Mining Frequent Patterns from Very High Dimensional Data: A Top-Down Row Enumeration Approach*

Hongyan Liu[1] Jiawei Han[2] Dong Xin[2] Zheng Shao[2]

[1]Department of Management Science and Engineering, Tsinghua University
hyliu@tsinghua.edu.cn
[2]Department of Computer Science, University of Illinois at Urbana-Champaign
{hanj, dongxin, zshao1}@uiuc.edu

Abstract

Data sets of very high dimensionality, such as microarray data, pose great challenges on efficient processing to most existing data mining algorithms. Recently, there comes a row-enumeration method that performs a bottom-up search of row combination space to find corresponding frequent patterns. Due to a limited number of rows in microarray data, this method is more efficient than column enumeration-based algorithms. However, the bottom-up search strategy cannot take an advantage of user-specified minimum support threshold to effectively prune search space, and therefore leads to long runtime and much memory overhead.

In this paper we propose a new search strategy, top-down mining, integrated with a novel row-enumeration tree, which makes full use of the pruning power of the minimum support threshold to cut down search space dramatically. Using this kind of searching strategy, we design an algorithm, TD-Close, to find a complete set of frequent closed patterns from very high dimensional data. Furthermore, an effective closeness-checking method is also developed that avoids scanning the dataset multiple times. Our performance study shows that the TD-Close algorithm outperforms substantially both Carpenter, a bottom-up searching algorithm, and FPclose, a column enumeration-based frequent closed pattern mining algorithm.

1 Introduction

With the development of bioinformatics, microarray technology produces many gene expression data sets, i.e., *microarray data*. Different from transactional data set, *microarray data* usually does not have so many rows (samples) but have a large number of columns (genes). This kind of very high dimensional data needs data mining techniques to discover interesting knowledge from it. For example, frequent pattern mining algorithm can be used to find co-regulated genes or gene groups [2, 14]. Association rules based on frequent patterns can be used to build gene networks [9]. Classification and clustering algorithms are also applied on *microarray data* [3, 4, 6]. Although there are many algorithms dealing with transactional data sets that usually have a small number of dimensions and a large number of tuples, there are few algorithms oriented to very high dimensional data sets with a small number of tuples. Taking frequent-pattern mining as an example, most of the existing algorithms [1, 10, 11, 12, 13] are column enumeration-based, which take column (item) combination space as search space. Due to the exponential number of column combinations, this method is usually not suitable for very high dimensional data.

Recently, a row enumeration-based method [5] is proposed to handle this kind of very high dimensional data. Based on this work, several algorithms have been developed to find frequent closed patterns or classification rules [5, 6, 7, 8]. As they search through the row enumeration space instead of column enumeration space, these algorithms are much faster than their counterparts in very high dimensional data. However, as this method exploits a bottom-up search strategy to check row combinations from the smallest to the largest, it cannot make full use of the minimum support threshold to prune search space. As a result, experiments show that it often cannot run to completion in a reasonable time for large *microarray data*, and it sometimes runs out of memory before completion. To solve these problems, we propose a new *top-down* search strategy for row enumeration-

*This work was supported in part by the National Natural Science Foundation of China under Grant No. 70471006 and 70321001, and by the U.S. National Science Foundation NSF IIS-02-09199 and IIS-03-08215.

based mining algorithm. To show its effectiveness, we design an algorithm, called *TD-Close*, to mine a complete set of frequent closed patterns and compare it with two bottom-up search algorithms, *Carpenter* [5], and *FPclose* [15]. Here are the main contributions of this paper:

(1) A top-down search method and a novel row-enumeration tree are proposed to take advantage of the pruning power of minimum support threshold. Our experiments and analysis show that this cuts down the search space dramatically. This is critical for mining high dimensional data, because the dataset is usually big, and without pruning the huge search space, one has to generate a very large set of candidate itemsets for checking.

(2) A new method, called *closeness-checking,* is developed to check efficiently and effectively whether a pattern is closed. Unlike other existing *closeness-checking* methods, it does not need to scan the mining data set, nor the result set, and is easy to integrate with the top-down search process. The correctness of this method is proved by both theoretic proof and experimental results.

(3) An algorithm using the above two methods is designed and implemented to discover a complete set of frequent closed patterns. Experimental results show that this algorithm is more efficient and uses less memory than bottom-up search styled algorithms, *Carpenter* and *FPclose*.

The remaining of the paper is organized as follows. In section 2, we present some preliminaries and the mining task. In section 3, we describe our top-down search strategy and compare it with the bottom-up search strategy. We present the new algorithm in section 4 and conduct experimental study in section 5. Finally, we give the related work in section 6 and conclude the study in section 7.

2 Preliminaries

Let T be a discretized data table (or data set), composed of a set of rows, $\mathcal{S} = \{r_1, r_2, \ldots, r_n\}$, where r_i ($i = 1, \ldots, n$) is called a row ID, or *rid* in short. Each row corresponds to a sample consisting of *k* discrete values or intervals, and $\mathcal{I}$ is the complete set of these values or intervals, $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$. For simplicity, we call each i_j an *item*. We call a set of *rids* S $\subseteq \mathcal{S}$ a *rowset*, and a rowset having *k rids* a *k-rowset*. Likewise, we call a set of items I $\subseteq \mathcal{I}$ an itemset. Hence, a table T is a triple ($\mathcal{S}$, $\mathcal{I}$, $\mathcal{R}$), where $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{I}$ is a relation. For a $r_i \in \mathcal{S}$, and a $i_j \in \mathcal{I}$, $(r_i, i_j) \in \mathcal{R}$ denotes that r_i contains i_j, or i_j is contained by r_i.

Let TT be the transposed table of T, in which each row corresponds to an item i_j and consists of a set of *rids* which contain i_j in T. For clarity, we call each row of TT a tuple. Table TT is a triple ($\mathcal{I}$, $\mathcal{S}$, $\mathcal{R}$), where $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{I}$ is a relation. For a *rid* $r_i \in \mathcal{S}$, and an item $i_j \in \mathcal{I}$, $(r_i, i_j) \in \mathcal{R}$ denotes that i_j contains r_i, or r_i is contained by i_j.

Example 2.1 (**Table and transposed table**) Table 2.1 shows an example table T with 4 attributes (columns): *A*, *B*, *C* and *D*. The corresponding transposed table TT is shown in Table 2.2. For simplicity, we use number *i* (*i = 1, 2, …, n*) instead of r_i to represent each *rid*. In order to describe our search strategy and mining algorithm clearly, we need to define an order of these rows. In this paper, we define the numerical order of *rids* as the order, i.e., a row *j* is greater than *k* if *j* > *k*.

Let minimum support (denoted *minsup*) be set to 2. All the tuples with the number of *rids* less than *minsup* is deleted from TT. Table TT shown in Table 2.2 is already pruned by *minsup*. This kind of pruning will be further explained in the following sections.

In this paper we aim to discover the set of the frequent closed patterns. Some concepts related to it are defined as follows.

Table 2.1 An example table T

r_i	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_1	c_2	d_2
3	a_1	b_1	c_1	d_2
4	a_2	b_1	c_2	d_2
5	a_2	b_2	c_2	d_3

Table 2.2 Transposed table TT of T

itemset	rowset
a_1	1, 2, 3
a_2	4, 5
b_1	1, 2, 3, 4
c_1	1, 3
c_2	2, 4, 5
d_2	2, 3, 4

Definition 2.1 (**Closure**) Given an itemset I $\subseteq \mathcal{I}$ and a rowset S $\subseteq \mathcal{S}$, we define

$$r(I) = \{ r_i \in \mathcal{S} \mid \forall i_j \in I, (r_i, i_j) \in \mathcal{R} \}$$

$$i(S) = \{ i_j \in \mathcal{I} \mid \forall r_i \in S, (r_i, i_j) \in \mathcal{R} \}$$

Based on these definitions, we define C(I) as the closure of an itemset I, and C(S) as the closure of a rowset S as follows:

$$C(I) = i(r(I))$$
$$C(S) = r(i(S))$$

Note that definition 2.1 is applicable to both table T and TT.

Definition 2.2 (**Closed itemset and closed rowset**) An itemset I is called a *closed itemset* iff I = C(I). Likewise, a rowset S is called a *closed rowset* iff S = C(S).

Definition 2.3 (**Frequent itemset and large rowset**) Given an absolute value of user-specified threshold *minsup*, an itemset I is called *frequent* if |r(I)| ≥ *minsup*, and a rowset S is called *large* if |S| ≥ *minsup*, where |r(I)| is called the *support* of itemset I and *minsup* is called the *minimum support threshold*. |S| is called the *size* of rowset S, and *minsup* is called *minimum size threshold* accordingly. Further, an itemset I is called *frequent closed itemset* if it is both closed and frequent. Likewise, a rowset S is called *large closed rowset* if it is both closed and large.

Example 2.2 (Closed itemset and closed rowset) In table 2.1, for an itemset $\{b_1, c_2\}$, $r(\{b_1, c_2\}) = \{2, 4\}$, and $i(\{2, 4\}) = \{b_1, c_2, d_2\}$, so $C(\{b_1, c_2\}) = \{b_1, c_2, d_2\}$. Therefore, $\{b_1, c_2\}$ is not a closed itemset. If *minsup* = 2, it is a frequent itemset. In table 2.2, for a rowset {1, 2}, $i(\{1, 2\}) = \{a_1, b_1\}$ and $r(\{a_1, b_1\}) = \{1, 2, 3\}$, then C(S) = {1, 2, 3}. So rowset {1, 2} is not a closed rowset, but apparently {1, 2, 3} is.

Mining task: Originally, we want to find all of the frequent closed itemsets which satisfy the minimum support threshold *minsup* from table T. After transposing T to transposed table TT, the mining task becomes finding all of the large closed rowsets which satisfy minimum size threshold *minsup* from table TT.

3 Top-down Search Strategy

Before giving our top-down search strategy, we will first look at what is bottom-up search strategy used by the previous mining algorithms [5, 6, 7, 8]. For simplicity, we will use *Carpenter* as a representative for this group of algorithms since they use the same kind of search strategy.

3.1 Bottom-up Search Strategy

Figure 3.1 shows a row enumeration tree that uses the bottom-up search strategy. By bottom-up we mean that along every search path, we search the row enumeration space from small rowsets to large ones. For example, first single rows, then *2-rowsets*, …, and finally *n-rowsets*. Both depth-first and breadth-first search of this tree belong to this search strategy.

In Figure 3.1, each node represents a rowset. Our mining task is to discover all of the large closed rowsets. So the main constraint for mining is the size of rowset. Since it is monotonic in terms of bottom up search order, it is hard to prune the row enumeration search space early. For example, suppose *minsup* is set to 3, although obviously all of the nodes in the first two levels from the root cannot satisfy this constraint, these nodes still need to be checked [5, 6, 7, 8]. As a result, as the *minsup* increases, the time needed to complete the mining process cannot decrease rapidly. This limits the application of this kind of algorithms to real situations.

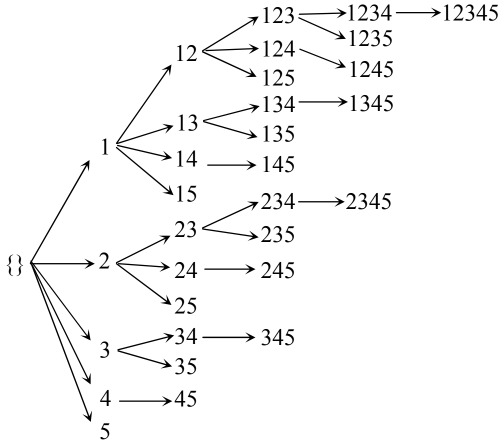

Figure 3.1 Bottom-up row enumeration tree

In addition, the memory cost for this kind of bottom-up search is also big. Take *Carpenter* as an example. Similar to several other algorithms [6, 7, 8], *Carpenter* uses a pointer list to point to each tuple belonging to an *x-conditional* transposed table. For a table with *n* rows, the maximum number of different levels of pointer lists needed to remain in memory is *n*, although among which the first (*minsup* – 1) levels of pointer lists will not contribute to the final result.

These observations motivate the proposal of our method.

3.2 Top-down Search Strategy

Almost all of the frequent pattern mining algorithms dealing with the data set without transposition use the anti-monotonicity property of *minsup* to speed up the mining process. For transposed data set, the *minsup* constraint maps to the size of rowset. In order to stop further search when *minsup* is not satisfied, we propose to exploit a top-down search strategy instead of the bottom-up one. To do so, we design a row enumeration tree following this strategy, which is shown in Figure 3.2. Contrary to the bottom-up search strategy, the top-down searching strategy means that along each search path the rowsets are checked from large to small ones.

In Figure 3.2, each node represents a rowset. We define the level of root node as 0, and then the highest level for a data set with *n* rows is (*n – 1*).

It is easy to see from the figure 3.2 that for a table with *n* rows, if the user specified minimum support threshold is *minsup*, we do not need to search all of rowsets which are in levels greater than (*n – minsup*) in the row enumeration tree. For example, suppose *minsup* = 3, for data set shown in Table 2.1, we can stop further search at level 2, because rowsets represented by nodes at level 3 and 4 will not contribute to the set of frequent patterns at all.

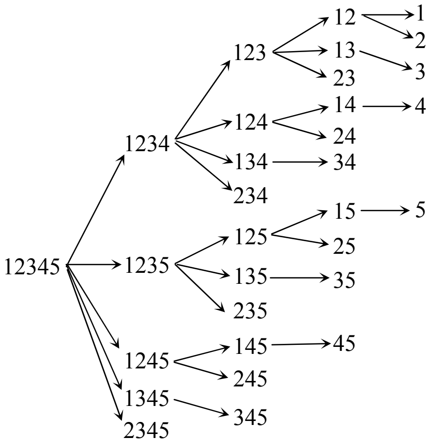

Figure 3.2 Top-down row enumeration tree

With this row enumeration tree, we can also mine the transposed table TT by divide-and-conquer method. Each node of the tree in Figure 3.2 corresponds to a sub-table. For example, the root represents the whole table TT, and then it can be divided into 5 sub-tables: table without *rid* 5, table with 5 but without 4, table with 45 but without 3, table with 345 but without 2, and table with 2345 but without 1. Here 2345 represents the set of rows {2, 3, 4, 5}, and same holds for 45 and 345. Each of these tables can be further divided by the same rule. We call each of these sub-tables *x-excluded transposed table*, where *x* is a rowset which is excluded in the table. Tables corresponding to a parent node and a child node are called *parent table* and *child table* respectively. Following is the definition of *x-excluded transposed table*.

Definition 3.1 (*x-excluded transposed table*) Given a rowset $x = \{r_{i1}, r_{i2}, \ldots, r_{ik}\}$ with an order such that $r_{i1} > r_{i2} > \ldots > r_{ik}$, a minimum support threshold *minsup* and its parent table $TT|_p$, an *x-excluded transposed table* $TT|_x$ is a table in which each tuple contains *rids* less than any of *rids* in *x*, and at the same time contains all of the rids greater than any of *rids* in *x*. Rowset *x* is called an *excluded rowset*.

Example 3.1 (*x-excluded transposed table*) For transposed table TT shown in Table 2.2, two of its *x-excluded tranposed tables* are shown in Tables 3.1 and 3.2 respectively, assuming *minsup* = 2.

Table 3.1 shows an *x-excluded tranposed table* $TT|_{54}$, where *x* = {5, 4}. In this table, each tuple only contains *rids* which are less then 4, and contains at least two such *rids* as *minsup* is 2. Since the largest *rid* in the original data set is 5, it is not necessary for each tuple to contain some other *rids*. Procedures to get this table are shown in Example 3.2.

Table 3.2 is an *x-excluded transposed table* $TT|_4$, where *x* = {4}. Its parent table is the table shown in Table 2.2. Each tuple in $TT|_4$ must contain *rid* 5 as it is greater than 4, and in the meantime must contain at least one rid less than 4 as *minsup* is set to 2. As a result, in Table 2.2 only those tuples containing *rid* 5 can be a candidate tuple of $TT|_4$. Therefore, only tuples a_2 and c_2 satisfy this condition. But tuple a_2 does not satisfy *minsup* after excluding *rid* 4, so only one tuple left in $TT|_4$. Note, although the current size of tuple c_2 in $TT|_4$ is 1, its actual size is 2 since it contains *rid* 5 which is not listed explicitly in the table.

Table 3.1 $TT|_{54}$

itemset	rowset
a_1	1, 2, 3
b_1	1, 2, 3
c_1	1, 3
d_2	2, 3

Table 3.2 $TT|_4$

itemset	rowset
c_2	2

The *x-excluded transposed table* can be obtained by the following steps.

(1) Extract from TT or its direct parent table $TT|_p$ each tuple containing all *rids* greater than r_{i1}.

(2) For each tuple obtained in the first step, keep only *rids* less than r_{ik}.

(3) Get rid of tuples containing less than (*minsup* – j) number of *rids*, where j is the number of rids greater than r_{i1} in S.

The reason of the operation in step 3 will be given in section 4. Note that the original transposed table corresponds to TT$|_{\phi}$, where ϕ is an empty rowset.

Figure 3.3 shows the corresponding excluded row enumeration tree for the row enumeration tree shown in Figure 3.2. This tree shows the parent-child relationship between the excluded rowsets.

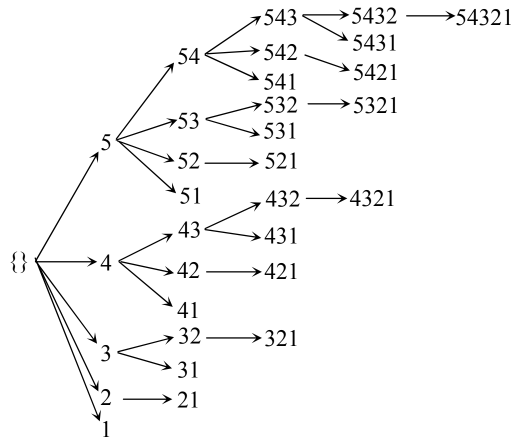

Figure 3.3 Excluded row enumeration tree

Example 3.2 (Procedure to get *x-excluded transposed table*) Take TT$|_{54}$ as an example, here is the step to get it. Table TT$|_5$ shown in Table 3.3 is its parent table.

Table 3.3 TT$|_5$

itemset	rowset
a_1	1, 2, 3
b_1	1, 2, 3, 4
c_1	1, 3
c_2	2, 4
d_2	2, 3, 4

(1) Each tuple in table TT$|_5$ is a candidate tuple of TT$|_{54}$ as there is no *rid* greater than 5 for the original data set.

(2) After excluding rids not less than 4, the table is shown in Table 3.4.

(3) Since tuple c_2 only contains one *rid*, it does not satisfy *minsup* and is thus pruned from TT$|_{54.}$ Then we get the final TT$|_{54}$ shown in Table 3.1.

Table 3.4 TT$|_{54}$ without pruning

itemset	rowset
a_1	1, 2, 3
b_1	1, 2, 3
c_1	1, 3
c_2	2
d_2	2, 3

From definition 3.1 and the above procedure to get *x-excluded transposed table* we can see that the size of the excluded table will become smaller and smaller due to the *minsup* threshold, so the search space will shrink rapidly.

As for the memory cost, in order to compare with *Carpenter*, we also use pointer list to simulate the *x-excluded transposed table*. What is different is that this pointer list keeps track of rowsets from the end of each tuple of TT, and we also split it according to the current rid. We will not discuss the detail of implementation due to space limitation. However, what is clear is that when we stop search at level (*n* – *minsup*), we do not need to spend more memory for all of the excluded transposed tables corresponding to nodes at levels greater than (*n* – *minsup*), and we can release the memory used for nodes along the current search path. Therefore, comparing to *Carpenter*, it is more memory saving. This is also demonstrated in our experimental study, as *Carpenter* often runs out of memory before completion.

4 Algorithm

To mining frequent closed itemsets from high dimensional data using the top-down search strategy, we design an algorithm, *TD-Close,* and compare it with the corresponding bottom-up based algorithm *Carpenter*. In this section, we first present our new *closeness-checking* method and then describe the new algorithm.

4.1 Closeness-Checking Method

To avoid generating all the frequent itemsets during the mining process, it is important to perform the closeness checking as early as possible during mining. Thus an efficient *closeness-checking* method has been developed, based on the following lemmas.

Lemma 4.1 Given an itemset $I \subseteq \mathcal{I}$ and a rowset $S \subseteq \mathcal{S}$, the following two equations hold:

$$r(I) = r(i(r(I))) \qquad (1)$$

$$i(S) = i(r(i(S))) \qquad (2)$$

Proof. Since r(I) is a set of rows that share a given set of items, and i(S) is a set of items that is common to a set of rows, according to this semantics and definition 2.1, these two equations are obviously correct.

Lemma 4.2 In transposed table TT, a rowset $S \subseteq \mathcal{S}$ is closed iff it can be represented by an intersection of a set of tuples, that is:

$$\exists I \subseteq \mathit{I}, \text{ s.t. } S = r(I) = \bigcap_j r(\{i_j\})$$

$$\text{where } i_j \in I, \text{ and } I = \{i_1, i_2, \ldots, i_l\}$$

Proof. First, we prove that if S is closed then s = r(I) holds. If S is closed, according to definition 2.2, S = C(S) = r(i(S)), we can always set I = i(S), so S = r(I) holds. Now we need to prove that if s = r(I) holds then S is closed. If s = r(I) holds for some $I \subseteq \mathcal{I}$, according to definition 2.1, C(S) = r(i(S)) = r(i(r(I))) holds. Then based on Lemma 4.1 we have r(i(r(I))) = r(I) = S, so C(S) = S holds. Therefore S is closed.

Lemma 4.3 Given a rowset $S \subseteq \mathcal{S}$, in transposed table TT, for every tuple i_j containing S, which means $i_j \in i(S)$, if $S \neq \bigcap_j r(\{i_j\})$, where $i_j \in i(S)$, then S is not closed.

Proof. For tuple $i_j \in i(S)$, $\bigcap_j r(\{i_j\}) \supseteq S$ apparently holds. If $S \neq \bigcap_j r(\{i_j\})$ holds, then $\bigcap_j r(\{i_j\}) \subset S$ holds, which means that there exists at least another one item, say y, such that $S \cup y = \bigcap_j r(\{i_j\})$. So $S \neq r(I)$, that is $S \neq r(i(S))$. Therefore S is not closed.

Lemmas 4.2 and 4.3 are the basis of our *closeness-checking* method. In order to speed up this kind of checking, we add some additional information for each *x-excluded transposed table*. The third column of the table shown in Tables 4.1 or 4.3 is just it. The so-called *skip-rowset* is a set of *rids* which keeps track of of the *rids* that are excluded from the same tuple of all of its parent tables. When two tuples in an *x-excluded transposed table* have the same rowset, they will be merged to one tuple, and the intersection of corresponding two *skip-rowsets* will become the current *skip-rowset*.

Example 4.1 (*skip-rowset* and merge of *x-excluded transposed table*) In example 3.2, when we got $TT|_{54}$ from its parent $TT|_5$, we excluded *rid* 4 from tuple b_1 and d_2 respectively. The *skip-rowset* of these two tuples in $TT|_5$ should be empty as they do not contain *rid* 5 in $TT|_\phi$. Therefore, the *skip-rowset* of these two tuples in $TT|_{54}$ is 4. Table 4.1 shows $TT|_{54}$ with this kind of *skip-rowsets*.

In Table 4.1, the first 2 tuples have the same rowset {1, 2, 3}. After merging these two tuples, it becomes Table 4.2. The *skip-rowset* of this rowset becomes empty because the intersection of an empty set and any other set is still empty. If the intersection result is empty, it means that currently this rowset is the result of intersection of two tuples. When it is time to output a rowset, this *skip-rowset* will be checked. If it is empty, then it must be a closed rowset.

Table 4.1 $TT|_{54}$ with *skip-rowset*

itemset	rowset	*skip-rowset*
a_1	1, 2, 3	
b_1	1, 2, 3	4
c_1	1, 3	
d_2	2, 3	4

Table 4.2 TT|54 after merge

itemset	rowset	*skip-rowset*
a_1 b_1	1, 2, 3	
c_1	1, 3	
d_2	2, 3	4

Table 4.3 TT|4 with skip-rowset

itemset	rowset	*skip-rowset*
c_2	2	4

4.2 The *TD-Close* Algorithm

Based on the top-down search strategy and the *closeness-checking* method, we design an algorithm, called ***TD-Close,*** to mine all of the frequent closed patterns from table T.

Figure 4.1 shows the main steps of the algorithm. It begins with the transposition operation that transforms table T to the transposed table TT. Then, after the initialization of the set of frequent closed patterns FCP to empty set and *excludedSize* to 0, the subroutine *TopDownMine* is called to deal with each *x-excluded transposed table* and find all of the frequent closed itemsets. The General processing order of rowsets is equivalent to the depth-first search of the row enumeration tree shown in Figure 3.2.

Subroutine *TopDownMine* takes each *x-excluded transposed table* and another two variables, *cMinsup excludedSize*, as parameters and checks each candidate rowset of the *x-excluded transposed table* to see if it is closed. Candidate rowsets are those large rowsets that occur at least once in table TT. Parameter *cMinsup* is a dynamically changing minimum support threshold as indicated in step 5, and *excludedSize* is the size of rowset *x*. There are five main steps in this subroutine, which will be explained one by one as follows.

```
Algorithm TD-Close
Input: Table T, and minimum support threshold, minsup
Output: A complete set of frequent closed patterns, FCP
Method:
    1. Transform T into transposed table TT
    2. Initialize $FCP = \Phi$ and $excludedSize = 0$
    3. Call TopDownMine($TT|_{\Phi}$, minsup, excludedSize)

Subroutine TopDownMine($TT|_x$, cMinsup, excludedSize)
Method:
    1. Pruning 1: if excludedSize >= (n–minsup) return;
    2. Pruning 2: If the size of $TT|_x$ is 1, output the
       corresponding itemset if the rowset is closed, and
       then return.
    3. Pruning 3: Derive $TT|_{x \cup y}$ and $TT|_x$’, where
       y is the largest rid among rids in tuples of $TT|_x$,
       $TT|_x$’ = {tuple $t_i$ | $t_i \in TT|_x$ and $t_i$ contains y},
       $TT|_{x \cup y}$ = {tuple $t_i$ | $t_i \in TT|_x$ and if $t_i$ contains y,
       size of $t_i$ must be greater than cMinsup}
       Note, we delete y from both $TT|_{x \cup y}$ and $TT|_x$’ .
    4. Output: Add to FCI itemset corresponding to
       each rowset in $TT|_{x \cup y}$ with the largest size k and
       ending with rid k.
    5. Recursive call:
       TopDownMine($TT|_{x \cup y}$, cMinsup, exclued Size+1)
       TopDownMine($TT|_x$’, cMinsup–1, excluedSize)
```

Figure 4.1 Algorithm *TD-Close*

In step 1, we apply pruning strategy 1 to stop processing current excluded transposed table.

Pruning strategy 1: If *excludedSize* is equal to or greater than (*n–minsup*), then there is no need to do any further recursive call of *TopDownMine*. *ExcludedSize* is the number of *rids* excluded from current table. If it is not less than (*n–minsup*), the size of each rowset in current transposed table must be less than *minsup*, so these rowsets are impossible to become large.

In step 2, we apply pruning strategy 2 to stop further recursive calls.

Pruning strategy 2: If an *x-excluded transposed table* contains only one tuple, it is not necessary to do further recursive call to deal with its child transposed tables.

The reason for this pruning strategy is apparent. Suppose the rowset corresponds to this tuple is S. From the itemset point of view, any child transposed table of this current table will not produce any different itemsets from the one corresponding to rowset S. From the rowset point of view, each rowset S_i corresponding to each child transposed table of S is a subset of S, and S_i cannot be closed because $r(i(S_i)) \supseteq S$ holds, and therefore $S_i \neq r(i(S_i))$ holds.

Of course, before return according to pruning strategy 2, the current rowset S might be a closed rowset, so if the *skip-rowset* is empty, we need to output it first.

Example 4.2 (pruning strategy 2) For table $TT|_4$ shown in Table 4.3, there is only one tuple in this table. After we check this tuple to see if it is closed (it is not closed apparently as its *skip-rowset* is not empty), we do not need to produce any child excluded transposed table from it. That is, according to excluded row enumeration tree shown in Figure 3.3, all of the child nodes of node {4} are pruned. This is because all of the subsets of the current rowset cannot be closed anymore since it is already contained by a larger rowset.

Step 3 is performed to derive from $TT|_x$ two child excluded transposed tables: $TT|_{x \cup y}$ and $TT|_x$’, where *y* is the largest *rid* among all *rids* in tuples of $TT|_x$. These two tables correspond to a partition of current table $TT|_x$. $TT|_{x \cup y}$ is the sub-table without *y*, and $TT|_x$’ is the sub-table with every tuple containing *y*. Since every rowset that will be derived from $TT|_x$’ must contain *y*, we delete *y* from $TT|_x$’ and at the same time decrease *cMinsup* by 1. Pruning strategy 3 is applied to shrink table $TT|_{x \cup y}$.

Pruning strategy 3: Each tuple *t* containing *rid y* in $TT|_x$ will be deleted from $TT|_{x \cup y}$ if size of *t* (that is the number of *rids t* contains) equals *cMinsup*.

Example 4.3 (pruning strategy 3) Suppose currently we have finished dealing with table $TT|_{54}$ which is shown in Table 4.2, and we need to create $TT|_{543}$ with *cMinsup* being 2. Then, according to pruning strategy 2, tuples c_1 and d_2 will be pruned from $TT|_{543,}$ because after excluding *rid* 3 from these two tuples, their size will become less than *cMinsup,* although currently they satisfy the *minsup* threshold. As a result, there is only one tuple $\{a_1b_1\}$ left in $TT|_{543,}$ as shown in Table 4.4.

We can get these two tables, TT$|_{x\cup y}$ and TT$|_x$’, by steps as follows. First, for each tuple *t* in TT$|_x$ containing *rid y*, delete *y* from it, and copy it to TT$|_x$’. And then check if the size of *t* is less than *cMinsup*. If not, keep it and in the meantime put *y* in the *skip-rowset*, otherwise get rid of it from TT$|_x$. Finally, TT$|_x$ becomes TT$|_{x\cup y}$, and at the same time TT$|_x$’ is obtained.

Step 4 is the major output step. If there is a tuple with empty *skip-rowset* and the largest size, say *k*, and containing *rid k* in TT$|_{x\cup y}$, then output it. Since this tuple has the largest size among all of the tuples in TT$|_{x\cup y}$, there is no other tuple that will contribute to it. Therefore, it can be output. For example, in table TT$|_{54}$ shown in Table 4.2, itemset a_1b_1 corresponding to rowset {1,2,3} can be output as its size is 3 which is larger than the size of the other two tuples, and it also contains the largest *rid* 3.

Step 5 conducts two recursive calls to deal with TT$|_{x\cup y}$ and TT$|_x$’ respectively.

5 Experimental Study

In this section, we will study the performance of the algorithm ***TD-Close***. Experiments for both synthetic data sets and real *microarray data* sets were conducted. *Carpenter* has already shown its much better performance than those column enumeration based algorithms such as CLOSET [10] and CHARM [11], so we only compare our algorithm with *Carpenter* and another column enumeration-based algorithm *FPclose*, which won the FIMI’03 best implementation award [15, 16]. All experiments were performed on a PC with a Pentium-4 1.5 Ghz CPU, 1GB RAM, and 30GB hard disk. All of the runtimes plotted in the figures include both computation time and I/O time.

For algorithm *FPclose*, we downloaded the source of the implementation from the FIMI repository [17].

For algorithm *Carpenter*, we implement it to our best knowledge according to paper [5] and its subsequent papers [6, 7, 8], and we also improve it by using a faster *closeness-checking* method, backward pruning, which is used in several other algorithms dealing with *microarray data* [6, 8]. The original *closeness-checking* method is that before outputting each itemset found currently, we must check if it is already found before. If not, output it. Otherwise, discard it. This method is slower than the backward pruning method, because itemset from very high dimensional data set usually contains large number of items, and it is not in specific order, so comparison with a large number of large itemsets takes long time. This is also the reason why some algorithms proposed after *Carpenter* use backward checking method. However, when dealing with every rowset, *Carpenter* with backward pruning strategy still needs to scan the corresponding conditional transposed table to find the largest common rowset, and also needs to scan the original transposed table to do backward pruning, so it needs lots of scan of the transposed table. In addition, to make the comparison fair, we just use a flat table to represent the transposed table TT instead of other data structure such as *FP-tree* that may speed up search to some extent.

5.1 Synthetic Data Sets

In order to test the performance of our top-down strategy based algorithm *TD-Close* with respect to several aspects, we use synthetic data set first. Figures 5.1 to 5.8 illustrate the change of running time as *minsup* decreases for data sets with different dimensions, tuples, and cardinalities. We use D#T#C# to represent specific dataset, where D# stands for dimension, the number of attributes of each data set, T# for number of tuples, and C# for cardinality, the number of values per dimension (or attribute). All data are generated randomly. In these experiments, D# ranges from 4000 to 10000, T# varies from 100, 150 to 200, and C# varies from 8, 10 to 12.

To test the performance of three algorithms with respect to the number of dimensions, we created 5 data sets with 4000, 6000, 8000 and 10000 dimensions respectively. Figures 5.1 to 5.4 show the effect of changing dimensionality on the runtime of these three algorithms.

We can see from Figure 5.1 that as *minsup* decreases, runtime of these three algorithms increases, and *TD-Close* is the fastest among these algorithms. Apparently, the increase speed of algorithm *TD-Close* and *Carpenter* is not fast, while when *minsup* reaches 10, the runtime of *FPclose* increases dramatically. This is because when *minsup* reaches 10, the number of frequent one items increases dramatically. Since *FPclose* is a column enumeration-based algorithm, the increase of the number of frequent items will lead to the explosion of the number of frequent itemsets which are needed to be checked one by one. On the other hand, row enumeration-based algorithms, such as *TD-Close* and *Carpenter*, search the row combination space. The number of rows influences the runtime of these algorithms much more than the number of frequent items does. The reason that *TD-Close* uses less time than *Carpenter* is that *TD-Close* can prune the search space much more than *Carpenter*, and can stop search much earlier than *Carpenter*.

Figures 5.2 to 5.4 also indicate the same trend as shown in Figure 5.1. What is different is that we cannot get all runtimes for *Carpenter* and *FPclose* for some datasets. For example, for dataset D6000T100C10, we cannot get the runtime for *Carpenter* when *minsup* is less than 13, and for *FPclose* when *minsup* is less than 11, which is either because it can not run to the end in reasonable time (in several hours) or due to memory error occurred during running. Same situations also happen in some of the following figures.

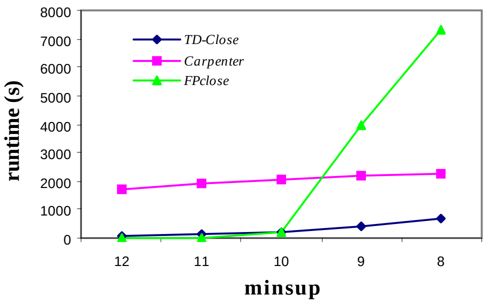

Figure 5.1 Runtime for D4000T100C10

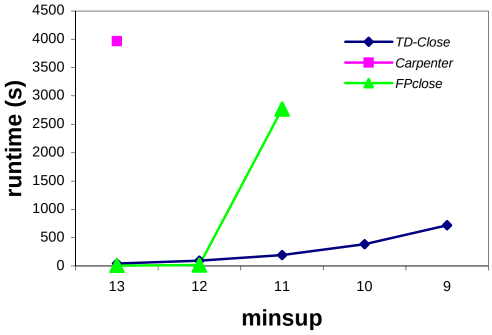

Figure 5.2 Runtime for D6000T100C10

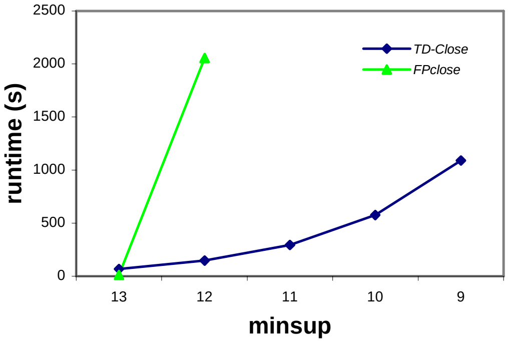

Figure 5.3 Runtime for D8000T100C10

From Figures 5.1 to 5.4, one can clearly see that all of these algorithms need more time for data sets with more dimensions for the same *minsup* value, and this is due to the rise of the number of itemsets as the number of dimensions increases.

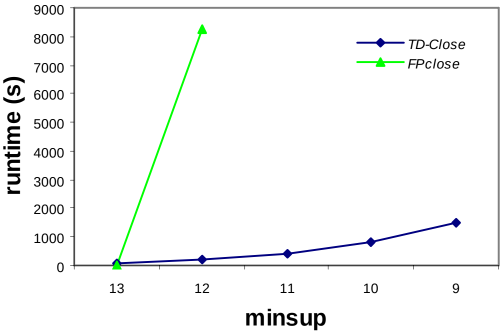

Figure 5.4 Runtime for D10000T100C10

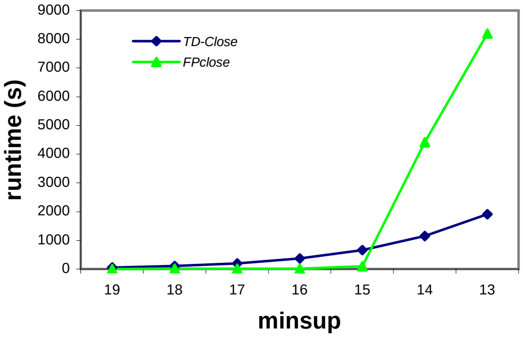

Figure 5.5 Runtime for D4000T150C10

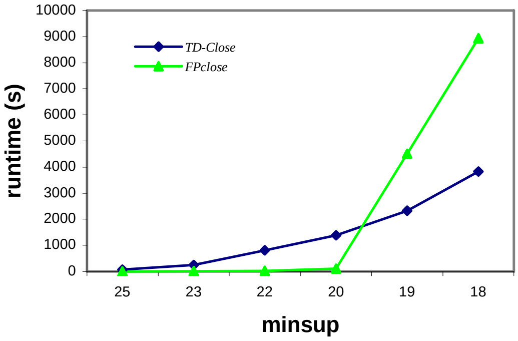

Figure 5.6 Runtime for D4000T200C10

To test the runtime of these algorithms with respect to the number of tuples, two more data sets are produced. One contains 150 tuples and the other 200 tuples, while dimension is 4000 and cardinality is 10. Figures 5.5 and 5.6 show the experimental results. For these two data sets, *Carpenter* cannot run to completion due to too long time needed. As for *TD-Close* and *FPclose*, we can see that for some relatively high values of *minsup*, *FPclose* runs a little faster than *TD-Close*. However, when *minsup* becomes relatively small, *TD-Close* runs much faster than *FPclose*. The reason is the same as explained above. That is, when *minsup* is high, *FPclose* can cut the itemset space to very small by *minsup* threshold so the search time is limited. But once the number of frequent items becomes large, the search space increases exponentially, while the number of rows remains relative stable.

In the above two groups of experiments, the cardinality of each data set is set to 10, which means each dimension of each dataset has 10 distinct values. To test the performance regarding different cardinalities, two other datasets are created, which correspond to 12 distinct values and 8 distinct values respectively. The number of dimension is 4000 and the number of tuples is 100. Experimental results are shown in Figures 5.7 and 5.8.

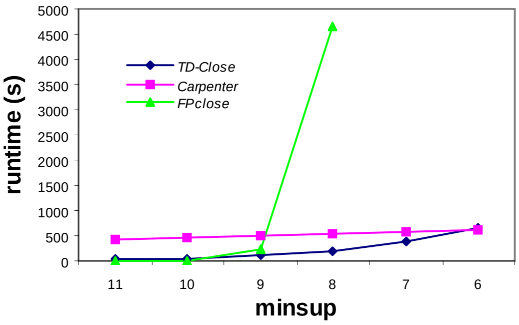

Figure 5.7 Runtime for D4000T100C12

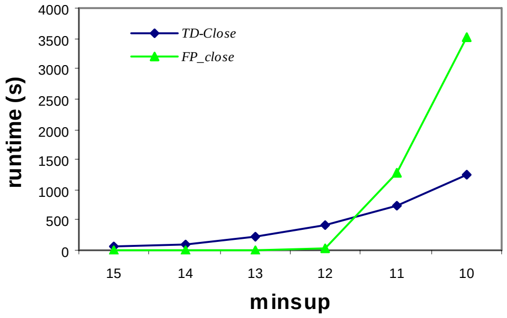

Figure 5.8 Runtime for D4000T100C8

Figure 5.7 shows us that when *minsup* becomes small, for example, less than 9, both *TD-Close* and *Carpenter* spend less time than *FPclose*. When *minsup* reaches 6, the runtime of *TD-Close* and *Carpenter* become almost the same. This is because when *minsup* becomes very low, the search space can be pruned by *minsup* becomes small, so the time saved by this kind of pruning in algorithm *TD-Close* becomes less.

Comparing with data set D4000T100C12, data set D4000T100C8 is relatively denser. With the same *minsup* threshold, the latter data set will produce more frequent patterns. Figure 5.8 shows almost the same features of *TD-Close* and *FPclose*. But obviously, it took them much more time than the former data set for the same *minsup* threshold. That is also the reason why we cannot get the exact runtime of *Carpenter* on it.

From these experimental results, one may see that for some relatively high *minsup*, *FPclose* is a little bit faster than *TD-Close*. But in that case, the runtime for both TD-close and FPclose is usually less than one minute, or only several minutes. So, the difference is not that significant. However, when *minsup* becomes low, it is obvious that *TD-Close* outperforms *FPclose* and *Carpenter* much.

5.2 Real Data Sets

Besides testing on the synthetic data sets, we also tested our algorithm on three real *microarray data* sets. They are clinical data on ALL-AML leukemia, lung cancer and breast cancer [5, 6]. We take the corresponding training data sets for experiments. ALL-AML has 38 rows and 7129 dimensions, Lung Cancer has 32 rows and 12533 dimensions, and Breast Cancer has 78 rows and 24481 dimensions. Before using frequent pattern mining algorithms, they are all discretized using the equi-depth binning method. To test the performance on different cardinality, we produce two sets of discretized data sets, one with five bins for each dimension, and another with 10 bins for each dimension.

The first group of experiments is done for these three data sets with 5 bins per dimension. Figures 5.9 to 5.11 show the runtime of *TD-Close*, *Carpenter,* and *FPclose* at different *minsup* values. Note that the *y*-axes in these figures are in logarithmic scale, and we plot the figure as *minsup* increase so that we can see the pruning power of *minsup* clearly.

Figure 5.9 shows that as *minsup* increases the runtime of both *TD-Close* and *Fpclose* reduces dramatically, while the runtime of *Carpenter* remains relatively stable. This is because *Carpenter* still needs to search the rowset space in which the size of each rowset is less than *minsup* although apparently they will not satisfy *minsup*. This figure also indicates that among these three algorithms, *TD-Close* is the fastest. We did not get the runtime of *FPclose* when *minsup* is

7, because when *minsup* is 8, it already spends much longer time than the other two algorithms.

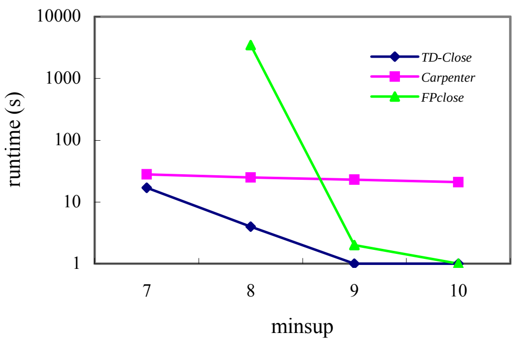

Figure 5.9 ALL-AML Leukemia

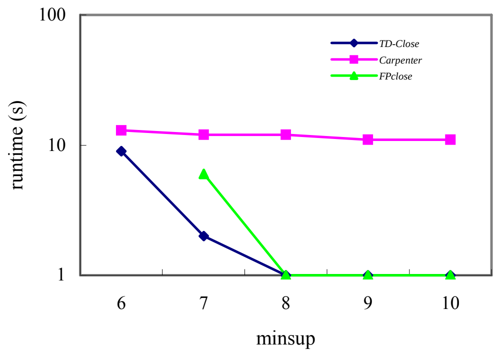

Figure 5.10 Lung Cancer

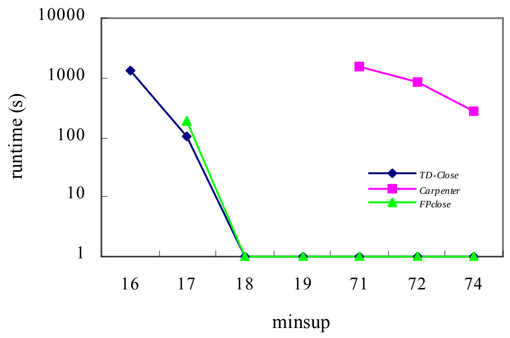

Figure 5.11 Breast Cancer

Figure 5.10 tells us almost the same situation as Figure 5.9 does. What is different is that *FPclose* cannot run successfully when *minsup* is 6 due to a memory error. This is because for this dataset, the number of dimensions is large. When *minsup* is 6, the number of itemsets needed to check is huge. While for row enumeration algorithms *TD-Close* and *Carpenter*, they only check rowsets instead of itemsets, and the number of rowsets does not change that dramatically.

For breast cancer data set, we cannot get the rumtime for *FPclose* when *minsup* is equal to or less than 16 because of memory error (run out of memory). Similarly, *Carpenter* cannot run to completion when *minsup* is 18 after about 11 hours of running. Therefore, in Figure 5.11, we can only see the runtime for *Carpenter* when *minsup* is not less than 71, and the runtime for *FPclose* when *minsup* is greater than 17. For *FPclose* and *TD-Close*, when the value of minsup is between 20 and 70, the runtime is less than one second, so these values are not shown in the x-axis.

The results of these experiments for the real world microarray data sets illustrate that *TD-Close* and *Fpclose* can make use of the anti-monotonic constraint *minsup* to prune the search space dramatically, while *Carpenter* cannot. Since the pruning strategies used in *TD-Close* benefit from the top-down search strategy, we can conclude that this search strategy is more effective and useful compared to the bottom-up search strategy for row enumeration-based search algorithms. Also, for very high dimensional dataset, row enumeration-based algorithm *TD-Close* outperform column enumeration-based algorithm *FPclose* very much.

6 Related Work

Since the last decade, many algorithms [1, 10, 11, 12, 13, 18, 19] have been proposed to find frequent itemsets from not very high dimensional data sets, such as transactional data sets. Suppose there are *n* different items in data set, these algorithms usually adopt a bottom-up strategy to search the itemset space that could be as large as 2^n. By bottom-up search, these algorithms can use the *minsup* threshold to stop further search the superset of an itemset once this itemset does not satisfy the *minsup* threshold. However, for very high dimensional data sets, since *n* becomes very large, the search space becomes huge. This leads to the low performance of these algorithms for very high dimensional data. As a result, a new group of algorithms [5, 6, 7, 8] were proposed to deal with long *microarray* data. Our work is directly related to these algorithms. In [5], an algorithm called *Carpenter* was proposed to find all of the frequent closed itemsets. *Carpenter* conducts a depth-first order traversal of the row-enumeration as shown in Figure 3.1, and checks each rowset corresponding to the node visited to see if it is frequent and closed. In [6], an algorithm called *Farmer* is developed to find the set of association-based classification rules. Farmer also searches the row enumeration tree by depth-first order. In [7], Algorithm *Cobbler* is proposed to find frequent closed itemsets by integrating row enumeration method with column enumeration method. It shows its high performance by conducting experiments on a data set with high dimension and a relatively large number of rows. In [8], an algorithm is proposed to find the top-*k* classification rules for each row of the *microarray* data set. All of these algorithms aim to facilitate the mining of frequent pattern by searching the row enumeration space, and they all search the space in a top-down style.

7 Conclusions

In this paper we propose a top-down search strategy for mining frequent closed patterns from very high dimensional data such as *microarray data*. Existing algorithms, such as *Carpenter* and several other related algorithms, adopt a bottom-up fashion to search the row enumeration space, which makes the pruning power of minimum support threshold (*minsup*) very weak, and therefore results in long mining process, even for high *minsup*, and much memory cost. To solve this problem, based on our top-down search strategy, a top-down style row enumeration method and an effective *closeness-checking* method are proposed. A new algorithm, *TD-Close*, is designed and implemented for mining a complete set of frequent closed itemsets from high dimensional data. Several pruning strategies are developed to speed up searching. Both analysis and experimental study show that these methods are effective and useful. Future work includes integrating top-down row enumeration method and column row enumeration method for frequent pattern mining from both long and deep large datasets, and mining classification rules based on association rules using top-down searching strategy.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases* (VLDB'94), pp. 487–499, Sept. 1994.

[2] C. Niehrs and N. Pollet. Synexpression groups in eukaryotes. *Nature*, 402: 483-487, 1999.

[3] Y. Cheng and G. M. Church. Biclustering of expression data. In *Proc of the 8th Intl. Conf. Intelligent Systems for Mocular Biology*, 2000.

[4] J. Yang, H. Wang, W. Wang, and P. S. Yu. Enhanced Biclustering on Gene Expression data. In *Proc. of the 3rd IEEE Symposium on Bioinformatics and Bioengineering* (*BIBE*), Washington DC, Mar. 2003.

[5] F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki. CARPENTER: Finding closed patterns in long biological datasets. In *Proc. 2003 ACM SIGKDD Int. Conf. On Knowledge Discovery and Data Mining*, 2003.

[6] G. Cong, A. K. H. Tung, X. Xu, F. Pan, and J. Yang. FARMER: Finding interesting rule groups in microarray datasets. In *Proc. 23rd ACM Int. Conf. Management of Data*, 2004.

[7] F Pan, A. K. H. Tung, G. Cong, X. Xu. COBBLER: Combining column and Row Enumeration for Closed Pattern Discovery. In *Proc 2004 Int. Conf. on Scientific and Statistical Database Management* (SSDBM'04), Santorini Island, Greece, June 2004. pp. 21-30.

[8] G. Cong, K.-L. Tan, A. K. H. Tung, X. Xu. Mining Top-k covering Rule Groups for Gene Expression Data. In *24th ACM International Conference on Management of Data*, 2005.

[9] C. Creighton and S. Hanash. Mining gene expression databases for association rules. *Bioinformatics*, 19, 2003.

[10] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery* (DMKD'00), pp. 11-20, Dallas, TX, May 2000.

[11] M. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. In *Proc. of 2002 SIAM Data Mining Conf.*, 2002.

[12] J. Han and J. Pei. Mining frequent patterns by pattern growth: methodology and implications. *KDD Exploration*, 2, 2000.

[13] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th Int. Conf. Database Theory* (ICDT'99), Jan. 1999.

[14] J. Li and L. Wong. Identifying good diagnostic genes or genes groups from gene expression data by using the concept of emerging patterns. *Bioinformatics*, 18:725–734, 2002.

[15] G. Grahne and J. Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*. Melbourne, Florida, Nov., 2003

[16] http://fimi.cs.helsinki.fi/fimi03/

[17] http://fimi.cs.helsinki.fi/.

[18] M. J. Zaki. Scalable algorithms for association mining. *IEEE Trans. Knowledge and Data Engineering*, 12:372{390, 2000.

[19] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. 2003 ACM SIGKDD Int. Conf. On Knowledge Discovery and Data Mining* (KDD'03), Washington, D.C., Aug 2003.