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* ($i = 1, 2, \ldots, n$) 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

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

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

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 $|r(I)| \geq minsup$, and a rowset S is called *large* if $|S| \geq minsup$, 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.

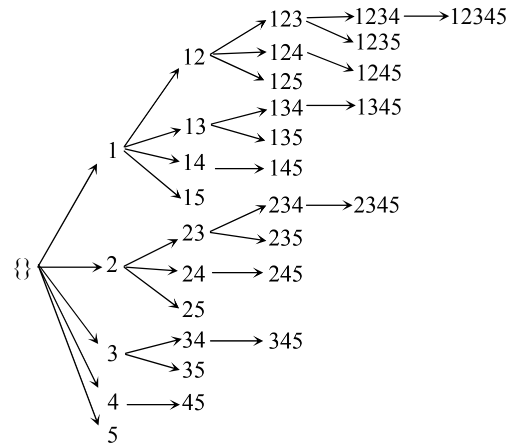

Figure 3.1 Bottom-up row enumeration tree

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.

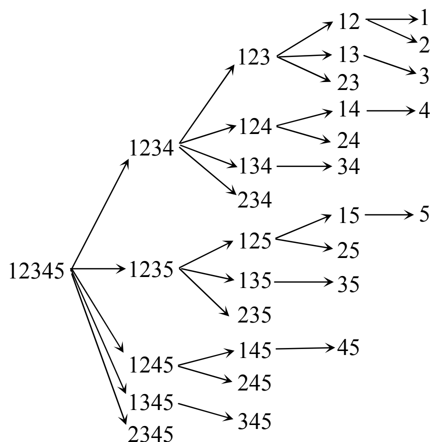

Figure 3.2 Top-down row enumeration tree

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.

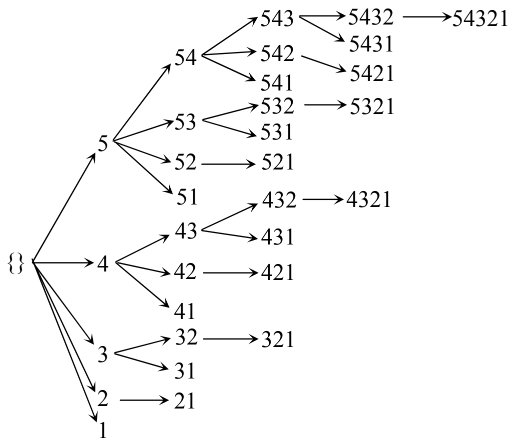

Figure 3.3 Excluded row enumeration tree

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.