

An Efficient Union Approach to Mining Closed Large Itemsets in DNA Microarray Datasets *

Ye-In Chang Jiun-Rung Chen Li-Wen Lee

National Sun Yat-Sen University/Dept. of Computer Science and Engineering
changyi@cse.nsysu.edu.tw

摘要

在DNA microarray中尋找closed large itemset，可以幫助我們了解基因是如何互相影響，及哪些基因常常會有一致性的表現。在本篇論文中，針對如何有效率地在DNA microarray中尋找closed large itemset，我們提出了一個新的方法，UMiner。這個方法採用了row enumeration的方法，並利用了聯集運算，來降低在探索row enumeration tree時所需的時間。此外，我們也提出了幾項刪減的技術，可以減少需要探索的row enumeration node數量。從實驗結果顯示，我們的方法在任何的support值及任意的closed large itemset平均長度，都能夠比前人所提出的RERII方法好。

關鍵字：封閉式項目集、基因表現、大項目集、微陣列、列列舉

Abstract

Mining closed large itemsets in DNA microarray datasets can help us know how genes affect each other, and what genes are usually co-expressed. In this paper, we propose a new method, UMiner, which utilizes row enumeration to mine closed large itemsets in the DNA microarray datasets. By applying the technique of union operations, we can reduce the time complexity of exploring each node in the row enumeration tree. Moreover, we develop four pruning techniques to reduce the number of nodes needed to be explored in the row enumeration tree. From the experiment results, we show that our UMiner method is always faster than the RERII method in all support values, no matter what the average length of the closed large itemsets is.

Keywords: closed itemset, gene expression, large itemset, microarray, row enumeration

1. Introduction

DNA Microarrays have opened the possibility of creating datasets of molecular information to represent many systems of biological or clinical interest [3]. A typical DNA microarray usually has a small number of records (less than one hundred), while the number of genes is typically in thousands. Since the dataset which comes from the biological experiments is large, many data mining techniques are used to study the data efficiently. Biologists need to know how genes or proteins

TID	Items			
1	a, b	L1	a:4 b:4 d:4	
2	a, b, d			
3	c, d	L2	ab:4 ad:3 bd:3	
4	a, b, d			
5	a, b, d	L3	abd:3	

(a)

(b)

(c)

Figure 1: An example of mining association rules: (a) the original dataset; (b) large itemsets; (c) closed large itemsets.

interact with each other, in order to help them understand what happened in a cell, or help them construct the gene network. Since the technique of *mining association rules* is used to find the associations of items, it can provide great help in bioinformatics. The technique of mining association rules is often applied in traditional transaction database. For example, it could find that many customers which buy bread will also buy milk in the transaction database. Then, “bread → milk” is an association rule. There is an example of mining association rules in protein databases shown in [4].

The main goal of mining association rules is to find the *large itemsets*, where a large itemset is a combination of items whose appearing times (*i.e.*, *support*) in the datasets is greater than a given threshold (*i.e.*, *the minimum support*). Closed large itemsets can further reduce the redundancy of large itemsets and do not lose any information, where a *closed large itemset* is a large itemset which has no superset with the same support as it [6]. Take table *T* in Figure 1-(a) as an example. Figure 1-(b) shows the large itemsets of table *T*, where the value of the minimal support is 2. By the characteristic of a large itemset — all subsets of a large itemset are also large itemsets, large itemsets such as “*ad*” and “*bd*” seem very obvious and are redundant. Figure 1-(c) shows the closed large itemsets of table *T*. Large itemset “*ab*” is a closed large itemset because there is no its superset (*e.g.*, “*abd*”) that has the same support count as it. Large itemset “*bd*” is not a closed large itemset, since large itemset “*abd*” contains “*bd*” and has the same support as “*bd*”. From this example, we could see that closed large itemsets can greatly reduce the number of large itemsets.

Therefore, there have been many methods proposed which focus on mining closed large itemsets. These methods can be roughly divided into three types: (1)

*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.

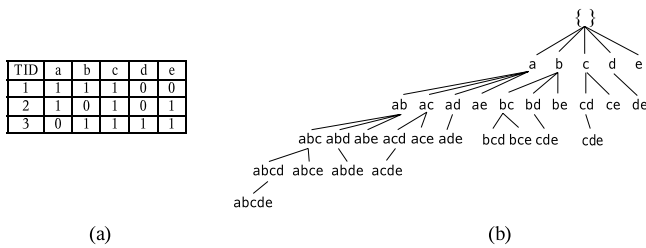


Figure 2: An example of column enumeration: (a) table P ; (b) the column enumeration tree of table P .

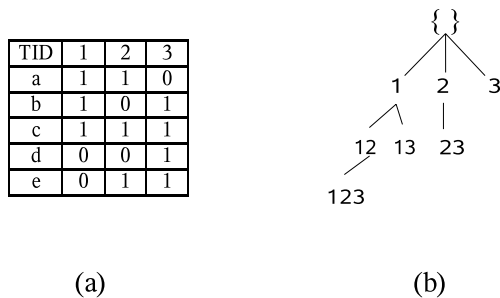


Figure 3: An example of row enumeration: (a) table TP , *i.e.*, the transposed table of table P ; (b) the column enumeration tree of table TP (*i.e.*, the row enumeration tree of table P).

using column enumeration; (2) using row enumeration; (3) using both column and row enumeration, where the column (row) enumeration means that enumerating itemsets from the combinations of the items stored in columns (rows). Take table P shown in Figure 2-(a) as an example, whose number of columns is larger than the number of rows. Figure 2-(b) shows the corresponding column enumeration tree. By exchanging columns and rows of table P , Figure 3-(a) shows the transposed table TP . The corresponding column enumeration tree of table TP , *i.e.*, the row enumeration tree of table P , is shown in Figure 3-(b). From Figure 3-(b), we observe that the number of nodes in the row enumeration tree of table P is less than that in the column enumeration tree. Recall that a DNA microarray dataset has the property that the number of columns is much larger than the number of rows. Therefore, row enumeration is more efficient than column enumeration when mining large itemsets in a DNA microarray. (Note that we could transform a DNA microarray into a table with items by transforming the numeric value in each entry into “0” and “1”, where “0” means this entry (or item) will not appear in this row. We could define a threshold, W , which determine whether an entry is “0” ($\leq W$) or “1” ($> W$).)

There have been several row enumeration methods proposed, *e.g.*, CARPENTER [5], FARMER [2], and RERII [1]. The CARPENTER method [5] and the FARMER method [2] both use a transposed table and the conditional transposed tables to generate the closed large itemsets. When the size of the row enumeration tree becomes large, these conditional transposed tables will cost lots of memory space. The RERII method [1] does not use the transposed table and the condi-

tional transposed tables for each enumeration node in the row enumeration tree, which saves lots of memory. Moreover, the RERII method uses several pruning techniques to make it more efficient than the CARPENTER method and the FARMER method.

Although the RERII method has better performance than the FARMER method and the CARPENTER method, the intersection operations at each enumeration node of the RERII method are very complex. The RERII method intersects two closed itemsets to get a new closed itemset for each enumeration node in the row enumeration tree. If the length of two closed itemsets are m and n , intersecting them to get a new closed itemset will need $(m+n-1)$ comparisons, which is similar to the performance of a merge sort. When the average number of items in the rows is large, *e.g.*, a DNA microarray dataset, the intersection operation will cost lots of time.

Therefore, in this paper, we propose a new method, UMiner, for mining closed large itemsets in the DNA microarray datasets. The UMiner method is based on the union operations, instead of the complex intersections and the string comparison operations in the RERII method. We group the items into item groups, and generate the itemset for each enumeration node by uniting the itemset of these item groups when we explore the enumeration tree. Moreover, we design four pruning techniques to prune the unnecessary branches in the row enumeration tree. From our performance study, we also show that our UMiner method is faster than the RERII method when using real DNA microarray datasets.

2. The RERII Method

Gao *et al.* proposed the RERII method [1], which is an improved version of the CARPENTER method [5]. Instead of using conditional transposed tables as the CARPENTER method, the RERII method stores the information that is used to help tree traversing in the row enumeration tree itself. Each node X of the tree will be represented as a three-element group $X = \{itemlist, sup, childlist\}$, where “*itemlist*” is the closed itemset corresponding to node X , “*sup*” is the number of rows at the node, and “*childlist*” is the list of child nodes of X . The RERII method will perform an intersection of the *itemlist* of node X with the *itemlists* of all its sibling nodes after X . Each intersection will result in a new node whose *itemlist* is the intersection, *sup* is $(X.sup + 1)$, and *childlist* will be available at the next level intersection. In this way, the row enumeration tree will be recursively expanded in a depth-first way. The RERII method also utilizes several pruning techniques to reduce the number of nodes needed to be explored. However, the intersection operations performed in each node will be very time-consuming.

3. A Union Approach

In our method, we will group items into *item groups* according to a *prefix tree* created from the transposed table, and use a *row-node table* to store them. If a

Row ID	Item List
1	b,d,e,f
2	a,c,e,f
3	a,c,d,e
4	a,b,c,d,e,g
5	a,b,c,d,e,f

(a)

Item	Row ID List
a	2,3,4,5
b	1,4,5
c	2,3,4,5
d	1,3,4,5
e	1,2,3,4,5
f	1,2,5
g	4

(b)

Figure 4: Transposing the original table to the transposed table: (a) the original table D ; (b) the transposed table TT .

large number of items could be grouped together, the number of item groups will decrease a lot, and the improvement of performance of our method will be large. We use the row enumeration approach to enumerate the combinations of rows, and use the row-node table to help us to generate the closed itemset for each combination of rows. There are three steps in our method. First, we add all tuples in the transposed table to a prefix tree. Then, we traverse the prefix tree created in the first step to create a row-node table. Finally, we explore the row enumeration tree with the help of the row-node table to generate closed large itemsets, and apply several pruning techniques to reduce the number of enumeration nodes in the row enumeration tree of our method.

In the following three subsections, we will describe the details of these three steps. Take the dataset D in Figure 4-(a) as our example. The transposed table TT of dataset D is shown in Figure 4-(b), where each tuple consists of an item and a row ID list.

3.1 Creating the Prefix Tree

The first step of grouping items is to create a prefix tree from the transposed table. In transposed table TT , each tuple t with the size of its row ID list (*i.e.*, the support of the item of this tuple) larger than the minimal support is added to the prefix tree from the root node. Each node in the prefix tree consists of a row ID and an item list. Figure 5 shows an example of how we create the prefix tree from transposed table TT shown in Figure 4-(b). If there exists a path whose row IDs are the same as the row ID list of tuple t , we directly add the item of t to the item list of each node along this path. Figure 5-(b) shows this case. Otherwise, we create a new branch for those row IDs in tuple t . Figure 5-(c) shows this case. Figure 6 shows the prefix tree after all tuples in transposed table TT shown in Figure 4-(b) are added to it. After we add all tuples in the transposed table TT to the prefix tree, we will use this tree to create a row-node table.

3.2 Creating a Row-Node Table

After the prefix tree has been generated, we will traverse this prefix tree once in a depth-first manner to create a row-node table. We assign Num to each node in an ascending order, and record every node that we encounter to the row-node table according to the row ID of the node, when we traverse the prefix tree. If

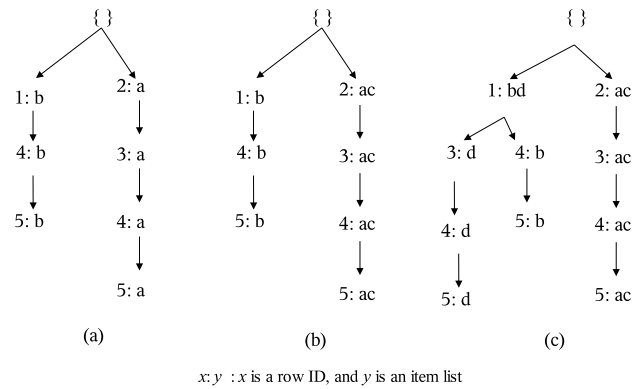


Figure 5: Adding tuples to the prefix tree: (a) adding tuples “a: 2, 3, 4, 5” and “b: 1, 4, 5”; (b) adding tuple “c: 2, 3, 4, 5”; (c) adding tuple “d: 1, 3, 4, 5”.

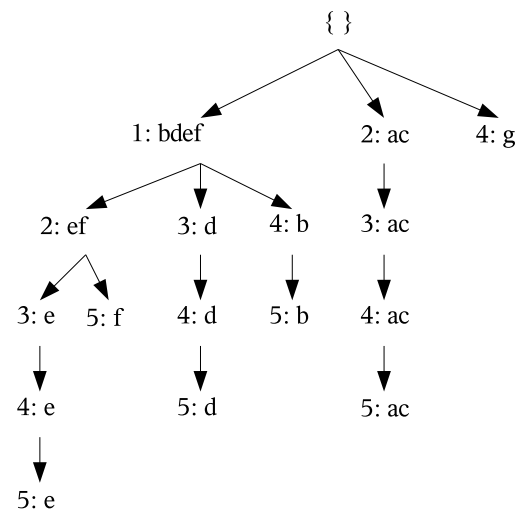


Figure 6: The prefix tree generated from the transposed table TT

the itemset of a node is the same as the itemset of its parent node in the prefix tree, we assign the same value of Num to this node as its parent node. In a row-node table, each node in the prefix tree will be recorded as $\langle Num, itemlist, r_{begin}, r_{end} \rangle$, where Num is the ID of a node in the prefix tree, and $itemlist$ is the related item list of this node. Moreover, r_{begin} and r_{end} denote the minimal and maximal Num 's of the child nodes of this node in the prefix tree, respectively. Take the prefix tree shown in Figure 6 as an example. Figure 7-(a) shows the labeled prefix tree after being traversed, and Figure 7-(b) shows the row-node table generated from this labeled prefix tree. In the row-node table, the row ID of the first tuple is 1, which means that all nodes in the prefix tree whose row ID is 1 will be recorded in this tuple. The item group list of the first tuple is “ $\langle (1), bdef, (2), (6) \rangle$ ”, which means that the node ID assigned to the recorded node is (1), and it contains itemset “bdef”. Moreover, nodes with node ID from (2) to (6) in the prefix tree are all its child nodes. After the row-node table has been generated, we can generate the closed itemsets by uniting the itemsets of some nodes in the item group list in a tuple.

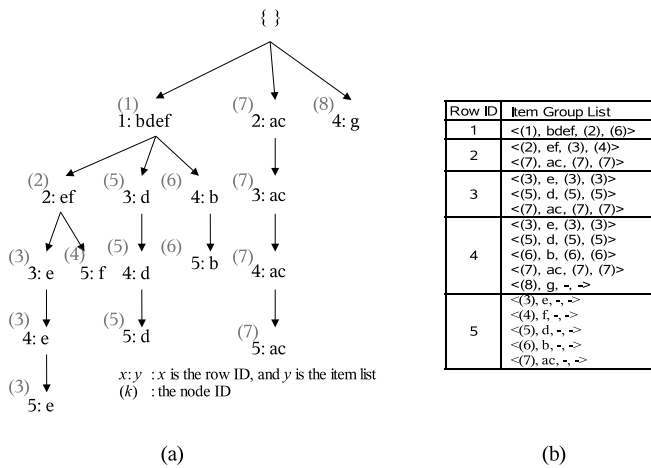


Figure 7: The labeled prefix tree and the row-node table: (a) the labeled prefix tree; (b) the row-node table *RNT* table.

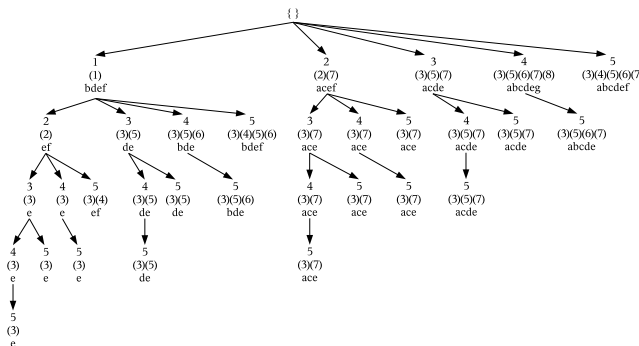


Figure 8: The complete enumeration tree without pruning

3.3 Exploring the Row Enumeration Tree

After the row-node table is created, we can use it to help us generate closed itemsets. We use the row enumeration approach to enumerate the rowsets for searching closed large itemsets, where a rowset is a set of rows that have been enumerated. Figure 8 shows the complete row enumeration tree of the transposed table *TT* in Figure 4-(b) after exploring. We will explain how each node in this tree is explored later. In our method, the items are grouped into several groups according to the prefix tree, and the information about these groups is recorded in the item group lists of the tuples in a row-node table. We call such a pattern in the item group list of a tuple in the row-node table as an *item group*. In Figure 8, a number without parentheses is the row ID (denoted as *RID*), and a number with parentheses is the node ID (denoted as *Num*) of the related item groups. Moreover, the letters under *Num* are the items in a closed itemset. Since each closed itemset will be bind to one unique combination of rows, we can find all closed itemsets by enumerating all combinations of rows.

Let *RowSets* be a set of all row ID's in the dataset. Take the dataset in Figure 4-(a) as an example. Initially, we have the *RowSets* = {1, 2, 3, 4, 5}. First, for each row ID *RID* in *RowSets*, we create a new enumer-

ation node with its row ID = *RID* under the root node of the row enumeration tree. Next, We find *NList*, which is a list of *Num*'s (i.e., the node ID's) of the item groups, for each new enumeration node. *NList* of a enumeration node at the first level with its row ID = *RID* is a set of *Num*'s of all the item groups in the tuple whose row ID = *RID* in the row-node table. Take the enumeration node whose row ID = 2 at the first level of the tree shown in Figure 8 as an example. *NList* of this node is a set of *Num*'s which are obtained from the item group list of the second row (i.e., row ID = 2) in Figure 7-(b), i.e., "(2)(7)". Then, we generate the closed itemset for each new enumeration node by uniting the itemsets of the item groups whose node ID's are in *NList* of each node. In the previous example, *NList* of that node is "(2)(7)". The itemsets of the item groups (2) and (7) are "ef" and "ac", respectively. By uniting itemset "ef" and itemset "ac", the closed itemset of that node will be "acef". Finally, for these new enumeration nodes, we add those whose *NList* is not empty to an enumeration list, denoted as *EList*, as candidates. For each candidate node in *EList*, we call Procedure *GetClosed* to explore the subtree of each candidate node recursively.

In Procedure *GetClose*, we first find the support value of the current node by adding the support value of its parent node by 1. In other words, the support value of a closed itemset at an enumeration node is the depth of this node in the row enumeration tree. Next, for each candidate node, denoted as *cnode*, we create a new enumeration node under the current node with its row ID = *RID*. We will find *NList* and the closed itemset for each new enumeration node with its row ID = *RID*. Take node "3" (i.e., the node whose row ID = 3) along path "2 3" in the row enumeration tree shown in Figure 8 as an example. In the parent node of node "3", there are two item groups whose *Num*'s are (2) and (7). First, we get a range "(3) - (4)" from the item group whose *Num* is (2), and get another range "(7) - (7)" from the item group whose *Num* is (7). Since the row ID of node "3" is 3, we search the tuple whose row ID = 3 for the item groups whose *Num*'s are in range "(3) - (4)" or "(7) - (7)". For range "(3) - (4)", we find an item group whose *Num* is (3). For range "(7) - (7)", we find an item group whose *Num* is (7). Therefore, the *NList* of node "3" is "(3)(7)".

All new enumeration nodes whose *NList* is not empty will be added to a new *EList* (denoted as *NewEList*) of the current node as candidates for recursively exploring. Then, the closed itemset of the new enumeration node will be generated by uniting the itemset of the item groups whose *Num* is in *NList* of the current node. In the previous example, the closed itemset of node "3" is the union of the itemsets of the item group with *Num* = (3) and the item group with *Num* = (7). The itemset of the item group with *Num* = (3) is "e", and the itemset of the item group with *Num* = (7) is "ac". Therefore, the closed itemset of node "3" is "ace". Then, if the support value of the closed itemset of the current node is larger than the minimal support, this itemset is a closed large itemset and is stored to

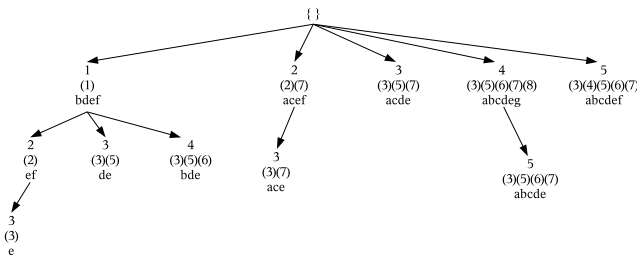


Figure 9: The pruned enumeration tree in our method

CLI.

By recursively calling Procedure *GetClose* to explore the subtree of each candidate node in *NewEList*, we can explore the complete row enumeration tree. After the exploration of the row enumeration is done, all closed large itemsets will be in *CLI*.

3.4 Pruning Techniques

Although we can find all of the closed itemsets by exploring entire row enumeration tree, many closed itemsets are redundant in the exploring process. It is not necessary to search the complete row enumeration tree, and we can prune some branches of the tree when the nodes satisfy some specific conditions.

First, if *NList* of an enumeration node is empty, we do not expand the subtree under this enumeration node. Since “no item group at this node” means “no closed itemset at this node”, all enumerations under this node will also have no closed itemset.

Second, if *Itemset* (*i.e.*, the closed itemset) of a child node is the same as the *Itemset* of the current enumeration node, the support value of the *Itemset* of the current node is increased by 1, and we do not expand the subtree under this child node. This is because we integrate the child node into the current node, and all enumeration nodes under this child node will be enumerated under the current node.

Third, if the sum of the support value of the current node and $|\{\forall cnode | cnode \in EList, cnode.RID > RID \text{ of the current node}\}|$ (or $|NewEList|$) is less than the minimal support, no further expanding under the current node is needed. Since $|NewEList|$ is the maximal possible height of the subtree under the current node, the sum means the maximal possible support value of the nodes in the subtree under the current node. If the sum is not larger than the minimal support, it means that there is no closed itemset whose support value is larger than the minimal support in this subtree.

Fourth, if the closed itemset of the current node has been found previously, no further expanding under this node is needed. The closed itemset of a child node in the subtree of the current node can only be a subset of the closed itemset of the current node in the row enumeration tree. Since the closed itemset of the current node has been found previously, it means that all possible subsets of the closed itemset of the current node has been found previously, too.

With these pruning techniques, we can greatly reduce our searching domain. The row enumeration tree after pruning is shown in Figure 9.

Table 1: The detailed information about the real DNA microarray datasets

Name	Rows	Columns
Breast Cancer	78	24481
Lung Cancer	31	12533
AML-ALL	37	7129

4. Performance

In the section, we study the performance of the UMiner method, and make a comparison with the RERII method. The experiments were performed on the machine with a AMD Sempron 2500+ CPU, 512 MB of main memory, running Gentoo Linux 2006.0 with kernel version 2.6.15-r1, and both methods were implemented in Java. Our experiments were run with three real microarray datasets, and were repeated for 20 times to calculate the average processing time.

4.1 The Real DNA Microarray Dataset

In order to study the performance of our method in the real situation, we use the real datasets which are the clinical datasets on breast cancer, lung cancer, and AML-ALL. The text versions of these data can be downloaded from <http://sdmc.lit.org.sg/GEDatasets/Datasets.html>.

The detailed information about these datasets is shown in Table 1. Before we use these datasets in our method and the RERII method, we need to transform the numeric value in each entry into “1” and “0”. We declare a variable, W , to be our threshold for defining whether a gene is up (*i.e.*, “1” if the entry value $> W$) or down (*i.e.*, “0” if the entry value $\leq W$). When the value of W decreases, the number of “1” will increase, which results in the increase of the average number of items in the closed large itemsets. By changing the value of W , we can change the average length of the closed large itemsets. The value of W of each dataset is chosen as low as possible such that both of the RERII method and our method will not run out of memory. We will measure the processing time of both our method and the RERII method from reading the dataset into the main memory, to the end of completely exploring the row enumeration tree.

4.2 Experiment Results

First, we show the comparison of the processing time between our UMiner method and the RERII method, when using the breast cancer dataset as the input. The value of W of this dataset is set to 0.6. Figure 10 shows the comparison of the processing time. From Figure 10, we show that the UMiner method is faster than the RERII method, no matter what value the minimal support is.

Figure 11 and Figure 12 show the experiment results when using the lung cancer dataset and the AML-ALL dataset as the inputs, respectively. The values of W are set to 1700 and 8200, respectively. From these two figures, we show that the UMiner method is faster than the RERII method, no matter what value the minimal support is.

