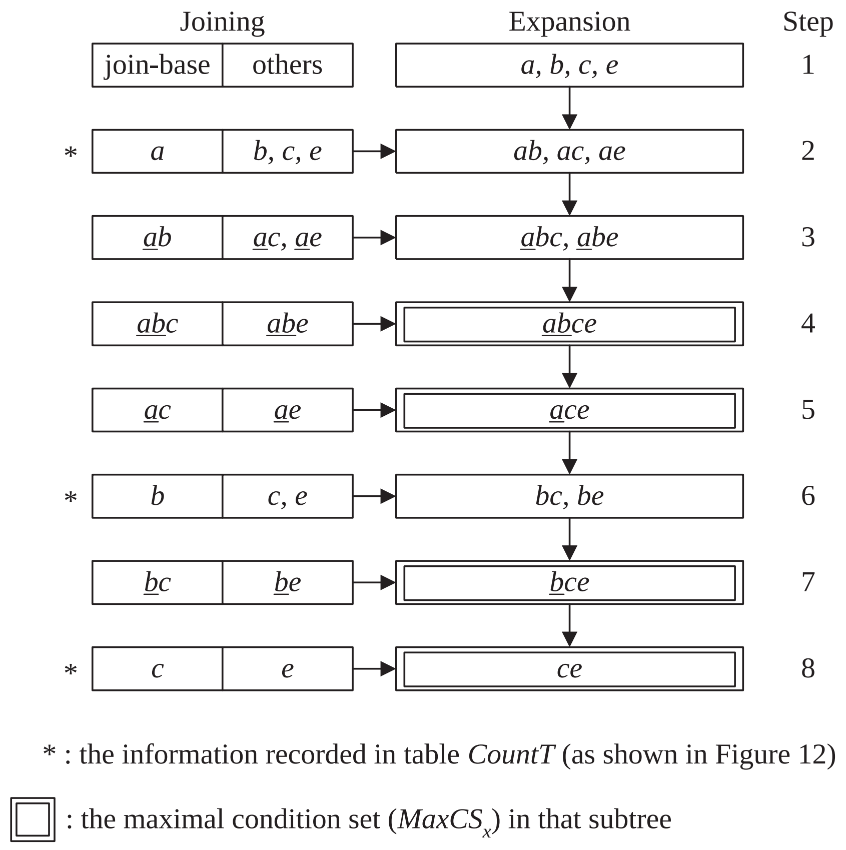


Fig. 15. The joining process of the CE-Tree

will be expanded after expanding node *abce*. Another property of the CE-Tree is that for the join-base CS_x and those condition sets which are joined with CS_x, their union set will be $MaxCS_x$. In the previous example, when CS_{ab} is selected as the join-base, we will join CS_{ab} with CS_{ac} and CS_{ae} to expand $SubT_{ab}$. We could see that $MaxCS_{ab}$ is CS_{abce}, which is the union set of CS_{ab}, CS_{ac}, and CS_{ae}. The reason is that all nodes in $SubT_{ab}$ are expanded step by step from these three ancestor nodes, *i.e.*, *ab*, *ac*, and *ae*. Therefore, $MaxCS_{ab}$ will be equal to the union set of CS_{ab}, CS_{ac}, and CS_{ae}.

In the CE-Tree, after enumerating a new condition set CS_x by the joining process, its corresponding set of bit strings (*i.e.*, its related genes), GS_x, will be derived by applying $\otimes$ operations on the sets of bit strings for those joined condition sets. For example, when we enumerate CS_{abc} by joining CS_{ab} with CS_{ac}, GS_{abc} will be derived by applying $\otimes$ operations on GS_{ab} and GS_{ac}. We will explain how GS_{abc} is derived in the following subsection.

3.3.2 The $\otimes$ Operation

As mentioned in the previous subsection, for any condition set CS_x in the CE-Tree, *e.g.*, CS_{abc}, it is enumerated by joining two subsets of CS_x, *e.g.*, CS_{ab} (*i.e.*, CS_y) and CS_{ac} (*i.e.*, CS_z), which have the same prefix, a, as CS_x. Then, GS_x (*e.g.*, GS_{abc}) will be derived by applying $\otimes$ operations on GS_y and GS_z (*e.g.*, GS_{ab} and GS_{ac}), where $y \subset x$ and $z \subset x$. The derived GS_x will indicate which genes have the similar behavior under CS_x. The $\otimes$ operation used here is the same as that used in the zCluster method (Yoon et al., 2005). Assume that A and B are two sets of bit strings. Then, the result of $A \otimes B$ is

$$A \otimes B = \{(a_i \text{ AND } b_j) | \forall a_i \in A, b_j \in B\}.$$

For example, if $A = \{110, 011\}$ and $B = \{101, 010\}$, $A \otimes B = \{110 \text{ AND } 101, 110 \text{ AND } 010, 011 \text{ AND } 101, 011 \text{ AND } 010\} = \{100, 010, 001, 010\}$.

Take $SubT_a$ shown in Figure 14 as an example. We have found the corresponding MDSs for CS_{ab}, CS_{ac}, and CS_{ae} in Step 1. Figure 16 shows the corresponding bit strings (*i.e.*, the related sets of genes) for these condition sets. Now, we enumerate those condition sets with 3 conditions under node *ab*. They are *abc* and *abe*, enumerated by joining *ab* with *ac* and *ae*, respectively. Then, we want to derive GS_{abc} and GS_{abe}. For GS_{abc}, since CS_{abc} is enumerated by joining CS_{ab} with CS_{ac}, we apply the $\otimes$ operations on GS_{ab} and GS_{ac}, to derive which genes occur in both of these two sets.

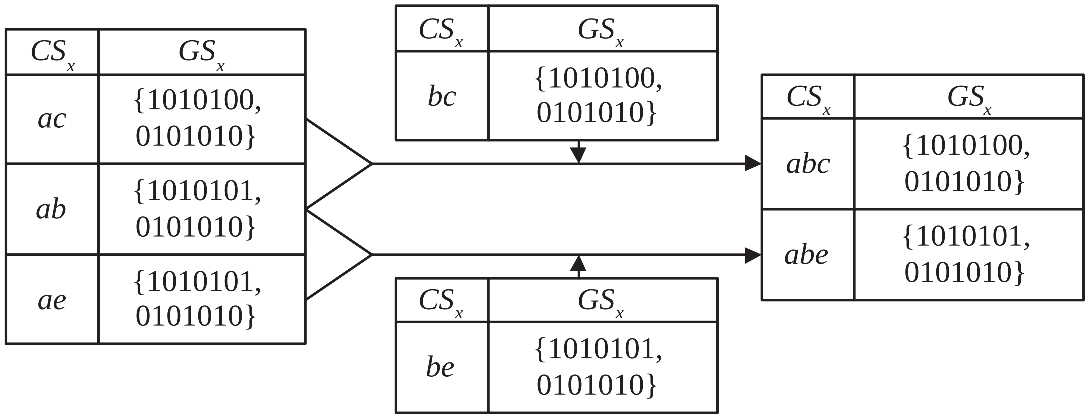


Fig. 16. The corresponding GS_x for CS_{ab}, CS_{ac}, and CS_{ae}

	a	b	$b-a$
gene 1	1	2	1
gene 2	3	5	2

(a)

	a	c	$c-a$
gene 1	1	6	5
gene 2	3	7	4

(b)

	b	c	$c-b$
gene 1	2	6	4
gene 2	5	7	2

(c)

Fig. 17. An example with $\delta = 1$: (a) a condition-pair MDS, ($\{a, b\}$, {gene 1, gene 2}); (b) a condition-pair MDS, ($\{a, c\}$, {gene 1, gene 2}); (c) the difference of $(c-b)$ between these two genes $= 2 > \delta$.

However, when GS_{abc} is derived by $GS_{ab} \otimes GS_{ac}$, one more set of bit strings, GS_{bc}, needs to be considered. The reason is that there may exist some genes similar to each other under not only conditions a and b but also conditions a and c, while they are different from each other under conditions b and c. Figure 17 shows such an example, where the tolerable deviation, δ, is 1. In other words, to ensure that GS_{abc} could be correctly derived, in addition to GS_{ab} and GS_{ac}, the gene set for the other subset of CS_{abc}, *i.e.*, GS_{bc}, also needs to be considered. Therefore, GS_{abc} is derived by

$$GS_{abc} = GS_{ab} \otimes GS_{ac} \otimes GS_{bc}.$$

With the similar process, GS_{abe} could also be derived, as shown in Figure 16. Then, according to the LBGD manner, we will enumerate CS_{abce} by joining

CS_{abc} with CS_{abe}. Originally, GS_{abce} is derived by the following Formula 1:

$$GS_{abce} = GS_{ab} \otimes GS_{ac} \otimes GS_{ae} \otimes GS_{bc} \otimes GS_{be} \otimes GS_{ce}, \tag{1}$$

i.e., the $\otimes$ result of sets of genes for all of the C_2^4 subsets of condition set $\{a, b, c, e\}$. The number of $\otimes$ operations used is 5. The zCluster method (Yoon et al., 2005) improves this derivation by replacing $GS_{ab} \otimes GS_{ac} \otimes GS_{bc}$ with GS_{abc}, resulting in the following Formula 2:

$$GS_{abce} = GS_{abc} \otimes GS_{ae} \otimes GS_{be} \otimes GS_{ce}. \tag{2}$$

Therefore, the number of $\otimes$ operations is reduced to 3. In the proposed CE-Tree method, we further improve the above derivation by replacing $GS_{ae} \otimes GS_{be}$ with GS_{abe}, resulting in the following Formula 3:

$$GS_{abce} = GS_{abc} \otimes GS_{abe} \otimes GS_{ce}. \tag{3}$$

Note that GS_{abe} is derived by $GS_{ab} \otimes GS_{ae} \otimes GS_{be}$. As compared to the replaced part, *i.e.*, $GS_{ae} \otimes GS_{be}$, GS_{abe} considers one more set of bit strings, *i.e.*, GS_{ab}. However, GS_{ab} is originally considered in Formula 1, which does not affect the final result of GS_{abce}. (The size of GS_{abe} is also less than or equal to the size of $GS_{ae} \otimes GS_{be}$, since GS_{abe} considers one more set of bit strings.) Therefore, the number of $\otimes$ operations is further reduced to 2, which is less than the number of $\otimes$ operations needed in the zCluster method, *i.e.*, 3.

We can derive GS_{abce} in such a way due to the LBGD manner. When node *abce* is expanded, we have already expanded nodes *abc* and *abe*. Therefore, we could utilize GS_{abc} and GS_{abe} to help us derive GS_{abce}. Note that in the zCluster method, it uses a depth-first manner to expand its tree. When node *abce* is expanded, node *abe* has not been expanded yet. Therefore, the zCluster method can not utilize GS_{abe} to derive GS_{abce}.

Table 2

A comparison of the number of $\otimes$ operations

method	$abcd$	$abcde$	$c_1c_2c_3...c_{n-2}c_{n-1}c_n$
zCluster	$abc\otimes$ $\underline{ad \otimes bd\otimes}$ cd (# of $\otimes$: 3)	$abcd\otimes$ $\underline{ae \otimes be \otimes ce\otimes}$ de (# of $\otimes$: 4)	$c_1c_2c_3...c_{n-2}c_{n-1}\otimes$ $\underline{c_1c_n \otimes c_2c_n \otimes ... \otimes c_{n-2}c_n\otimes}$ $c_{n-1}c_n$ (# of $\otimes$: $n-1$)
CE-Tree	$abc\otimes$ $\underline{abd\otimes}$ cd (# of $\otimes$: 2)	$abcd\otimes$ $\underline{abce\otimes}$ de (# of $\otimes$: 2)	$c_1c_2c_3...c_{n-2}c_{n-1}\otimes$ $\underline{c_1c_2c_3...c_{n-2}c_n\otimes}$ $c_{n-1}c_n$ (# of $\otimes$: 2)

Table 2 shows a comparison of the number of $\otimes$ operations needed between the zCluster and the CE-Tree method, where the underline parts show the differences of the formulas between the zCluster method and the proposed method. From this table, we can see that the information of the underline part of one formula in the proposed method always contains that in the zCluster method. To derive GS_x for CS_x which contains n conditions, the number of $\otimes$ operations needed by the zCluster method is $(n-1)$, while this number is always 2 in the CE-Tree method. Therefore, with such a technique, we could reduce the number of $\otimes$ operations needed, and improve the efficiency of the joining process.

With a similar process, we can enumerate all the combinations of condition sets shown in Figure 14, and find all MDSs for these condition sets by performing the $\otimes$ operations. After performing the $\otimes$ operations, those derived bit strings whose number of on-bits is less than NR are pruned. The reason is that the number of on-bits in a bit string means the number of genes which have similar behavior under one condition set. If there is no bit string for one condition set, we will not join this condition set with other condition sets. (In other words, we can prune such a node from the CE-Tree.)

For each generated MDS (CS_x, GS_x), it is added to the set of pClusters, PC, if it satisfies the following definitions of a pCluster:

(1) The size of CS_x is not less than NC.

(2) GS_x contains at least one bit string with at least NR on-bits, which means the number of genes is at least NR.

(3) There does not exist another pCluster which contains this MDS; that is, a pCluster must be the maximal one.

In the CE-Tree, we expand all the nodes based on the LBGD manner, and derive the information of genes of each node by the $\otimes$ operations. The CE-Tree has two properties for conditions as mentioned in the previous subsection. Moreover, the CE-Tree has one property for genes. This property is that for node x, the number of on-bits of any bit string in GS_x will not be greater than that in GS_y, where node y is one of the ancestor nodes of node x. (We denote this property as the *CE-Tree_gene* property.) The reason is that one $\otimes$ operation contains several AND operations. The number of on-bits of a bit string will decrease, as the number of times of the $\otimes$ operations that are applied increases. In the following subsection, we will utilize the properties of the CE-Tree to develop three bounding techniques for the expansion of the CE-Tree.

3.3.3 The Bounding Techniques

By expanding the entire CE-Tree as mentioned previously, we can find all the pClusters. However, there may exist some nodes which do not need to be expanded. Therefore, to avoid exhaustively enumerating, we further develop a branch-and-bound strategy to efficiently enumerate these condition sets in the CE-Tree.

Assume that CS_X of node X at level i is the current join-base, and (CS_Y, GS_Y) is a pCluster found previously. We have the following three bounding situations to avoid expanding the subtree under node X:

Bound 1: The size of $MaxCS_X < NC$.

Bound 2: $CS_Y \supset MaxCS_X$ and $GS_Y = GS_X$.

Bound 3: $CS_Y \supset MaxCS_X$, $GS_Y \neq GS_X$, but $GS_Y = GS_Z$, where CS_Z is any one of those condition sets joined with CS_X at the same level i.

The reason for the technique of Bound 1 is that since $MaxCS_X$ (*i.e.*, the maximal condition set in the subtree under node X) contains less than NC conditions, it is impossible to generate a pCluster containing at least NC conditions in this subtree. There are two cases for this situation. The first case occurs in those last $(NC-i)$ nodes at level i in the same subtree, where i is also the size of a condition set at level i. Figure 18 shows an example for the first case of Bound 1, where this tree is the same CE-Tree as shown in Figure 14. In this example, assume that NC is 3. For the last $(3-1)$ nodes at level 1 of the tree shown in Figure 18, *i.e.*, nodes c and e, we do not further expand their subtrees, since both of the lengths of $MaxCS_c$ and $MaxCS_e$ will be less than 3. If NC is 4, all of $SubT_b$, $SubT_c$, and $SubT_e$, *i.e.*, subtrees under the last $(4-1)$ nodes at level 1, will not be expanded.

The second case of Bound 1 will occur if there is no bit string of related genes with at least NR on-bits for any node at level i. In this case, this node is pruned, which may result in a change of the last $(NC-i)$ nodes at level i. Then, the second case of Bound 1 becomes the first case of Bound 1 after pruning that node. Figure 19 shows an example of this case. In this example,

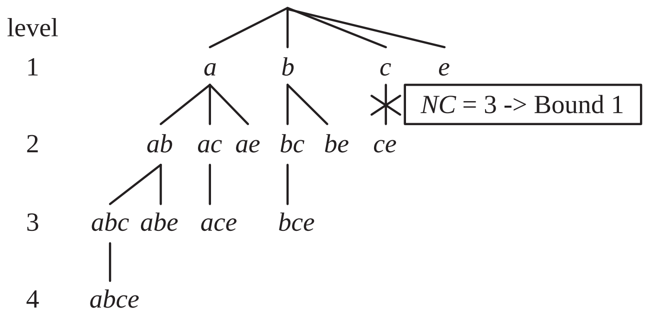


Fig. 18. The CE-Tree after applying the technique of Bound 1

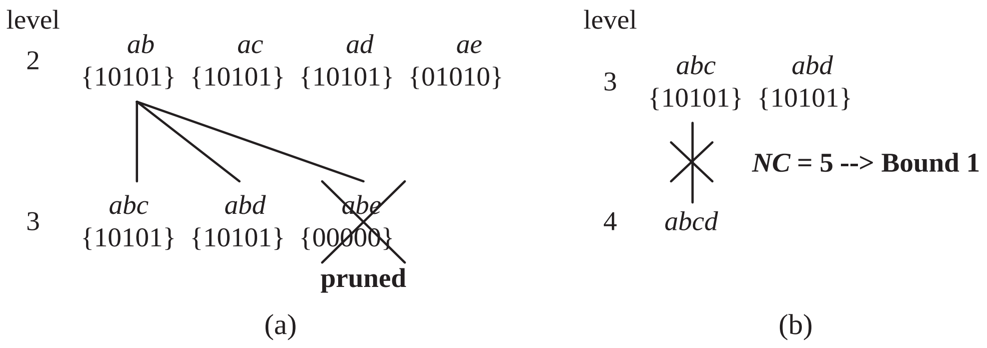


Fig. 19. The situation of Bound 1: (a) pruning *abe*; (b) applying the technique of Bound 1 if $NC = 5$.

assume that there are three nodes, *i.e.*, *abc*, *abd*, and *abe*, at level 3, as shown in Figure 19-(a). After performing the $\otimes$ operations, we find that there is no bit string of genes (with at least NR on-bits) for node *abe*, and node *abe* is pruned. Therefore, there exist only two nodes, *abc* and *abd*, at level 3. If NC is 5, we do not need to further expand subtrees under the last $(5-3)$ nodes, *i.e.*, $SubT_{abc}$ and $SubT_{abd}$, at level 3, as shown in Figure 19-(b).

The reason for the technique of Bound 2 is that when the situation of Bound 2 is satisfied, any MDS generated in $SubT_X$ must have been already contained by pCluster (CS_Y, GS_Y). (Note that the current join-base is CS_X.) According to the $CE\text{-}Tree\_gene$ property, the number of on-bits of one bit string in GS_X must be greater than or equal to the number of on-bits of one bit string at any node of $SubT_X$. Moreover, $MaxCS_X$ means the maximal condition set in

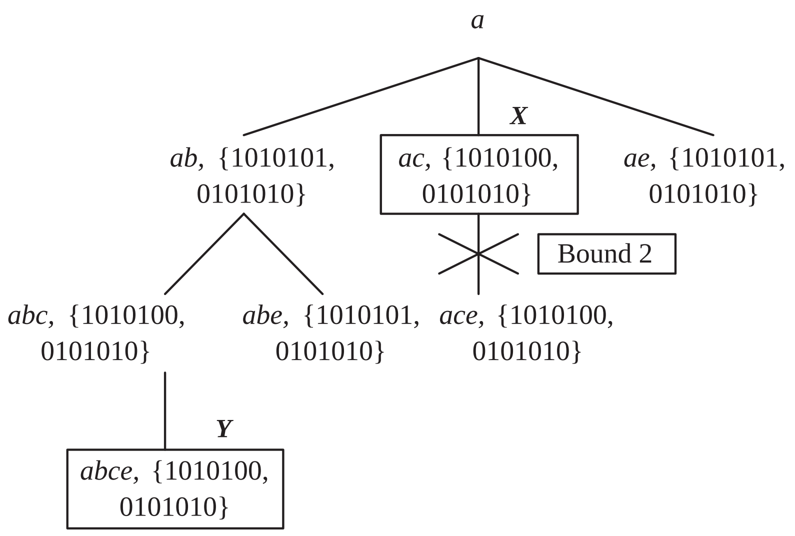


Fig. 20. An example of the situation of Bound 2

$SubT_X$. Therefore, if there exists pCluster (CS_Y, GS_Y), where CS_Y contains $MaxCS_X$ and GS_Y equals GS_X, the MDS generated from any node of $SubT_X$ must be contained by this pCluster. Since a pCluster must be the maximal one, those MDSs contained by a pCluster can not be pClusters, and we do not need to expand these nodes in the CE-Tree. For example, assume that we have found a pCluster, $(CS_Y, GS_Y) = (\{a, b, c, e\}$, {1010100, 0101010}), as shown in Figure 20. According to the LBGD expanding manner, next, we will expand $SubT_{ac}$ (*i.e.*, $CS_X = CS_{ac}$). However, the expansion of this subtree can be bounded by the technique of Bound 2. The reason is that there has already existed a pCluster, $(\{a, b, c, e\}, \{1010100, 0101010\})$, where $CS_Y = CS_{abce} = \{a, b, c, e\}$ contains $MaxCS_X = MaxCS_{ac} = \{a, c, e\}$, and $GS_Y = GS_{abce} = \{1010100, 0101010\}$ equals to $GS_X = GS_{ac}$. Therefore, it is impossible to generate a new pCluster in $SubT_X = SubT_{ac}$, and we do not further expand this subtree.

The reason for the technique of Bound 3 is similar to that of Bound 2. The difference between Bound 2 and Bound 3 is that when CS_X is the join-base,

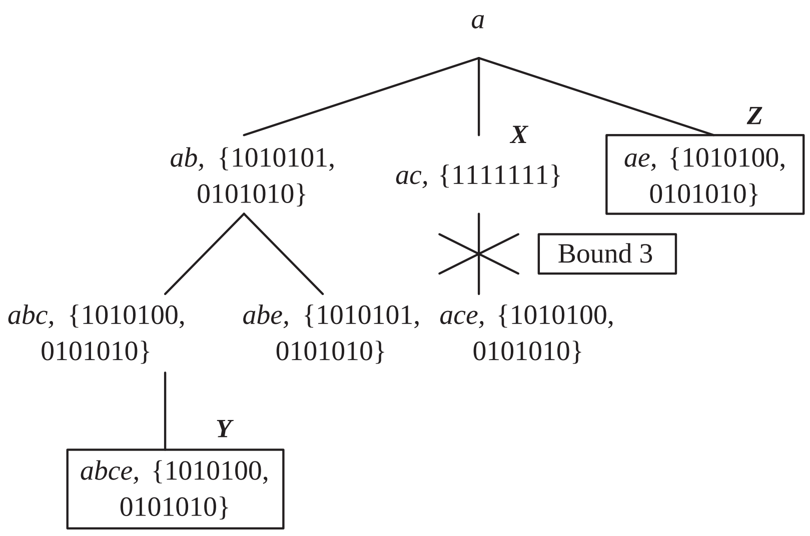


Fig. 21. An example of the situation of Bound 3

Bound 2 considers GS_X, while Bound 3 considers GS_Z, where CS_Z is any of those condition sets joined with CS_X. The set of bit strings at any node of $SubT_X$ is derived by applying $\otimes$ operations on GS_X and all GS_Z. According to the *CE-Tree_gene* property, if there exists pCluster (CS_Y, GS_Y), where CS_Y contains $MaxCS_X$ and GS_Y equals GS_Z, the MDS generated from any node of $SubT_X$ must also be contained by this pCluster. (This case will occur even if $GS_Y \neq GS_X$.) Figure 21 shows such an example. In this example, the join-base is $CS_X = CS_{ac}$, and $CS_Y = CS_{abce} = \{a, b, c, e\}$ contains $MaxCS_X = MaxCS_{ac} = \{a, c, e\}$. (Note that CS_Z is CS_{ae} in this example.) Although $GS_{abce} = \{1010100, 0101010\}$ is not equal to GS_{ac} (*i.e.*, $GS_Y \neq GS_X$), GS_{abce} is equal to GS_{ae} (*i.e.*, $GS_Y = GS_Z$). Therefore, the situation of Bound 3 is satisfied. We could see that in this case, for the set of bit strings at any node of $SubT_X = SubT_{ac}$, *i.e.*, GS_{ace} in this example, the number of on-bits of any bit string in GS_{ace} will be limited by $GS_Z = GS_{ae}$, since $GS_{ace} = GS_{ac} \otimes GS_{ae} \otimes GS_{ce}$.

The techniques of Bound 2 and Bound 3 could be applied due to the *CE*-

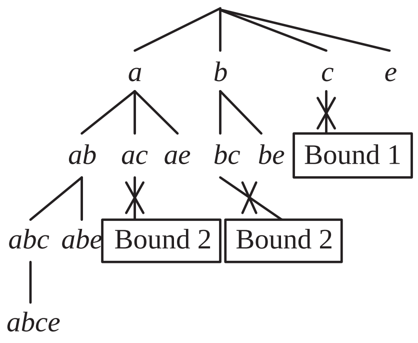


Fig. 22. The CE-Tree after applying the bounding techniques

$Tree\_cond$ property. That is, CS_{abce} is enumerated before some of its subsets, *e.g.*, CS_{ace} in this example. Therefore, after finding a pCluster, we could early bound the expansion of those nodes whose condition sets and sets of bit strings will be contained by this pCluster, *i.e.*, branch-and-bound. Figure 22 shows the final CE-tree for the previous example. As compared to a complete enumeration tree with 5 items, whose total number of nodes is $2^5 = 32$, we only expand 12 nodes based on the proposed pruning and bounding techniques. The resulting reduction rate is $(32-12)/32 = 62.5\%$. Therefore, we could efficiently expand the tree without exhaustively enumerating all possible combinations.

3.4 Improving the Joining Process

As mentioned previously, we apply the $\otimes$ operations on those sets of bit strings to derive which genes have similar behavior under the current enumerating conditions. Since $A \otimes B = \{(a_i \text{ AND } b_j) | \forall a_i \in A, b_j \in B\}$, this operation is similar to the traditional join operation. Therefore, we can apply a strategy similar to the hash join approach (DeWitt et al., 1984; Kitsuregawa et al., 1983) to reduce the number of AND operations needed. For example, in Figure 23-(a), assume that A and B are two sets of bit strings, where each of them

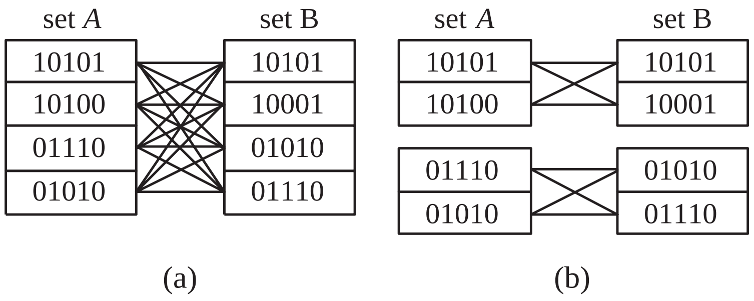


Fig. 23. An example of joining two sets: (a) directly joining; (b) grouping bit strings first and then joining.

contains 4 bit strings. Then, to derive the result of $(A \otimes B)$, it will need $(4*4) = 16$ AND operations. If we group those similar bit strings (*i.e.*, those bit strings which have enough on-bits in the same positions) together first and apply $\otimes$ operations only on those similar bit strings, the number of AND operations needed will be decreased. For example, in Figure 23-(b), according to the distribution of on-bits in these bit strings, we could separate the bit strings in each of sets A and B into two groups. Then, we apply the $\otimes$ operations only on those similar groups between sets A and B. Therefore, the number of AND operations needed becomes $(2 * 2 + 2 * 2) = 8$, which is only a half of the original number of AND operations needed.

3.4.1 The Signature Table

Based on the idea mentioned previously, in the proposed method, for the set of bit strings in each node of the CE-Tree, we utilize a new structure, the *signature table* (table SG), to group them. In table SG, we try to group those bit strings within one node of the CE-Tree into different tuples, *i.e.*, one group per tuple. Those bit strings which have enough on-bits in the same positions are stored in the same group (tuple). Then, for each group, we utilize a signature to represent the distribution of on-bits of bit strings in this group.

By simply checking the signature of one group, we could know what type of bit strings may be in this group, and avoid joining it with other dissimilar groups whose signatures are far different from its signature.

Table SG consists of H tuples, where H is a parameter which could be dynamically adjusted according to the usable memory. Each tuple consists of two parts: the signature, Sig, which is the OR result of all bit strings stored in this tuple, and the set of bit strings, BSS. Let $|b_0b_1...b_{m-1}|$ mean the number of on-bits among bits $b_i, 0 \leq i \leq (m-1)$. Then, we use a simple rule to decide whether a new bit string, BS, could be added into one group (tuple), GP, as follows.

Rule 1 *If* $(|BS \text{ OR } GP.Sig| - |GP.Sig|) \leq T$*, we say that bit string BS is similar to group* GP *and can be added into* GP*, where* $GP.Sig$ *is the signature of* GP *and* T *is the threshold assigned by the user.*

This rule means that if the increment of the number of on-bits between the result after applying the OR operation and the original signature is still inside a threshold, T, we consider them as the same group. Otherwise, they will be in different groups. For example, assume that $T = 1$ and there are three bit strings. Initially, table SG is empty. For the first bit string, “10100”, it is directly stored in the first tuple of table SG. The signature of the first tuple is also set to this bit string. For the second bit string, “10101”, it can be added to the first tuple, since $(|10101 \text{ OR } 10100| - |10100|) = (|10101| - |10100|) = 3 - 2 = 1 \leq 1$. After this bit string is added into the first tuple, the signature of this tuple is also set to the OR result, *i.e.*, “10101” OR “10100” = “10101”. For the third bit string, “01010”, it can not be added into the first tuple, since $(|01010 \text{ OR } 10101| - |10101|) = 5 - 3 = 2 > 1$. Therefore, “01010” is stored in a new tuple, and the signature of this new tuple is also set to “01010”.

According to Rule 1, bit strings in each node of the CE-Tree will be added into the signature table of this node. If table SG is full and BS is not similar to any one of groups in this table, we store BS in tuple t, where the value of $(|BS \text{ OR } t.Sig| - |t.Sig|)$ is the smallest one among all tuples.

3.4.2 The New Joining Process

After creating these signature tables, we could utilize them to improve the joining process. For example, in the CE-Tree shown in Figure 14, originally, we derive GS_{abc} by $GS_{ab} \otimes GS_{ac} \otimes GS_{bc}$. Now, we apply a new function, $TableJoin$, to derive the result by utilizing the signature tables for GS_{ab}, GS_{ac}, and GS_{bc}.

Figure 24 shows function $TableJoin$. Assume that A and B are two sets of bit strings, and $SGTable_A$ and $SGTable_B$ are the signature tables for set A and set B, respectively. In function $TableJoin$, we will try to generate a new signature table, which stores the result of $A \otimes B$, by joining $SGTable_A$ and $SGTable_B$. For tuple $t_1 \in SGTable_A$ and tuple $t_2 \in SGTable_B$, if the number of on-bits of the result of $(t_1.Sig \text{ AND } t_2.Sig)$ is not less than NR, we add the result of $(t_1.BSS \otimes t_2.BSS)$ to the result set of bit strings, $ResultBS$. The reason is that the signature of each tuple, Sig, is the OR result of all bit strings stored in this tuple. If the number of on-bits of the result of $(t_1.Sig \text{ AND } t_2.Sig)$ is less than NR, it is impossible to generate a bit string with at least NR on-bits by applying the AND operation on one bit string in t_1 and another bit string in t_2.

Figure 25 shows an example of two signature tables, $SGTable_A$ and $SGTable_B$.

```
Function $TableJoin(SGTable_A, SGTable_B)$: a signature table;
begin
    foreach tuple $t_1 \in SGTable_A$ do
        foreach tuple $t_2 \in SGTable_B$ do
            if $(|t_1.Sig$ AND $t_2.Sig| \geq NR)$ then
                add $t_1.BSS \otimes t_2.BSS$ to $ResultBS$;
    Let $NewSG$ be a new signature table;
    foreach bit string $bs \in ResultBS$ do
        if $(|bs| \geq NR)$ then
            $InsertToSGTable(bs, NewSG)$;
    return $NewSG$;
end;
```

Fig. 24. Function *TableJoin*

Assume that NR is 3. For the first tuples of $SGTable_A$ and $SGTable_B$, their Sig are "1010101" and "1010100", respectively. Since |1010101 AND 1010100| $= 3 \geq NR$, we will apply the $\otimes$ operation on their BSS. However, for the first tuple of $SGTable_A$ and the second tuple of $SGTable_B$, their BSS can not be applied with the $\otimes$ operation, since |1010101 AND 0101010| $= 0 < NR$. We could see that in this case, for any bit string in the first tuple of $SGTable_A$, *e.g.*, "1010100", and any bit string in the second tuple of $SGTable_B$, *e.g.*, "0001010", the number of on-bits in their AND result must be less than NR $(= 3)$. In this example, the total number of AND operations needed is $(2*2+4*4+4*4) = 36$, where $2*2$ is for checking the signatures and $4*4$ is for each of the $\otimes$ operations. If we do not utilize the signature table, the number of AND operations needed by directly applying the $\otimes$ operation on those bit strings in $SGTable_A$ and $SGTable_B$ will be $(8*8) = 64$, which is almost twice as large as the number of AND operations with a signature table.

In function *TableJoin* shown in Figure 24, all of the $\otimes$ result are stored in a

TID	*Sig* (the signature)	*BSS* (the set of bit strings)
0	1010101	1010101, $\underline{1010100}$, 1000101, 1010001
1	0101010	0101010, 0001010, 0101000, 0100010

(a)

TID	*Sig* (the signature)	*BSS* (the set of bit strings)
0	1010100	1010100, 0010100, 1000100, 1010100
1	0101010	0100010, 0101000, $\underline{0001010}$, 0100010

(b)

Fig. 25. An example of the signature tables: (a) $SGTable_A$; (b) $SGTable_B$.

new set of bit strings, $ResultBS$. Then, we also generate a new signature table, $NewSG$, and apply procedure $InsertToSGTable$ to insert each bit string in $ResultBS$ into $NewSG$, as mentioned in Subsection 3.4.1. Finally, $NewSG$ is returned as the function result. Therefore, when we join *ab*, *ac*, and *bc* to form *abc* in the CE-Tree shown in Figure 14, the new signature table for condition set *abc*, SGT_{abc}, is derived by

$$SGT_{abc} = TableJoin(TableJoin(SGT_{ab}, SGT_{ac}), SGT_{bc}),$$

where SGT_{ab}, SGT_{ac}, and SGT_{bc} are signature tables for *ab*, *ac*, and *bc*, respectively. By utilizing the signature tables, we could efficiently derive the set of bit strings in each node of the CE-Tree, since the number of AND operations needed will be decreased significantly. Table 3 summarizes the improvements of the CE-Tree method as compared to those previous proposed methods which generate gene-pair MDSs and perform the complex duplicating processes.

4 Performance

In this section, we study the performance of the CE-Tree method. We implemented the CE-Tree method in Java, and performed all experiments on a Fedora Linux virtual machine (512 MB of memory, an 1.79 GHz Intel-compatible

Table 3

The summarization of improvements of the CE-Tree method

Step / Method	Constructing MDSs		Expanding the tree	Performing processes at each node		Reducing the number of expanding nodes	
	Gene-pair MDSs	Condition-pair MDSs	The help of gene-pair MDSs	Joining	Duplicating	Pruning	Bounding
pClustering MaPle zCluster	Yes	Yes	Yes	Yes	Yes	Yes	No
CE-Tree	No	Yes	No	Yes *	No	Yes	Yes

*: A new data structure to improve this process

processor) over Windows XP through VMware middleware.

4.1 Experiment Data

To analyze the performance of the proposed method, we use several synthetic data sets and real microarray data sets as the experiment data. We generate synthetic data sets by using a method similar to that used in (Wang et al., 2002). Initially, a synthetic data set is simulated by a 2-dimensional matrix with random values ranged from 0-1500. Then, we embed a fixed number of $perfect$ pClusters ($i.e.$, $\delta = 0$) into this matrix (Wang et al., 2002). The related parameters are shown in Table 4.

Table 5 shows the real microarray data sets used in our experiments. The yeast microarray data set (Tavazoie et al., 1999) is widely used in microarray clustering research, obtained from the yeast *Saccharomyces Cerevisiae* cell cycle expression levels. The SRBCT (Small, Round Blue Cell Tumors) microarray data set (Khan et al., 2001) consists of expression levels from the small, round blue cell tumors of childhood, where most of values of these expression levels are less than 1. The ALL-AML microarray data set (Brunet et al., 2004) is often used in microarray classification research, composed of samples from

Table 4

Parameters used in the generation of synthetic data

Parameter	Description
r	The number of rows of the matrix
c	The number of columns of the matrix
k	The number of embedded pClusters
nr	The number of rows of an embedded pCluster
nc	The number of columns of an embedded pCluster

Table 5

The real microarray data sets

Name	The number of genes	The number of samples
Yeast	2884	17
SRBCT	2308	83
ALL-AML	5000	38

27 ALL (*acute lymphoblastic leukemia*) patients and 11 AML (*acute myeloid leukemia*) patients.

4.2 Accuracy of the CE-Tree Method

In this subsection, we experiment the accuracy of the CE-Tree method. We use several synthetic data sets as the experiment data. Figure 26-(a) shows an example of a simple synthetic data set with $r = 10, c = 5, k = 3, nr = 3$, and $nc = 3$. Since there are k perfect pClusters embedded in each synthetic data

Table 6

A comparison of the number of pClusters found

r	c	nr	nc	k	The CE-Tree method	zCluster
3000	30	30	6	30	30	5
3000	30	30	6	40	40	4
3000	30	30	6	50	50	2
5000	50	50	10	10	10	8
5000	50	50	10	20	20	2

set, there should be k pClusters found with $\delta = 0$. Figures 26-(b) and 26-(c) show these k $(= 3)$ pClusters in this synthetic data set. The proposed method can find all these pClusters. However, the zCluster method can not find the pCluster shown in Figure 26-(c), where the executable file of the zCluster method is downloaded from the *url* given in (Yoon et al., 2005). Table 6 shows other synthetic data sets and the experiment results of the CE-Tree method and the zCluster method. We could see that for these data sets, the CE-Tree method could find the same number of pClusters as the value of k. However, with unknown reasons, the numbers of pClusters found by the zCluster method are all less than k, which seems to have some missing cases.

4.3 Efficiency of the CE-Tree Method

In this subsection, we experiment the efficiency of the CE-Tree method. First, we compare the execution time between generating object-pair MDSs applied in most of the previous proposed methods (Wang et al., 2002; Yoon et al., 2005)

gene \ condition	0	1	2	3	4
0	91	658	498	664	667
1	337	259	451	25	375
2	219	194	245	200	203
3	37	564	756	254	680
4	391	510	446	380	598
5	270	244	436	26	360
6	325	90	6	378	62
7	0	119	181	443	207
8	457	576	481	338	664
9	161	570	360	576	579

(a)

gene \ condition	0	1	4
4	391	510	598
7	0	119	207
8	457	576	664

gene \ condition	1	2	4
1	259	451	375
3	564	756	680
5	244	436	360

(b)

gene \ condition	1	3	4
0	658	664	667
2	194	200	203
9	570	576	579

(c)

Fig. 26. An example of the missing case in the zCluster method: (a) the original microarray data; (b) pClusters found by the zCluster method; (c) the missing pCluster.

and the CE-Tree method. Next, we compare the execution time between the MicroCluster method (Zhao and Zaki, 2005) and the CE-Tree method.

4.3.1 Generating Object-Pair MDSs

Both in (Wang et al., 2002) and (Yoon et al., 2005), the process of generating pClusters consists of the following steps: (1) generating the object-pair MDSs; (2) generating the condition-pair MDSs; (3) using these found MDSs to generate pClusters. The time complexity of Step 1, *i.e.*, generating object-pair MDSs, is $O(N^2 M \log M)$, where N is the number of genes and M is the number of conditions. However, in microarray data, the number of genes (objects) is much larger than the number of conditions (columns). Generating object-pair MDSs needs much longer time than generating condition-pair MDSs. In this subsection, we compare the time of generating object-pair MDSs with the total execution time of the proposed method. For generating object-pair MDSs, we implement Algorithm 1 described in (Yoon et al., 2005), and we use the same quick-sort method for sorting data.

Figure 27 shows the simulation results of the execution time for synthetic data sets. These synthetic data sets are generated by the same parameters as those in (Wang et al., 2002), where k is 30 and δ is 3. Figure 27-(a) shows the result of the execution time with parameters $c = 30, nr = 0.01 * r, nc = 6$, and varying r. Figure 27-(b) shows the result of the execution time with parameters $r = 3000, nr = 30, nc = 0.1*c$, and varying c. From Figure 27, we could observe that generating object-pair MDSs needs even longer time than the CE-Tree method, when the number of objects or columns is large. This is because we generate only the condition-pair MDSs. Moreover, when we generate pClusters by these condition-pair MDSs, we apply several techniques to prune or bound many situations which do not need to be considered. Therefore, the proposed method could find the pClusters efficiently.

Figure 28 shows the experiment results for the real microarray data sets, where Figures 28-(a) and 28-(b) are for the SRBCT microarray data set (Khan et al., 2001) and the ALL-AML microarray data set (Brunet et al., 2004), respectively. We apply 5 different cases of parameters (δ, NR, NC) for each of these two microarray data sets. From this figure, we could observe that

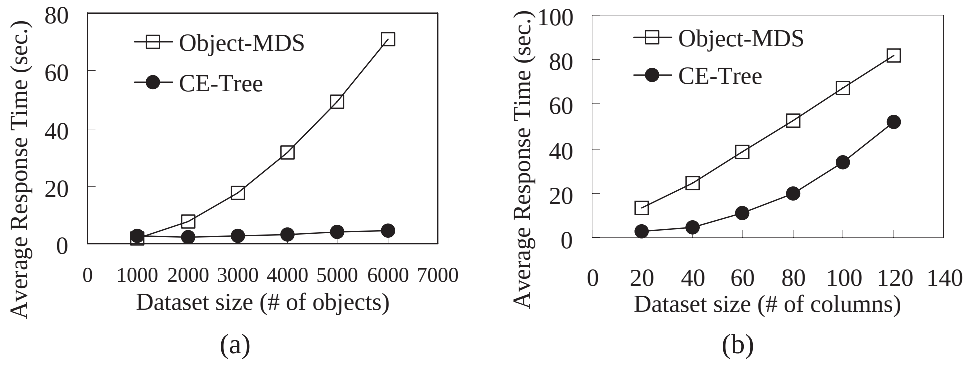


Fig. 27. A comparison of execution time between generating object-pair MDSs and the proposed method: (a) varying the number of objects; (b) varying the number of columns.

the whole process of the CE-Tree method needs less time to find pClusters than the process of generating only object-pair MDSs for these data sets. The reasons are the same as those mentioned previously. Note that the time for generating object-pair MDSs is only affected by the size of the microarray data. Therefore, in Figure 28, the time for generating object-pair MDSs is fixed for each of these two microarray data sets, no matter what values the three parameters are.

4.3.2 The MicroCluster Method

In this subsection, we compare the execution time between the MicroCluster method (Zhao and Zaki, 2005) and the CE-Tree method. The source code of the MicroCluster method is kindly given by the original authors. This source code is written in C++. However, there have been many researches pointing out that on Intel-based hardware, especially with Linux, the performance gap between C++ and Java is small enough to be of little or no concern to programmers (Bull et al., 2001). In order to avoid a bad implementation of the MicroCluster method, we directly compare the execution time of this C++

δ	NR	NC	pClusters found	CE-Tree (sec)	Object-pair MDS (sec)
0.005	30	3	19	14.41	
0.01	50	3	105	19.14	
0.015	75	3	23	19.24	31.226
0.02	100	3	5	19.51	
0.025	125	3	12	20.17	

(a)

δ	NR	NC	pClusters found	CE-Tree (sec)	Object-pair MDS (sec)
0	1200	3	80	4.8	
5	1300	3	2	6	
10	1300	3	953	10.9	61.7
20	1400	3	4	15.6	
30	1475	3	241	19.2	

(b)

Fig. 28. The experiment results for real microarray data sets: (a) SRBCT; (b) ALL-AML.

code with that of the CE-Tree method written in Java.

Since the MicroCluster method originally finds scaling biclusters, we slightly modify Step 1 of our CE-Tree method to find the same biclusters. That is, for any two conditions, we evaluate the ratio of their expression levels of each gene, instead of originally evaluating the difference. We use the real yeast microarray data (Tavazoie et al., 1999) as the input data. The basic parameters used are $\delta = 0.001$, $NR = 50$, and $NC = 3$. Figure 29 shows the results of the execution time of the MicroCluster method and the CE-Tree method, where Figures 29-(a), 29-(b), and 29-(c) show the results of varying δ, NR, and NC, respectively. From these figures, we could observe that the CE-Tree method outperforms the MicroCluster method in terms of the execution time. Although the MicroCluster method also generates only the condition-pair MDSs, it needs to solve the *Maximal Clique* problem, a well-known NP-Complete problem. Therefore,

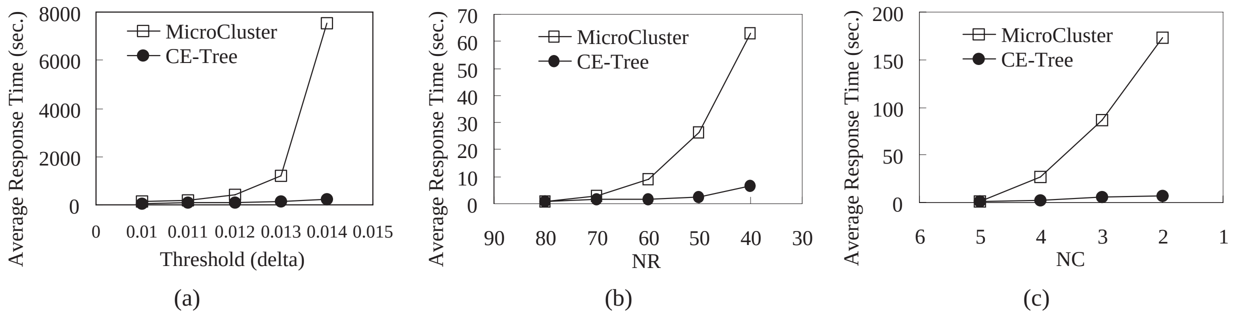


Fig. 29. The results of the execution time of the MicroCluster method and the CE-Tree method: (a) varying δ; (b) varying NR; (c) varying NC.

in Figure 29, we could observe the execution time of the MicroCluster method increases quickly. For the proposed CE-Tree method, in the worst case, the time complexity of generating pClusters is also the same as that of solving a NP-Complete problem, *i.e.*, $O(2^M)$. However, we develop several pruning and bounding techniques, which could efficiently prevent the worst case occurring.

4.4 Discussion on pClusters from Real Data

In this subsection, we discuss the biological meaning of the pClusters found from real microarray data. In the previous experiment on the real yeast microarray data, when the parameters were set to $\delta = 0$, $NR = 35$, and $NC = 6$, we could find two pClusters, as shown in Figure 30. Then, we utilized a *GO* (Gene Ontology) annotation searching tool, GO Term Finder (www.yeastgenome.org/cgi-bin/GO/goTermFinder.pl), to verify the corresponding biological meaning. The GO project provides a controlled vocabulary to describe gene and gene product attributes in any organism, and is a collaborative effort to address the need for consistent descriptions of gene products in different databases (cited from www.geneontology.org). By using the GO Term Finder, we could find significant shared GO terms for describing genes

Samples	Column No. (0-16): 10 11 13 14 15 16
Genes	YAR061W YBR032W YBR298C YCR063W YCR081W YCR107W YDL113C YDL206W YDL210W YDL239C YDL247W YDR118W YDR259C YEL004W YEL006W YEL052W YER060W YFL054C YGL006W YGL229C YGL250W YGR065C YGR088W YGR122W YGR197C YIL097W YIL120W YIR007W YJL083W YJL147C YKL222C YKR076W YML066C YMR034C YNL191W YOL023W YOR173W

(a)

Samples	Column No. (0-16): 4 5 6 7 8 12
Genes	YAL064W YBL095W YBR295W YCR063W YDL221W YDL242W YDR544C YEL052W YER075C YFL056C YGL090W YGL131C YGR153W YGR212W YHR014W YHR015W YHR171W YIL171W YJR120W YKR104W YLL013C YLR176C YLR374C YML066C YMR133W YMR306W YNL092W YNR068C YNR073C YOL114C YOL117W YOL163W YOR040W YOR173W YPR015C YPR061C

(b)

Fig. 30. The pClusters found from the real yeast microarray data: (a) the first pCluster consisting of 37 genes and 6 samples; (b) the second pCluster consisting of 36 genes and 6 samples.

within the found pClusters. Table 7 shows the searching result of GO Terms for the pCluster shown in Figure 30-(a). We list only those significant shared terms with *p-values* less than 0.01, where a p-value is a score of significance. The closer the p-value is to zero, the more significant the particular GO term associated with the group of genes is (cited from www.yeastgenome.org). From Table 7, we could observe that genes within the same pCluster significantly share the same GO terms. This means that these genes may jointly participate in some activities. Although we could not find significant shared GO terms for the pCluster in Figure 30-(b) (due to many genes being annotated as "unknown cellular component/molecular function/biological process"), there may exist some interesting relationships among these genes waiting for biologists to discover them.

4.5 Efficiency of Pruning and Bounding Techniques

In this subsection, we experiment the efficiency of the proposed pruning and bounding techniques in the CE-Tree method. We use the synthetic data generated in the same way mentioned previously as the input data. The default values of parameters are $r = 7000$, $c = 50$, $nr = 50$, and $nc = 10$. We vary the value of k, *i.e.*, the number of embedded pClusters. Table 8 shows the simulation results of the reduction rate of the number of expanded nodes in the CE-Tree, where "Density of pClusters" means the ratio of the total size of all pClusters to the size of the entire microarray data. From this table, we could observe that as the density of pClusters increases, the efficiency of the proposed pruning and bounding techniques decreases. The reason is that the more potential pClusters there exist, the easier one node in the CE-Tree passes the pruning and bounding situations. However, since we do not generate object-pair MDSs, we could still find pClusters efficiently even if there exist a lot of pClusters in the microarray data, as described in Subsection 4.3.

Table 7

Terms from the function ontology for the genes in Fiugre 30-(a)

GO term	Genes annotated to the term	P-value
transporter activity	YBR298C, YDL210W, YDL247W, YEL004W, YEL006W, YER060W, YFL054C, YGL006W, YGR065C, YIL120W, YMR034C	0.0000635
transmembrane transporter activity	YBR298C, YDL210W, YDL247W, YEL004W, YER060W, YFL054C, YGL006W, YIL120W	0.00303
substrate-specific transporter activity	YBR298C, YDL210W, YDL247W, YEL004W, YER060W, YFL054C, YGL006W, YMR034C	0.00537
substrate-specific transmembrane transporter activity	YBR298C, YDL210W, YDL247W, YEL004W, YER060W, YFL054C, YGL006W	0.00876

5 Conclusion

Biclustering has proved of great value for finding the interesting patterns from the microarray expression data. In this paper, based on the pCluster model, we have proposed a new method, CE-Tree, to solve the problem of biclustering

Table 8

The reduction rate of the number of expanded nodes in the CE-Tree

k	Density of pClusters	The reduction rate
5	0.7%	99.9%
10	1.4%	89.2%
15	2.1%	63.6%
20	2.8%	35%
25	3.5%	10%

for DNA microarray data. The CE-Tree method could find all pClusters without generating gene-pair MDSs or performing complex duplicating processes. Moreover, we have made use of the properties of the LBGD expanding manner to develop three bounding techniques for the CE-Tree method. We have also developed the signature tables, which utilize the idea of the traditional hash join approach, to efficiently support the joining process in the CE-Tree. From the simulation results on synthetic and real microarray data, we have shown that the CE-Tree method is more efficient than those previous methods which need to generate gene-pair MDSs, since the total execution time of the CE-Tree method is less than the time for generating only the gene-pair MDSs. On the other hand, although the MicroCluster method does not generate gene-pair MDSs, we also have shown that the CE-Tree method outperforms the MicroCluster method in terms of the execution time. Furthermore, we have experimented the efficiency of the proposed pruning and bounding techniques.

Acknowledgments

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-95-2221-E-110-079-MY2. The authors also like to thank “Aim for Top University Plan” project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

References

Agrawal, R., Srikant, R., 1994. Fast Algorithms for Mining Association Rules. Proc. of the 20th Int. Conf. on Very Large Data Bases, 487–499.

Aguilar-Ruiz, J. S., Oct. 2005. Shifting and Scaling Patterns from Gene Expression Data. Bioinformatics 21 (20), 3840–3845.

Brunet, J. P., Tamayo, P., Golub, T. R., Mesirov, J. P., 2004. Metagenes and Molecular Pattern Discovery Using Matrix Factorization. In: Proc. of the National Academy of Science. pp. 4164–4169.

Bull, J. M., Smith, L. A., Pottage, L., Freeman, R., 2001. Benchmarking Java Against C and Fortran for Scientific Applications. In: Proc. of ACM-ISCOPE Conf. on Java Grande. pp. 97–105.

Chen, M. S., Han, J., Yu, P. S., Dec. 1996. Data Mining: An Overview from Database Perspective. IEEE Trans. on Knowledge and Data Engineering 8 (6), 866–883.

Cheng, Y., Church, G. M., 2000. Biclustering of Expression Data. In: Proc. of the 8th Int. Conf. on Intelligent Systems for Molecular Biology. pp. 93–103.

Chi, Y., Wang, H., Yu, P. S., Muntz, R. R., 2004. Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window. In: Proc. of the 4th IEEE Int. Conf. on Data Mining. pp. 59–66.

DeWitt, D. J., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., June 1984. Implementation Techniques for Main Memory Database Systems. ACM SIGMOD Record 14 (2), 1–8.

Khan, J., Wei, J. S., Ringner, M., Saal, L. H., Ladanyi, M., Westermann, F., Berthold, F., Schwab, M., Antonescu, C. R., Peterson, C., Meltzer, P. S., June 2001. Classification and Diagnostic Prediction of Cancers Using Expression Profiling and Artificial Neural Networks. Nature Medicine 7 (6), 673–679.

Kitsuregawa, M., Tanaka, H., Moto-Oka, T., 1983. Application of Hash to Data Base Machine and Its Architecture. New Generation Computing 1 (1), 63–74.

Madeira, S. C., Oliveira, A. L., Jan.–March 2004. Biclustering Algorithms for Biological Data Analysis: A Survey. IEEE/ACM Trans. on Computational Biology and Bioinformatics 1 (1), 24–45.

Merz, P., Nov. 2003. Analysis of Gene Expression Profiles: An Application of Memetic Algorithms to the Minimum Sum-of-Squares Clustering Problem. Biosystems 72 (1–2), 99–109.

Pei, J., Zhang, X., Cho, M., Wang, H., Yu, P. S., 2003. Maple: A Fast Algorithm for Maximal Pattern-based Clustering. In: Proc. of the 3rd IEEE Int. Conf. on Data Mining. pp. 259–266.

Tan, M. P., Smith, E. N., Broach, J. R., Floudas, C. A., June 2008. Microarray Data Mining: A Novel Optimization-Based Approach to Uncover Biologically Coherent Structures. BMC Bioinformatics 9 (268), 1–21.

Tavazoie, S., Hughes, J. D., Campbell, M. J., Cho, R. J., Church, G. M., July 1999. Systematic Determination of Genetic Network Architecture. Nature Genetics 22 (3), 281–285.

Wang, H., Wang, W., Yang, J., Yu, P. S., 2002. Clustering by Pattern Similarity in Large Data Sets. In: Proc. of ACM SIGMOD Int. Conf. on Management of Data. pp. 394–405.

Yang, J., Wang, H., Wang, W., Yu, P. S., Oct. 2005. An Improved Biclustering Method for Analyzing Gene Expression Profiles. Int. Journal on Artificial Intelligence Tools 14 (5), 771–789.

Yang, J., Wang, W., Wang, H., Yu, P. S., 2002. δ-Clusters: Capturing Subspace Correlation in a Large Data Set. In: Proc. of the 18th Int. Conf. on Data

Eng. . pp. 517–528.

Yoon, S., Nardini, C., Benini, L., Micheli, G. D., Oct.–Dec. 2005. Discovering Coherent Biclusters from Gene Expression Data Using Zero-Suppressed Binary Decision Diagrams. IEEE/ACM Trans. on Computational Biology and Bioinformatics 2 (4), 339–354.

Zhao, L., Zaki, M. J., Nov./Dec. 2005. MicroCluster: Efficient Deterministic Biclustering of Microarray Data. IEEE Intelligent Systems 20 (6), 40–49.