Mining Frequent Itemsets without Support Threshold: with and without Item Constraints

Yin-Ling Cheung, Ada Wai-Chee Fu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
{ylcheung, adafu}@cse.cuhk.edu.hk

Abstract

In classical association rules mining, a minimum support threshold is assumed to be available for mining frequent itemsets. However, setting such a threshold is typically hard. In this paper, we handle a more practical problem, roughly speaking, it is to mine N k-itemsets with the highest supports for k up to a certain k_{max} value. We call the results the N-most interesting itemsets. Generally, it is more straightforward for users to determine N and k_{max}. We propose two new algorithms, *LOOPBACK* and *BOMO*. Experiments show that our methods outperform the previously proposed Itemset-Loop algorithm, and the performance of BOMO can be an order of magnitude better than the original FP-tree algorithm even with the assumption of an optimally chosen support threshold. We also propose the mining of "N-most interesting k-itemsets with item constraints". This allows user to specify different degrees of interestingness for different itemsets. Experiments show that our proposed *Double FP-trees* algorithm, which is based on BOMO, is highly efficient in solving this problem.

Index: Association rules, N-most interesting itemsets, FP-tree, item constraints.

1 Introduction

Mining association rules is an important problem in data mining [1, 3, 4, 7, 8, 9, 11, 12, 13, 15, 20, 21, 23]. An example of such a rule is: $\forall x \in persons, buys(x, \text{“}bread\text{”}) \Rightarrow buys(x, \text{“}butter\text{”})$ where x is a variable and *buy(x,y)* is a predicate that says that person x buys item y. This rule indicates that a high percentage of people who buy bread also buy butter. A more formal definition of association rules is given in [3]. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Any subset of I is called an **itemset**. Let D be a database of a set of transactions. Each transaction, T, consists of a set of items from I, i.e., $T \subseteq I$. A transaction T is said to contain

X if $X \subseteq T$, and $X \subseteq I$. An association rule has the form $X \Longrightarrow Y$, where $X, Y \subset I$, and $X \cap Y = \emptyset$. The **support** of a rule $X \Longrightarrow Y$ is $s\%$ in D if $s\%$ of the transactions in D contain $X \cup Y$. The **support** of an itemset Z is the percentage of transactions in D containing Z. The **confidence** of a rule $X \Longrightarrow Y$ is $c\%$ if $c\%$ of the transactions in D that contain X also contain Y. Typically, this method requires a user specified minimum support threshold and a minimum confidence threshold, and all the association rules having support above the minimum support threshold and confidence above the confidence threshold are returned as answer.

Mining association rules can be divided into two steps: (1) finding all the itemsets having sufficient supports; (2) generating association rules from these frequent (or large) itemsets. After Step (1), Step (2) is straightforward and trivial in terms of computational time, hence a lot of effort is focused on Step (1), i.e., we would like to mine all itemsets of cardinality > 1, whose supports are greater than a certain threshold value.

However, without specific knowledge, users will have difficulties in setting the support threshold to obtain their required results. If the support threshold is set too large, there may be only a small number of results or even no result. In which case, the user may have to guess a smaller threshold and do the mining again, which may or may not give a better result. If the threshold is too small, there may be too many results for the users, too many results can imply an exceedingly long time in the computation, and also extra efforts to screen the answers. As an example of the difficulty in choosing a threshold, for the census data of United States 1990 available at the web site of IPUMS-98 [10], for two different sets of data, the thresholds for finding a reasonable number of itemsets are found to differ by an order of magnitude. See Tables 4 and 5 in Section 5 for an example where the reasonable support thresholds may vary from 0.3% to 21.42% for different data sets of different nature.

In [16], and also in [18], mining based on "closed itemsets" is addressed. For a closed itemset X, no superset is contained in the same set of transactions that contain X. This partially alleviate the problem of the result size. However, it does not address the basic problem of setting a suitable support threshold.

Another argument against the use of a uniform threshold for all itemsets is that the probability of occurrence of a larger size itemset is inherently much smaller than that of a smaller size itemset. From our observations, it would be better for users to specify a threshold on the amount of results instead of a fixed threshold value for all itemsets of all sizes. For example, in multimedia data querying, we find that it is much more natural to allow users to specify the number of nearest neighbors rather than to specify a certain threshold on the "distance" from their query point. We have first introduced this problem definition for mining association rules in [7].

Definition 1 *A* k**-itemset** *is a set of items containing* k *items.*

Definition 2 *The* N**-most interesting** k**-itemsets** *: Let us sort the* k*-itemsets by descending support values, let* S *be the support of the* N*-th* k*-itemset in the sorted list. The* N*-most interesting* k*-itemsets are the set of* k*-itemsets having support* $\geq S$*.* [1]

Definition 3 *The* N**-most interesting itemsets** *is the union of the* N*-most interesting* k*-itemsets for each* $1 \leq k \leq k_{max}$*, where* k_{max} *is the upper bound of the size of itemsets we would like to find. We say that an itemset in the* N*-most interesting itemsets is* **interesting**.

Other than basket data, an important application for mining the N-most interesting itemsets is for the problem of *subspace clustering*, where we are interested to find clusters hidden in different subspaces of the given data space. In particular, it can be applied to the problem and methodology as set up in [2].

Recent work has highlighted the significance of **constraint-based mining technique** [19, 14, 17], users can specify their focus in mining, by means of a specific set of constraints that allow them to explore and control the interesting patterns. For example: In a supermarket, we may only want to know the relationships among items of particular types, such as soft drink and alcohol. [25] proposes *support constraints* as a way to specify general constraints on minimum support. It employs a support pushing technique that allows the highest possible minimum support to be pushed so as to tighten up the search space while preserving the essence of Apriori. However, the algorithm still requires a user specified support threshold. In Section 3, we define a more practical problem of finding N-most frequent itemsets with constraints as defined in [25, 17], and propose some solutions to this problem.

In the next section, we give some background of our work. We present our new approaches for mining N-most interesting itemsets in Section 3. A preliminary version of the above result can be found in [6]. We then define the item constraints problem in Section 4, and introduce the Double FP-trees algorithm. We also consider the case of using maximum support thresholds in Section 4.3. Experimental results are given in Section 5.

[1] If multiple itemsets have the same support S, we pick all of them according to Definition 2. Therefore, the resulting set may contain more than N itemsets. There is an extreme case where too many patterns exist with the same support S, in this case, we suggest to report the scenario to user instead of returning all the patterns.

2 Background

To our knowledge, Itemset-Loop [7] is the first algorithm for the mining of N-most interesting itemsets. Itemset-Loop makes use of the apriori candidate generation mechanism which relies on the property of subset closure: if a k-itemset is large then all its subsets are also large. This property does not hold for mining N-most interesting itemsets. That is, if a k-itemset is among the N-most interesting k-itemsets, its subsets may not be among the N-most interesting itemsets. Therefore we examine other algorithms for the association rule mining problem. We note that the FP-tree [9] approach does not rely on the candidate generation step. We therefore consider how to make use of the FP-tree for the N-most interesting itemsets mining.

An FP-tree (frequent pattern tree) is a variation of the *trie* data structure, which is a prefix-tree structure for storing crucial and compressed information about frequent patterns. It consists of one root labeled as "$NULL$", a set of item prefix subtrees as the children of the root, and a **frequent-item header table**. Each node in the item prefix subtree consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* indicates which item this node represents, *count* indicates the number of transactions containing items in the portion of the path reaching this node, and *node-link* links to the next node in the FP-tree carrying the same item-name, or null if there is none. Each entry in the frequent-item header table consists of two fields, *item-name* and *head of node-link*. The latter points to the first node in the FP-tree carrying the *item-name*.

TID	Items	Sorted Frequent Items
001	a, b, c, d	c, d, a, b
002	b, c, d, e	c, d, b, e
003	a, c, d	c, d, a
004	e, f	e

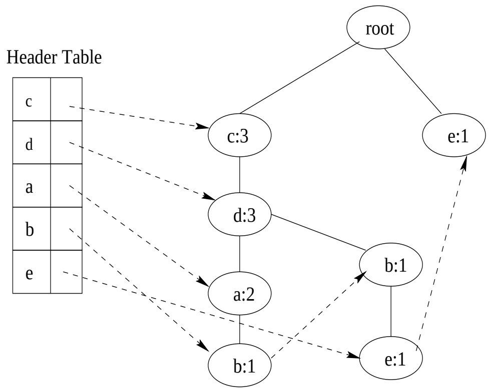


Figure 1: Database and corresponding FP-tree

Let us illustrate by an example the algorithm to build an FP-tree using a user specified minimum support threshold, ξ. Suppose we have a transaction database shown in Figure 1 with $\xi = 2$. By scanning the database, we get the sorted ($item$:$support$) pairs, $\langle$(c:3),

(d:3), (a:2), (b:2), (e:2), (f:1)⟩. The frequent 1-itemsets are: c, d, a, b, e. We use the tree construction algorithm in [9] to build the corresponding FP-tree. We scan each transaction and insert the frequent items (according to the above sorted order) to the tree. First, we insert $\langle c, d, a, b\rangle$ to the empty tree. This results in a single path: root($NULL$) $\rightarrow (c:1) \rightarrow (d:1) \rightarrow (a:1) \rightarrow (b:1)$. Then, we insert $\langle c, d, b, e\rangle$. This leads to two paths with c and d being the common prefixes: root($NULL$) $\rightarrow (c:2) \rightarrow (d:2) \rightarrow (a:1) \rightarrow (b:1)$, and root($NULL$) $\rightarrow (c:2) \rightarrow (d:2) \rightarrow (b:1) \rightarrow (e:1)$. Third, we insert $\langle c, d, a\rangle$. This time, no new node is created, but the first path is changed to: root($NULL$) $\rightarrow (c:3) \rightarrow (d:3) \rightarrow (a:2) \rightarrow (b:1)$. Finally, we insert $\langle e\rangle$ to the tree and we get the resulting tree as shown in Figure 1, which also shows the horizontal links for each frequent 1-itemset in dotted-line arrows.

With the initial FP-tree, we can mine frequent itemsets of size k, where $k \geq 2$. An FP-growth algorithm [9] is used for the mining phase. We may start from the bottom of the header table and consider item e first. There are two paths: $(c:3) \rightarrow (d:3) \rightarrow (b:1) \rightarrow (e:1)$ and $(e:1)$. Since the second path contains only item e, we get only one prefix path for e: ⟨ c:1, d:1, b:1 ⟩, which is called e's **conditional pattern base** (the count for each item is one because the prefix path only appears once together with e). We also call e the **base** of this conditional pattern base. Construction of an FP-tree on this conditional pattern base (**conditional FP-tree**), which acts as a transaction database with respect to item e, results in an empty tree since the support for each conditional item is less than ξ in the building phase. Next, we consider item b. We get the conditional pattern base: ⟨ c:1, d:1, a:1 ⟩, ⟨ c:1, d:1 ⟩. Construction of an FP-tree on this conditional pattern base results in an FP-tree with a single path: root($NULL$) $\rightarrow (c:2) \rightarrow (d:2)$. Mining this resulting FP-tree by forming all the possible combinations of items c and d with the appending of b. If we represent all frequent itemsets in the form of (itemset: count), we get the frequent itemsets $(cb:2)$, $(db:2)$, and $(cdb:2)$. Similarly, we consider items a, d and c to get the frequent itemsets. The resulting large itemsets after the mining phase are: $\{c:3, d:3, a:2, b:2, e:2, cd:3, ac:2, ad:2, bc:2, bd:2, acd:2, bcd:2\}$.

3 Mining N-most Interesting Itemsets

In this section, we introduce two new algorithms for mining N-most interesting itemsets. We adopt ideas of the FP-tree structure [9] in our algorithms. Let D be the given transaction database. We assume an upperbound of k_{max} on the size of interesting itemsets to be found.

We keep a current resulting set of N-most interesting k-itemsets, and call it $result_k$; a current support threshold ξ is kept for all the itemsets. The variable ξ_k stores the current support threshold for the k-itemsets.

3.1 A Build-Once and Mine-Once approach, BOMO

In the original FP-tree method [9], the FP-tree is built only with the items with sufficient support. However, in our problem setting, there is no support threshold given initially. We therefore propose to build a complete FP-tree with all items in the database. Note that this is equivalent to setting the initial support threshold ξ to zero. The size of an FP-tree is bounded by the size of its corresponding database because each transaction will contribute at most one path to the FP-tree, with the path length equal to the number of items in that transaction. Since there is often a lot of sharing of frequent items among transactions, the size of the tree is usually much smaller than its original database [9].

Although the initial value of ξ is zero, it will be dynamically increased as we progress with the mining step. Besides ξ, we use different thresholds for itemsets of different sizes. When nothing is known about the itemsets, we can set ξ_k to be zero for all k, $1 \leq k \leq k_{max}$. With $\xi_k = 0$, we would blindly include any k-itemsets in our current set of result, $result_k$. However, in the process of mining, we shall generate k-itemsets with their supports. Once we have encountered any N k-itemsets for some $k \leq k_{max}$, we know that we are interested in k-itemsets with supports at least as great as any of these N k-itemsets.

Observation 1 *Given any N k-itemsets for some $1 \leq k \leq k_{max}$, the N-most interesting k-itemsets have support $\geq$ any of these N itemsets.*

Therefore during the mining phase, we adjust ξ_k once we have found at least N itemsets of size k: we assign ξ_k to be the support of the N-th most frequent k-itemset discovered so far. In order to do this, we maintain a sorted list of the support values of the N most frequent k-itemsets discovered, for each $1 \leq k \leq k_{max}$. Figures 2, 3, and 4 show the main algorithm, the tree-building step and the mining step. The values of ξ_k are used to determine if an itemset should be included in the current $result_k$. Here is an example. Suppose $N = 5$ and $k_{max} = 3$, we set $\xi = 0$, $\xi_1 = \xi_2 = \xi_3 = \xi = 0$. Suppose we have found the following 1-itemsets: $\{ a : 8,\ b : 8,\ c : 6,\ d : 6,\ e : 4,\ f : 4 \}$. Then $\xi_1 = 4$. Considering large 2-itemsets, suppose $\{cd : 5, ab : 8, ac : 6, bd : 4, bc : 6, ad : 6\}$ have been found so far in the mining process. We can set ξ_2 to 5 since the support of the N-th most frequent 2-itemset so far, cd, is 5.

In the mining phase, we set the initial threshold value of ξ to be zero. During mining, we increase ξ by assigning to it the minimum value among the supports of the N-th most frequent k-itemset discovered so far for $1 \leq k \leq k_{max}$:

$$\xi = min(\xi_1, \xi_2, \ldots, \xi_{k_{max}}) \tag{1}$$

We shall see later that as the threshold becomes greater, the pruning power will also be greater.

Lemma 1 *At the end of the first for loop of NFP-mine() (see Line (1) in Figure 4), if there are at least N different items in the set of k-itemsets discovered so far, then for $j < k$, $\xi_j \geq \xi_k$.*

Proof: If $\xi_k = 0$, it is trivial. Consider $\xi_k > 0$. Suppose the N-most frequent k-itemsets contains N different items, let the itemsets be $I_1, I_2..., I_N$. From I_1 we can form k subsets of $(k-1)$-itemsets. In I_i, if it contains some items not contained in $I_1, ..., I_{i-1}$, then if there is only one such item x, we can form $k-1$ subsets with $k-1$ elements each, each subset formed by removing each of the items not equal to x. These subsets are different from all the $(k-1)$-itemsets generated from $I_1, ..., I_{i-1}$. If I_i contains more than one items which are not in $I_1, ..., I_{i-1}$, one can form k subsets with $k-1$ elements by removing any item from I_i. When we finish the subset formation from $I_1, \ldots, I_N$, we can get at least N $(k-1)$-itemsets, and their supports are $\geq \xi_k$. We can repeat the argument with smaller subsets until we come to the 1-itemsets. In the FP-tree mining process, the subsets are generated either earlier or at the same iteration as the generation of the k-itemset. Therefore, for $j < k$, $\xi_j \geq \xi_k$. □

Corollary 1 *At the end of the first for loop of NFP-mine(), if there are at least N different items in the set of k_{max}-itemsets discovered so far, then $\xi = \xi_{k_{max}}$.*

The above is true since $\xi = min(\xi_1, \xi_2, \ldots, \xi_{k_{max}})$ and Lemma 1 implies that for $j < k_{max}$, $\xi_j \geq \xi_{k_{max}}$. Therefore, if the condition in the Corollary is satisfied, then instead of updating ξ as $min(\xi_1, \xi_2, \ldots, \xi_{k_{max}})$, we can update ξ only when $\xi_{k_{max}}$ is updated. The pruning effect will be unchanged, but less work is spent on the update of ξ and ξ_i.

3.1.1 Pruning conditional FP-trees

Based on the fact that an element of the header table cannot form a conditional pattern tree that consists of itemsets having supports greater than the support of this element, we have the following pruning step for BOMO: When forming the conditional FP-tree for an element, α, in the header table, we compare the support of α, which is equal to the total sum of the counts in the horizontal link of α, with ξ. If it is smaller than ξ, we can stop forming the conditional FP-trees for all the elements starting from α to the bottom element of the header table.

3.1.2 Pre-evaluation step

To enhance the performance of BOMO, we use a **pre-evaluation** step to evaluate a better initial lower bound for ξ and ξ_k, $2 \leq k \leq k_{max}$. We assign an array, C_k, of size N for ξ_k, for $2 \leq k \leq k_{max}$. Consider the FP-tree built from NFP-build(), for each path of length k from the root node, we examine the k-th node (last node in the path). Among all such k-th nodes, we pick the N largest values of the counts stored in the nodes, and these values are entered into C_k. For example, in the FP-tree in Figure 1, consider $k = 3$, and $N = 2$. Nodes labeled a:2 and b:1 are the only third elements at all the paths leading from the root node, and their counts are kept in C_3, namely, the counts of 2 and 1. Then we assign ξ_k to the N-th largest count value stored in C_k. Consider the example in Figure 1, with $N = 1$, and $k_{max} = 3$. We get two paths root$(NULL) \rightarrow (c : 3)$, root$(NULL) \rightarrow (e : 1)$ for k=1; one path root$(NULL) \rightarrow (c : 3) \rightarrow (d : 3)$ for k=2; two paths root$(NULL) \rightarrow (c : 3) \rightarrow (d : 3) \rightarrow (a : 2)$, root$(NULL) \rightarrow (c : 3) \rightarrow (d : 3) \rightarrow (b : 1)$ for k=3. Therefore the N-th largest values stored in C_1, C_2, C_3 are 3, 3 and 2, respectively, and we initialize ξ_1, ξ_2, ξ_3 to be 3, 3 and 2, respectively. Using Equation (1) we can determine ξ. We use these initial lower bounds for our mining phase. From experiment, we found that the improvement of this pre-evaluation step can be up to 10% of the total execution time.

3.1.3 Construction Order of Conditional FP-trees

We also study the starting position when processing the frequent-item header table (Line (2) of Figure 4). The ordering can have significant impact since the thresholds ξ, ξ_k, $1 \leq k \leq k_{max}$, are updated dynamically and if we encounter more suitable itemsets with high support counts earlier, we can set the thresholds better and increase the subsequent pruning power.

Top-down: One possible choice is to start from the top of the header table and go down the table to form conditional pattern base for each item in the table. For example, in Figure 1, the ordering will be {c, d, a, b, e}. This is based on the observation that frequent items are most likely located at the top levels of the FP-tree and hence we can prune the less

Algorithm : BOMO algorithm

Input: D, k_{max}, N

Output: N-most interesting k-itemsets for $1 \leq k \leq k_{max}$.

(1) Let $result_k$ be the resulting set of interesting k-itemsets. Set $result_k = \emptyset$.

(2) Scan the transaction database, D. Find the support of each item.

(3) Sort the items by their supports in descending order, denote it as *sorted-list*; determine ξ_1.

(4) $\xi_2 = \xi_3 = \cdots = \xi_{k_{max}} = \xi = 0$

(5) Create a FP-tree, T, with only a root node with label being "NULL".

(6) NFP-build(T, D, *sorted-list*, ξ).

(7) NFP-mine(T, $\emptyset$, ξ, ξ_1, ξ_2, ..., $\xi_{k_{max}}$)

Figure 2: BOMO algorithm for mining N-most interesting itemsets.

Algorithm : BOMO Tree-Building Phase

NFP-build(T, D, sorted-list, ξ)

(1) *selected-list* $\leftarrow$ sorted 1-itemsets list whose supports $\geq \xi$.

(2) Update the FP-tree as follows:

For each transaction, $Trans$ in D

If $Trans$ consists of some items which are in *selected-list*

Select and sort the items, which are in *selected-list*, in $Trans$ according to their supports.

Let $[i|I]$ be the sorted frequent item list in $Trans$, where i is the first element and

I is the remaining list. Invoke $Insert\_NFPtree([i|I], T)$.

Insert_NFPtree($[i|I]$, T)

(1) If T has a child C such that C.item-name = i.item-name,

then increment C's count by 1;

else create a new node C, and let its count be 1, its parent link be linked to T,

and its node-link be linked to nodes with the same item-name via the node-link structure.

(2) If I is non-empty, invoke $Insert\_NFPtree(I, C)$.

Figure 3: BOMO Tree Construction.

Algorithm : BOMO Mining Phase

NFP-mine($Tree$, α, ξ, ξ_1, ξ_2, ..., $\xi_{k_{max}}$)

{

If $Tree$ contains a single path P

(1) then for each combination, β, of the nodes in the path P do

(a) generate itemset $\beta \cup \alpha$ with $support = minimum\ support$ of nodes in β

(b) if $\xi_{|\beta\cup\alpha|} \leq support$

then

insert $\beta \cup \alpha$ to $result_{|\beta\cup\alpha|}$; update $\xi_{|\beta\cup\alpha|}$; update ξ if necessary

if $result_{|\beta\cup\alpha|}$ contains itemset X with support $< \xi_{|\beta\cup\alpha|}$ then remove X from $result_{|\beta\cup\alpha|}$;

(2) else for each a_i in the header of $Tree$ do

(c) generate itemset $\beta = a_i \cup \alpha$ with $support = a_i.support$

(d) construct β's conditional pattern base using ξ and

then β's conditional FP-tree $Tree_\beta$

(e) if $Tree_\beta \neq \emptyset$ then NFP-mine($Tree_\beta$, β, ξ, ξ_1, ξ_2, ..., $\xi_{k_{max}}$)

}

Figure 4: Algorithm for mining phase of N-most interesting itemsets.

frequent itemsets by finding the more frequent itemsets first. However, itemsets of larger sizes are usually distributed widely throughout the tree, and as a result, with this ordering, the increase rate of ξ_k for large k is slow while that for small k is fast. According to Equation (1), ξ is also increased slowly. This can lead to a large number of large conditional FP-trees.

Bottom-up: The other extreme is to go from the bottom of the header table upwards to the top. For example, in Figure 1, the ordering will be {e, b, a, d, c}. However, the bottom items have the smallest supports and the corresponding itemsets discovered will also have small supports, and hence the pruning power is also small.

Starting from the middle: We have tried different starting positions and scanning orders of the header table in our experiments, and find that starting at the middle of the header table; then from the middle item going upwards to the top and then from the (middle+1) item down to the bottom of the table, is a good choice. For example, in Figure 1, the ordering will be {a, d, c, b, e}. The reason is that the increase rate of ξ is faster and less conditional FP-trees are formed. This ordering will be used in all of our experiments discussed in Section 5. We found that it can achieve 1% to 10% improvement in time efficiency.

3.2 A Loop-back approach, LOOPBACK

Compared to the original FP-tree algorithm in [9], BOMO requires building an initial FP-tree with all items, while the original method may build an initial FP-tree for a subset of the items. Hence we may consider the possibility of building a smaller initial FP-tree. In order to do so, we should have an initial support threshold $\xi > 0$. Here we suggest such an approach, the initial value of ξ is determined by the smallest support among the N most frequent 1-itemsets. Let us consider the example in Section 2, with $N = 3$. Since the support of the third largest 1-itemset is 2, the threshold ξ is set to 2. We use the tree building algorithm to build the FP-tree. Therefore, we get c, d, a, b, and e as the large 1-itemsets. The constructed FP-tree is the same as that in Figure 1.

Using the previous example to illustrate the mining mechanism, we start from the top of the header table and invoke NFP-mine($Tree_{init}$, $NULL$, ξ, ξ_1, ξ_2, ..., $\xi_{k_{max}}$). Assume k_{max} = 3, i.e., we find large itemsets up to size $k = 3$, the resulting large itemsets once we execute NFP-mine() are: k=1: $\langle c:3, d:3, a:2, b:2, e:2\rangle$; k=2: $\langle cd:3, ac:2, ad:2, bc:2, bd:2\rangle$; k=3: $\langle acd:2, bcd:2\rangle$.

Up to this point, there may be cases that the number of k-itemsets, where $k \geq 2$, is less than N because the threshold, ξ, found in the building phase is not small enough. For the above example, there are only two large 3-itemsets found. There are not enough 3-itemsets, since $N = 3$. Therefore, a smaller ξ should be used in order to mine more itemsets in the mining phase. However, this method is exhaustive and we use a loop-back method to handle the problem. As long as there are not enough itemsets for certain level(s), we decrease ξ by a factor f, $0 < f < 1$, such that $\xi_{new} = \xi \times f$. Let us call the original ξ value ξ_{old}. We call NFP-build() to update the FP-tree in an incremental manner (see the discussion of incremental tree building below). Then we can call NFP-mine() to mine itemsets of supports $\geq \xi_{new}$. This is our basic idea for LOOPBACK.

After we have sorted the supports of items we can find the N-most interesting 1-itemsets. We initialize the support threshold value, ξ. For LOOPBACK approach, we set ξ to be the support of the N-th sorted largest 1-itemset.[2]

[2]If $k \geq N$, we are sure that using only the large 1-itemsets for the initial FP-tree is not enough to form itemsets of size $\geq N$. Therefore, we should choose the support of the $(k+1)$-th 1-itemset as the threshold during the building phase of the initial round.

3.2.1 Some Pruning Considerations

We can use an incremental approach to build or update the FP-tree. For each loop back (round), there is no need to rebuild the whole tree using the building phase. We only need to consider transactions which contain any newly added 1-itemsets (itemsets with support smaller than ξ_{old} in the previous round, but larger than or equal to ξ_{new} in the current round) in each round. We insert new branches to or modify the counts of existing branches of the FP-tree built in the previous round.

Skipping k-itemsets: In each round of loop-back, we keep itemsets which have been found so far from previous rounds, and use both ξ_{old} and ξ_{new} to filter the itemsets found during the mining phase. We do not need to generate itemsets whose supports $\geq \xi_{old}$, because they have already been found and kept in previous rounds.

Each time we loop back and redo NFP-mine(), we do not need to consider any more k-itemsets if we have already found the N or more number of itemsets of size k because we have found N-most interesting k-itemsets which have supports $\geq \xi_{old}$. If we continue to consider k-itemsets we shall only discover k-itemsets of smaller support values.

Lemma 2 *If N or more k-itemsets are found in a current round, then the N-most interesting k-itemsets are found.*

Skipping old items: In the mining phase, we do not need to consider all the items in the header table. We only need to consider items which are newly added to the header table in the current round as well as old items which were the base items of the conditional pattern bases for some itemsets having supports $< \xi_{old}$ in the previous round.

3.3 Generalization: Varying thresholds N_k for k-itemsets

In the previous consideration, we fix a number N on the resulting number of itemsets for itemsets of all sizes considered. However, in general frequent itemsets will be more numerous for smaller itemsets and less so for itemsets of greater size. It would be more flexible if we allow the user to specify possibly different numbers, N_k, of resulting k-itemsets for different values of k. This is a generalization of the original problem definition. With the generalization, we need to modify the two algorithms we proposed before. However, the change is very minor. We only have to change the meaning of ξ_k, $1 \leq k \leq k_{max}$. ξ_k will be the support of the N_k-th most frequent k-itemset discovered so far, $\xi_k = 0$ if the number of k-itemsets discovered so far is less than N_k. The other parts of the algorithms remain intact.

4 Mining with Item Constraints

If we mine the N-most interesting itemsets, though we have removed the requirement of support threshold, the uniform threshold on the number of itemsets is still a restriction. As pointed out in [25], a more flexible set up is to allow users to set different thresholds for different items or itemsets. An example given in [25] is that in a supermarket scenario, the itemset {bread, milk} is usually much more frequent than the itemset {food processor, pan}. However, the latter is a valuable itemset even though the occurrence is less frequent. Here we aim at achieving this flexibility.

Consider a set of items, I, partitioned into several bins $B_1, B_2, \ldots, B_m$ where each bin B_i contains a set of items in I. We define **item constraint** in a way similar to the support constraint defined in [25] and [17], however, instead of a constraint by support, we set a constraint by the number of itemsets. Item constraint IC_i has the form: $IC_i(B_{i_1}, \ldots, B_{i_s}) = N_i$, where $s \geq 0$, and B_{i_j} and B_{i_l} may be equal. It specifies what particular items or groups of items should or should not be present in the pattern [25].

We adopt the concepts of open and closed interpretations in [25]. An itemset (or pattern) I **matches** a constraint IC_i in the **open interpretation** if I contains at least one item from each bin in IC_i and these items are distinct. An itemset I matches a constraint IC_i in the **closed interpretation** if I contains exactly one item from each bin in IC_i and these items are distinct, I does not contain any other items.

With open interpretation, consider the set X of all k-itemsets containing at least one item from each B_{i_j} for a given IC_i, for $|IC_i| \leq k \leq k_{max}$, where k_{max} is a user defined parameter for the maximum size of an itemset to be mined. For each such value of k, we sort the k-itemsets according to their supports in descending order. Let the N_i-th greatest support be ξ, then we say that all k-itemsets in X with support not less than ξ are **interesting** for the constraint, for each $|IC_i| \leq k \leq k_{max}$. Note that the same threshold value, N_i, is used for all possible values of k. We call these itemsets the **N_i-most interesting itemsets for IC_i in the open interpretation**.

For closed interpretation, consider the set X of all k-itemsets containing exactly one item from each B_{i_j} for a given IC_i, therefore the size of these itemsets is $k = |IC_i|$. We sort these k-itemsets according to their supports in descending order. Let the N_i-th greatest support be ξ, then we say that all k-itemsets in X with support not less than ξ are **interesting** for the constraint. We call these itemsets the **N_i-most interesting itemsets for IC_i in the closed interpretation**.

Given IC_i, and N_i for $1 \leq i \leq c_{max}$, where c_{max} is the number of item constraints, we would like to find the N_i-most interesting itemsets for each IC_i, in either the open interpretation, or the closed interpretation. Note that k_{max} is also necessary for the open interpretation.

Bin	Items
B_1	a
B_2	b, c
B_3	d, e
B_4	f

Constraint	Bins	N_i
IC_1	B_1, B_2	N_1
IC_2	B_3, B_4	N_2
IC_3	B_2	N_3
IC_4	B_1, B_2, B_3	N_4
IC_5	B_1, B_2, B_4	N_5
IC_6	B_1, B_2, B_3, B_4	N_6

Figure 5: Partition of the set of items into different bins and the item constraints.

Suppose we partition a set of items into four bins as shown in the first table in Figure 5, and specify the constraints in the second table in the same figure. Consider itemsets $I_1 = (acd)$, $I_2 = (abf)$, $I_3 = (bc)$, the corresponding **bin patterns** for I_1, I_2, I_3 are (B_1, B_2, B_3), (B_1, B_2, B_4), and (B_2, B_2) respectively. We say that I_1 **matches** IC_1, IC_3, and IC_4 in the open interpretation, and matches only IC_4 in the closed interpretation; I_2 matches IC_1, IC_3, and IC_5 in the open interpretation, and matches IC_5 in the closed interpretation; I_3 matches IC_3 in the open interpretation, but matches none of the constraints in the closed interpretation.

In this section we propose two algorithms for mining N-most interesting itemsets with item constraints. The first algorithm is a straightforward modification of the BOMO algorithm. The second algorithm improves on the first one by maintaining the constraints information with a second FP-tree. The reasons for choosing BOMO instead of LOOPBACK are that BOMO outperforms LOOPBACK in most cases in our empirical study.

4.1 Single FP-tree Approach

With the closed interpretation, an itemset that matches a given constraint IC_i must be of size $|IC_i|$ since it contains exactly one element from each bin in IC_i. Therefore, we need only one dynamic support threshold ξ_i for such itemsets. The value of ξ_i is set to be the support of the itemset that matches IC_i which has the N_i-th highest support among all such itemsets that match IC_i discovered so far. We also set a global threshold,

$$\xi = min(\xi_1, \xi_2, \ldots, \xi_{c_{max}}) \tag{2}$$

With the open interpretation, an itemset that matches IC_i can have size equal to or greater than $|IC_i|$. Therefore the itemset size ranges from $|IC_i|$ to k_{max}. We assume that k_{max} is greater than the size of all item constraints $|IC_i|$. For each constraint IC_i and each possible itemset size k, we use a support threshold ξ_{ik} for pruning. The value of ξ_{ik} is set to be the support of the k-itemsets that matches IC_i which has the N_i-th highest support among all such k-itemsets that match IC_i discovered so far. Any newly encountered k-itemset that satisfies IC_i but is smaller than ξ_{ik} is not considered interesting. ξ_{ik} is initialized to zero, for all possible values of i and k. We also set a global threshold,

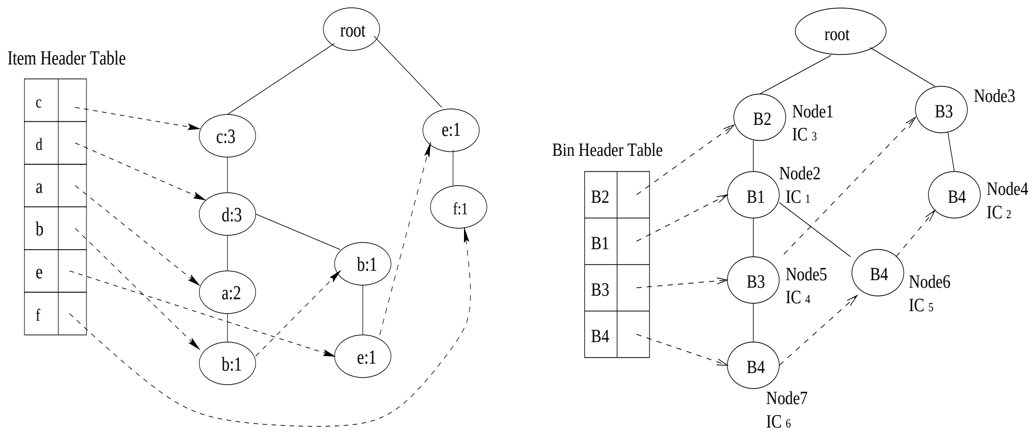
$$\xi = min(\text{values of } \xi_{ik} \text{ for all possible } i \text{ and } k) \tag{3}$$

We use the BOMO algorithm as the basic architecture and apply a simple constraint matching mechanism. We build an FP-tree for all the items in the transactions and mine for interesting itemsets based on the BOMO algorithm. ξ is used as the support threshold for building conditional FP-trees. With the closed interpretation, an itemset, I, having support $\geq \xi_i$ is to be matched with IC_i. If there is a match, we add I to the current result set and update ξ_i. With the open interpretation, if $|IC_i| \leq k \leq k_{max}$, then an k-itemset, I, having support $\geq \xi_{ik}$ is to be matched with IC_i. If there is a match, we add I to the current result set and update ξ_{ik} if necessary.

4.2 Double FP-trees Approach

If the number of constraints is large, the single FP-tree approach will consume a lot of computation and storage in the matching process. In the second approach, we try to use a compact data structure to store the constraints. We propose to employ an FP-tree for storing the set of constraints since it is a highly compact structure. We observe three advantages in doing this: (1) The FP-tree is usually smaller than the original constraint set [9] since constraints may share common bins. (2) We can perform the matching of itemsets and constraints in a more efficient way. (3) We can also have a better support pruning strategy. We propose to employ a special order of scan on the FP-tree storing the constraints so as to speed up the pushing of the dynamic minimum support thresholds as well as using a pruning strategy to tighten the threshold values.

4.2.1 Closed Interpretation



(a) Transaction FP-tree (b) Constraint FP-tree

Figure 6: Double FP-trees.

We build an FP-tree, **transaction FP-tree**, for the items in the transactions. We can also build another FP-tree, **constraint FP-tree**, for the bins in the constraints.

Table 1: Elements in a constraint FP-tree node.

Notation	Description
$n.c$	$n.c = i$ if the constraint IC_i is represented by the bins from the root to node n.
$\xi(n)$	If $n.c = i$, $\xi(n)$ is the N_i-th largest support of itemsets matching IC_i so far.
$\xi_{min}(n)$	The minimum value of ξ among the nodes of the subtrees rooted by a node n.
node-link	The next node in the FP-tree carrying the same bin, or null if there is none.

Figure 6 shows the FP-trees constructed from the database given in Section 2 and the constraints given in Figure 5. The node structure of a constraint FP-tree is shown in Table 1, which is different from the node structure of a transaction FP-tree described in Section 2.

In constructing the constraint tree, we first scan the constraint set and find the support for each bin. Then, we create an empty constraint FP-tree with root = "NULL". For each constraint, IC_i, the bins in the header table of the tree are sorted in descending order of their supports. We insert all the bins in the constraint as a path to the tree in a way similar to the transaction FP-tree, except we do not need to worry about the counts since there will not be any repeated constraints. Common prefixes between patterns of different constraints share the same tree path, otherwise we create a path with new nodes. For the node, $Node$, representing the last bin of the pattern IC_i, we set $Node.c$ to i denoting IC_i. We say that $Node$ **corresponds to** constraint IC_i. If no such constraint can be determined for $Node$, then $Node.c$=0, indicating that it does not correspond to any constraint. For each node $Node$, $\xi_{min}(Node)$ is initialized to zero. If $Node.c$=0, then we set $\xi(Node) = MAXINT$, where $MAXINT$ is a large number greater than the number of transactions in the given database. This will indicate that the node does not get involved in setting the overall global threshold ξ. If $Node.c \neq 0$, then we set $\xi(Node) = 0$ as an initial value.

Observation 2 *A bin pattern formed by the bins from the root to any node of the constraint FP-tree can match at most one constraint in the closed interpretation.*

Using the previous example, there is no constraint matching the pattern represented by the bins from the root to its right child $Node_3$ representing the bin pattern (B_3), therefore the element $Node_3.c$ is undefined (zero). On the other hand, element $Node_2.c = 1$ because the pattern (B_2, B_1) represented by the root node down to $Node_2$ matches $IC_1(B_1, B_2)$.

Overall strategy: The basic idea of our Double FP-trees approach is that we visit each item a in the header table of the conditional transaction FP-tree, T, of a base pattern α, and form the conditional transaction FP-tree for the base pattern $a \cup \alpha$ using the minimum support threshold from the constraint FP-tree, T_{IC}. Note that the initial transaction tree can be treated as a conditional transaction FP-tree with $\alpha = \phi$.

Update of ξ_{min} at the root node: This minimum support threshold can be set to ξ_{min} of the root of T_{IC} and is exactly equal to ξ in Equation 2. This minimum support threshold can be increased during the mining process, whenever we find an itemset interesting for a constraint IC_i, by updating ξ of the corresponding constraint FP-tree node and pushing the value of ξ_{min} to the ancestors of this node till the root.

We perform the above steps recursively until a single path conditional FP-tree is obtained for a base pattern α. Then we generate each possible itemset combination from the single path SP and try matching the itemset with the constraints in the constraint FP-tree in a top-down manner. Matching is done by the following steps: (1) Translate the itemset to the corresponding bin patterns. (2) Sort the bin patterns according to the ordering at the bin header table. (3) Try matching the sorted bin patterns to any path in the constraint FP-tree.

Example: Let the base pattern be $\alpha = \alpha_1\alpha_2...\alpha_e$. Suppose the single path SP is $root(NULL) \rightarrow \beta_1 \rightarrow \beta_2 \ldots \rightarrow \beta_n$. Then we generate each possible itemset combination ($2^n - 1$ combinations) from SP and try matching the itemset with the constraints in the constraint FP-tree in a top-down manner. Suppose we have m bins, $B_1, B_2, \ldots, B_m$. Let $B_{b1}B_{b2}\ldots B_{bn}B_{b(n+1)}\ldots B_{b(n+e)}$ be the bin pattern of the itemset $\beta_1\beta_2\ldots\beta_n\alpha_1\alpha_2\ldots\alpha_e$. We sort the bin pattern according to the top-down order in the constraint header table, and get $B_{s1}B_{s2}\ldots B_{s(j+e)}$. At each tree level l (level of root is zero), there is at most one node matching the l-th bin (B_{sl}) in the pattern. If a matching is found, we match the next bin ($B_{s(l+1)}$) with the children of this node. We say that the itemset matches a constraint IC_i if its sorted bin pattern matches exactly a tree path $root(NULL) \rightarrow B_{t1} \rightarrow B_{t2} \ldots \rightarrow B_{t(j+e)}$ (i.e. $B_{s1} = B_{t1}, \ldots B_{s(j+e)} = B_{t(j+e)}$), and the constraint represented by the bottom matched node of this path is IC_i.

We include the itemset in the current resulting set if we find a match between the itemset and constraint IC_i, and the support of the itemset is not less than the N_i-th greatest support of the interesting itemsets (for IC_i) found so far. Figures 7 to 9 show the Double FP-trees algorithm.

Observation 3 *The value of ξ at the root of T_{IC} is equal to the value of ξ of Equation 2 at any point of the execution.*

Example: We illustrate the algorithm in more details using the previous running example from Figure 5. We set $N_1 = N_2 = \ldots = N_6 = 1$, and $k_{max} = 4$. We build the transaction FP-tree and the constraint FP-tree according to the supports of items and bins respectively using a function similar to $NFP\text{-}build$ in BOMO. Assume we start from the bottom of the header table of the FP-tree, T, in Figure 6(a), we visit the item f first. The corresponding bin pattern of $f \cup \alpha$, where α ($= \emptyset$) is the base pattern for T, is (B_4). Since ξ_{min} (=0) of the root of the constraint tree, T_{IC}, is not greater than the support of $f \cup \alpha$ (=1), we try to match the bin pattern B_4 with T_{IC} using function $Mapping$. There is no match between B_4 and the child nodes of the root of T_{IC}, so we discard $f \cup \alpha$. Next we form the conditional FP-tree of f using threshold = ξ_{min} (=0) of root of T_{IC}. We get $(fe : 1)$, which form the sorted bin pattern (B_3, B_4) and it is interesting for IC_2. So we update $\xi_{min}(Node_4)$ = $\xi(Node_4)$ = support of fe = 1, and push the value of ξ_{min} upwards to the ancestors of $Node_4$ using function $Update\_T_{IC}$.

Next we consider item e in T, the corresponding bin pattern of $e \cup \alpha$ is (B_3). Since support of e (=2) is greater than ξ_{min} of root of T_{IC}, we try matching $e \cup \alpha$ with T_{IC}. Although $Node_3$ matches B_3, $Node_3.c$ is undefined, therefore we discard $e \cup \alpha$. Then we consider the conditional FP-tree of e, which is a single path tree, $root(NULL) \rightarrow (c : 1) \rightarrow (d : 1) \rightarrow (b : 1)$. We get the possible itemset combinations: $(eb : 1), (ebd : 1), (ebdc :$

$1), (ed : 1), (edc : 1), (ec : 1)$, and the corresponding bin patterns: (B_2, B_3), (B_2, B_3, B_3), (B_2, B_2, B_3, B_3), (B_3, B_3), (B_2, B_3, B_3), (B_2, B_3). Since none of them matches T_{IC}, they are discarded.

Similarly we can process the items b, a, d and c in the item header table. □

Algorithm : DFP-tree algorithm

Input: D, $B_1, B_2, \ldots, B_{b_{max}}$, $IC = \{IC_1, IC_2, \ldots, IC_{c_{max}}\}$, $N_1, N_2, \ldots, N_{c_{max}}$

Output: N_i-most interesting itemsets for each IC_i where $1 \leq i \leq c_{max}$.

(1) Let $result_{ik}$ be the resulting set of interesting k-itemsets matching constraint IC_i.
  Set $result_{ik} = \emptyset$.
(2) Scan the transaction database, D. Find the support of each item.
(3) Sort the items by their supports in descending order, denote it as *sorted-item-list*.
(4) Build a transaction FP-tree, T, with all the items.
(5) Scan the constraints, IC. Find the support of each bin.
(6) Sort the bins by their supports in descending order, denote it as *sorted-bin-list*.
(7) Create a constraint FP-tree, T_{IC}, with *sorted-bin-list* and IC.
(8) DFP-mine(T, T_{IC}, $\emptyset$)

Figure 7: DFP-tree algorithm for mining N-most interesting itemsets with constraints.

Order of Scan: Instead of visiting each item in the item header table sequentially from head to tail or from tail to head of the table for the transaction FP-tree, we employ a new ordering scheme. Starting from the bottom of the bin header table for the constraint FP-tree, for each bin B, we first build the conditional transaction FP-trees which consist of any item in the item header table that belongs to B. This bottom-up approach allows the values of $\xi(node)$ and $\xi_{min}(node)$ of the leaf nodes to be updated (increased) as soon as possible, and hence we can speed up pushing the nodes in upper levels of the constraint FP-tree. Since the propagation of ξ_{min} will increase the value of ξ, we can achieve better pruning power. Experiment shows that this ordering can achieve 6% improvement in time efficiency.

From the running example, the order of visiting the item header table in Figure 6(a) is $f \rightarrow e \rightarrow d \rightarrow a \rightarrow b \rightarrow c$. Therefore, after we have visited bin B_4, we can set ξ_{min} of $Node_4$, $Node_6$ and $Node_7$ to $MAXINT$ and invoke function $Update\_T_{IC}$ to push ξ_{min} to $Node_2$, $Node_3$, $Node_5$ and their ancestors if necessary.

Algorithm : DFP-tree Mining Phase

DFP-mine(T, T_{IC}, α)

{

If T contains a single path P

(1) then for each combination, β, of the nodes in the path P do

(a) generate itemset $\beta \cup \alpha$ with $support$ = $minimum\ support$ of nodes in β

(b) Mapping(T_{IC}, $\beta \cup \alpha$, $support$)

(2) else for each a_i in the header of T do

(c) generate itemset $\beta = a_i \cup \alpha$ with $support = a_i.support$

(d) $\xi \leftarrow \xi_{min}$(root of T_{IC})

(e) construct β's conditional pattern base using ξ and then β's conditional FP-tree T_β

(f) if $T_\beta \neq \emptyset$ then DFP-mine(T_β, T_{IC}, β)

}

Figure 8: Algorithm mining N-most interesting itemsets with constraints.

Algorithm : Mapping

Mapping(T_{IC}, γ, $support$)

{ Find the path P, if any, from T_{IC} that matches the bin pattern of γ

Let $P.lowest$ be the last node in P

If $P.lowest.c \neq 0$ and $\xi(P.lowest) \leq support$

(a) add γ to the result set, $result_{ck}$, where c is the matched constraint $IC_c = P.lowest.c$, k is the size of γ

(b) $\xi(P.lowest) \leftarrow$ the N_c-th largest support in $result_{ck}$

(c) Update_T_{IC}($P.lowest$)

}

Update_T_{IC}($node_{IC}$)

{ (1) $\xi_{min}(node_{IC}) \leftarrow$ min($\xi(node_{IC})$, min(ξ_{min} of all children of $node_{IC}$))

(2) If there is a change of value for $\xi_{min}(node_{IC})$ then invoke Update_T_{IC}(parent of $node_{IC}$)

}

Figure 9: Matching itemset with constraints in constraint FP-tree.

Pruning Strategy: When generating the conditional FP-tree of an item a, instead of setting the threshold to be ξ_{min} of the root of T_{IC} as shown in step 2(d) in Figure 8, we set the threshold ξ to be the minimum value of ξ_{min} among all the constraint FP-tree nodes belonging to the bin of a. This increases the pruning effect during mining. Experiment shows that this pruning can achieve 6% to 8% improvement in time efficiency.

Lemma 3 *An itemset containing an item a with support s is not interesting if s is less than the minimum value of ξ_{min} among all the constraint FP-tree nodes belonging to the bin of a in the closed interpretation.*

For example, the threshold for item f's conditional FP-tree is the minimum ξ_{min} among $Node_4$, $Node_6$ and $Node_7$ in Figure 6(b).

Corollary 2 *Given an itemset I with support s, let B_{lowest} be the last bin in the sorted bin pattern for I according to the constraint header table, I is not interesting if s is less than the minimum value of ξ_{min} among all the constraint FP-tree nodes belonging to B_{lowest}.*

Therefore we can further strengthen the pruning power of ξ in step 2(e) of Figure 8 by finding a suitable ξ_{min} from the items in β for building the conditional FP-trees of β.

4.2.2 Open Interpretation

We employ similar strategies for the open interpretation as for the close interpretation. There is some more complication since for each constraint IC_i, we need to consider the k-itemsets for $|IC_i| \leq k \leq k_{max}$. Therefore, at each node $Node$ in the constraint FP-tree, if it corresponds to a constraint IC_i, we keep a list of values ξ_{ik} for $|IC_i| \leq k \leq k_{max}$. Instead of at most one branch of the constraint FP-tree matching an itemset, there may be more than one branch matching an itemset in the open interpretation. As a result, the *Mapping* function in Figure 9 would take more time.

4.3 Maximum Support Thresholds

In this subsection, we consider the case of pruning interesting itemsets which have supports greater than a user specified **maximum support threshold** ξ_{max}. In other words, we would like to find the N_i-most interesting itemsets for each constraint IC, where the supports of these itemsets are less than the maximum support threshold ξ_{max}. The rationale of employing

ξ_{max} is that itemsets with very high supports can be trivial to the users. Suppose we have a database for maternity medical records at a local hospital, then all patients in the records will be female and almost all will be living in the same country. Hence the support of the itemset containing "female" and the country name will be very high. Such trivial frequent itemsets are typically not interesting. On the other hand, itemsets with smaller supports may be more interesting since users may not have the explicit knowledge about these itemsets.

A simple way to employ the maximum support thresholds with our algorithm is to discard itemsets of supports greater than ξ_{max} instead of inserting them into the current result. The other parts of the algorithm remain unchanged.

5 Performance Evaluation

We first compare the performance of BOMO and LOOPBACK with the Itemset-Loop algorithm [7] and a modified version of FP-tree algorithm for mining N-most interesting itemsets. Then we evaluate the Double FP-trees and Single FP-tree algorithms for constraint-based mining. All experiments are carried out on a SUN ULTRA 5_10 machine running SunOS 5.6 with 512MB Main Memory. Both synthetic and real datasets are used.

5.1 N-most Interesting Itemsets

Real Data: Three sets of real data are from the census of United States 1990 [10], and BMP-POS [26]. The first set is a small database **(tiny.dat)** of the census data with 77 different items and 5577 tuples, the second set is a large database **(small.dat)** of the census data with 77 different items and 57972 tuples. These are also the data set used for the previous work of algorithm Itemset-Loop [7]. The third one **(BMS-POS)** is a point-of-sales data set from a large electronics retailer, which contains 515597 tuples with 1657 different items.

Synthetic Data: Several sets of synthetic data are generated from the synthetic data generator in [5]. The generator follows the data generation method in [3] with the parameter setting and datasets shown in Table 2 and Table 3.

For each dataset (real or synthetic), we perform the experiment under different values of N in the N-most interesting itemsets. We vary N from 5, to 200, and k_{max} from 4 to 16. We compare the performance of our new approaches with the Itemset-Loop algorithm. We also evaluate the performance of the FP-tree algorithm when a known (optimal) threshold,

Table 2: Parameter setting.

Parameter	Description	Value
$\|D\|$	Number of transactions	100K, 1000K
$\|T\|$	Average size of the transactions	5, 10, 20
$\|I\|$	Average size of the maximal potentially Large itemsets	2, 4, 6, 8, 10
$\|L\|$	Number of maximal potentially large itemsets	2000, 10000
M	Number of items	1K, 50K
C	Correlation between patterns	0.25

Table 3: Synthetic data description.

Dataset	$\|T\|$	$\|I\|$	$\|D\|$	$\|M\|$
T5.I2.D100K	5	2	100K	1K
T20.I6.D100K	20	6	100K	1K
T20.I8.D100K	20	8	100K	1K
T20.I10.D100K	20	10	100K	1K
T10.I4.D1M	10	4	1000K	50K

ξ_{opt}, is given, i.e. a threshold which is just small enough to make sure that all the required N-most interesting k-itemsets are of supports greater than or equal to this threshold, this method does not need any loop-back. [3] Tables 4 and 5 show the ideal thresholds for different datasets.[4] We measure the total response time as the total CPU and I/O time used for both the building and the mining phases. Each data point plotted in the graphs is determined by the mean value of several runs of the experiment.

First, we compare the performance using different datasets. For each level k, we find the N most interesting itemsets. The threshold decrease rate, f, is set to 0.2. We find that

[3]It is highly unlikely that a real application can determine an optimal threshold. However, we use this setting as a kind of bound for optimal performance for comparison.

[4]The support values listed here are in terms of the percentage of the transactions that contain an itemset.

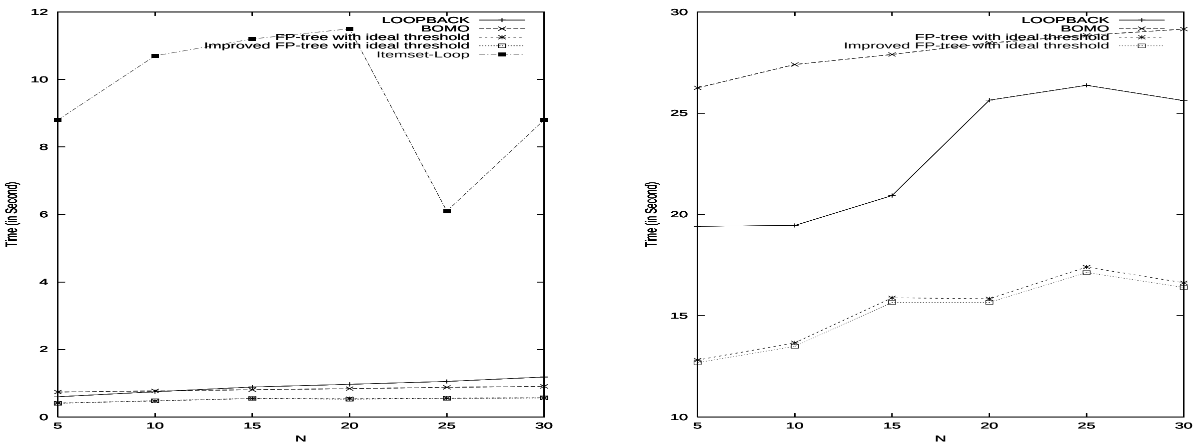
Table 4: Ideal thresholds, ξ_{opt} (%), for different datasets with $k_{max} = 4$.

Dataset	N					
	5	10	15	20	25	30
tiny.dat	9.79	7.02	6.36	6.06	5.73	5.48
small.dat	21.42	19.39	18.04	14.13	12.58	10.50
T5.I2.D100K	0.30	0.29	0.27	0.25	0.23	0.22

Table 5: Ideal thresholds, ξ_{opt} (%), for different datasets with $k_{max} = 10$.

Dataset	N					
	5	10	15	20	25	30
tiny.dat	0.23	0.21	0.19	0.17	0.17	0.17
small.dat	7.46	7.46	6.9	6.9	6.3	6.3
T20.I10.D100K	0.49	0.49	0.46	0.54	0.51	0.49

all our approaches outperform Itemset-Loop and have similar performance as the FP-tree algorithm with ideal threshold, ξ_{opt}, for real datasets. The execution times for Itemset-Loop in Figures 10 (b) and 11 are too large to be shown. For example, for tiny.dat with $N = 20$, $k_{max} = 10$, the execution time of Itemset-Loop is about 32000 sec.



(a) small.dat, $k_{max} = 4$

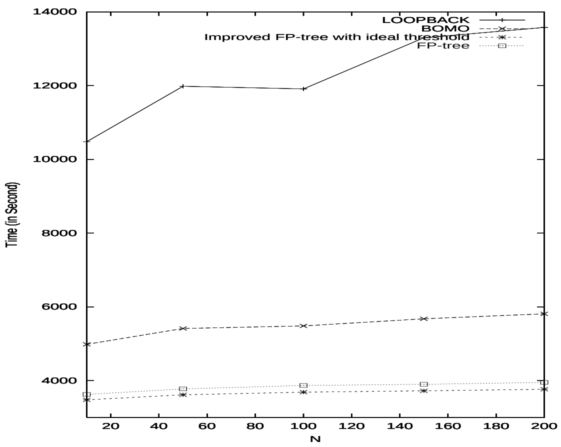
(b) tiny.dat, $k_{max} = 10$

Figure 10: Real datasets.

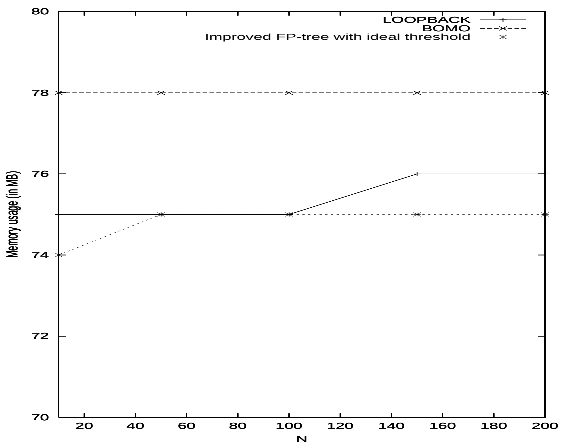
It may be expected that with the optimal threshold of ξ_{opt}, the original FP-tree algorithm [9] discussed in Section 2 should be the fastest because it does not require any loop back and it can build the smallest necessary initial FP-tree. However, in some cases we found that this method (denoted by **FP-tree with ideal threshold**) requires more total response time than our methods. The reason is that the number of k-itemsets of supports $\geq \xi_{opt}$ is too large for some values of k so that a large number of itemsets are generated. From experiment, we find that if ξ_{opt} is too small and N is large, then the original FP-tree algorithm does not perform well even with an ideal threshold. An example is the dataset T5.I2.D100K with $N = 20$, $k_{max} = 10$ and $\xi_{opt} = 0.002\%$, the execution time of FP-tree algorithm (180000 sec) is more than 100 times of that of our methods (about 1400 sec for LOOPBACK).

To improve the performance of the original FP-tree method, we use the set of thresholds, ξ_k, $1 \leq k \leq k_{max}$, and dynamically update ξ_k as in our NFP-tree algorithm for the pruning of small itemsets. We denote this method by **Improved FP-tree with ideal threshold**.

Figure 10(b) shows that LOOPBACK is faster than BOMO for the small real dataset. The whole mining process only requires a few number of loopbacks, and ξ_{opt} is large. Therefore, building a complete FP-tree in BOMO becomes an overhead relative to the small trees built in LOOPBACK, and the enhancements in BOMO are not significant. Figure 11b shows that BOMO requires more memory than others.



(a) execution time for $k_{max} = 10$

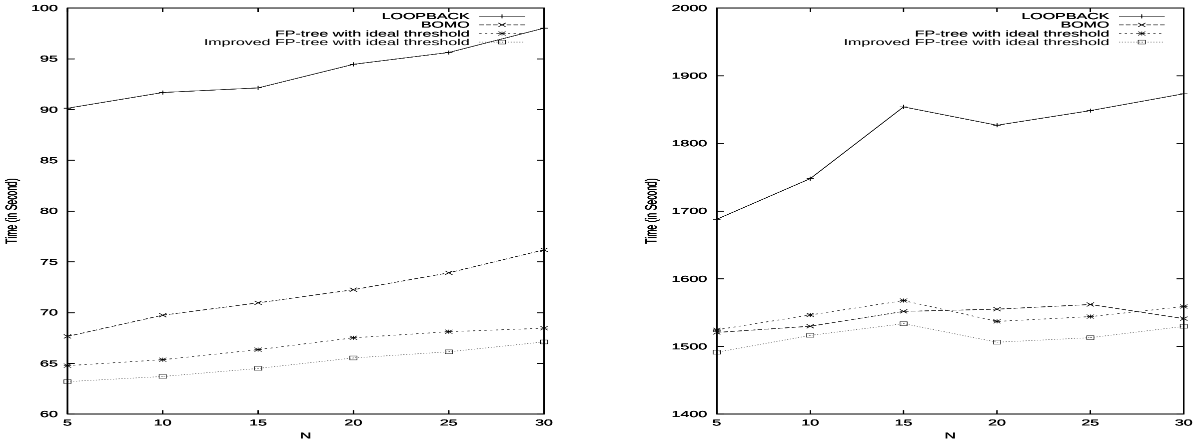


(b) memory usage for $k_{max} = 10$

Figure 11: Real dataset: BMP-POS.

Since the synthetic datasets are much larger, it requires a longer time to scan the database in the building phase of each loop. Also, the mining phase requires much more time than the building phase. The avoidance of redundant work in BOMO becomes significant. Therefore, LOOPBACK takes longer time to complete the mining. See Figures 12 to 14.

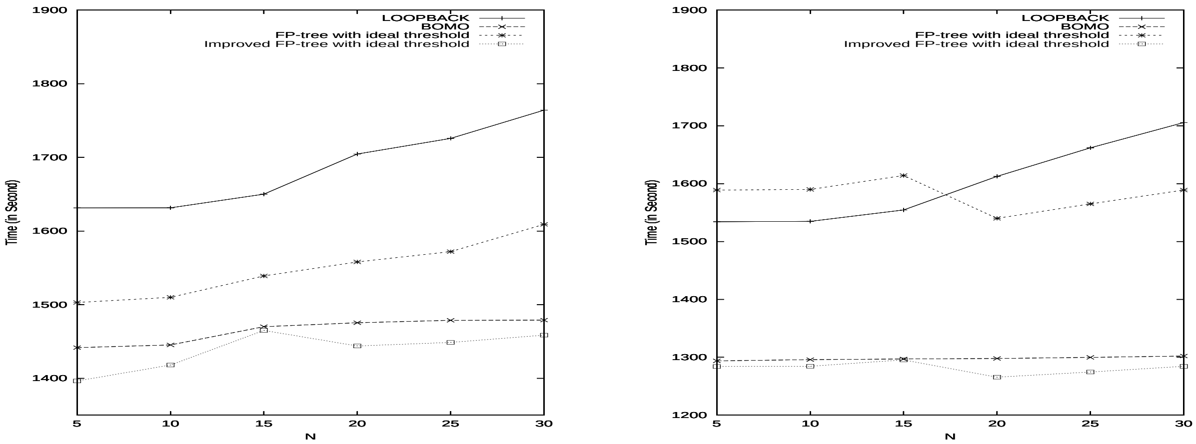
Among the new approaches, BOMO is the fastest and is comparable to the Improved FP-tree method. BOMO does not require any loopbacks and hence eliminate any redundant work in both tree building and mining. All experiments show that the main memory requirement for storing the complete FP-tree is less than 100MB, and the total memory usage for the largest dataset is around 300MB as shown in Figure 14. This shows that BOMO is a good choice for mining N-most interesting itemsets.



(a) T5.I2.D100K

(b) T20.I6.D100K

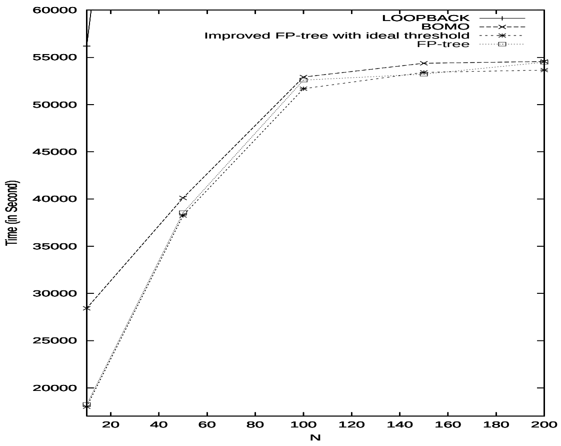
Figure 12: Synthetic datasets with $k_{max} = 4$.



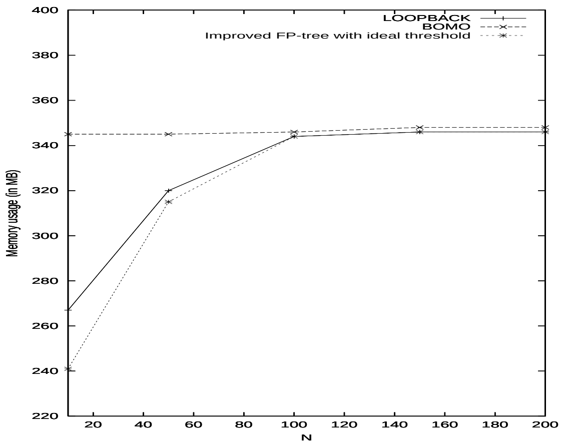
(a) T.20.I8.D100K

(b) T.20.I10.D100K

Figure 13: Synthetic datasets with $k_{max} = 10$.



(a) execution time for $k_{max} = 10$



(b) memory usage for $k_{max} = 10$

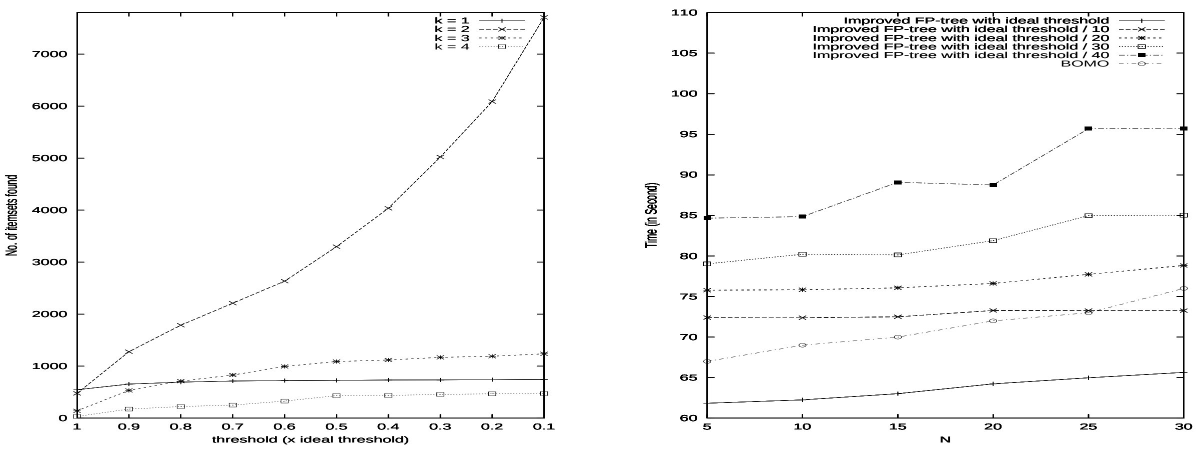
Figure 14: Synthetic dataset: T10.I4.D1M.

Generalization: Next, we test the performance of our algorithms under different N values for different values of k. We set N to be 30, 27, 24, ..., 3 for k = 1, 2, 3, ..., 10 respectively when $k_{max} = 10$. We set N to be 30, 20, 10, 5 for k = 1, 2, 3, 4 respectively when $k_{max} = 4$. Table 6 shows the execution times for the different methods. Again, BOMO is comparable to the improved FP-tree algorithm with an ideal threshold.

Table 6: Execution time (sec) with different N for different k.

Dataset	k_{max}	LOOPBACK	BOMO	Improved FP-tree with ideal threshold
tiny.dat	4	0.5	0.8	0.5
tiny.dat	10	2.7	2.3	1.7
small.dat	4	6.6	26.8	6.6
small.dat	10	25.6	29.2	7.3
T5.I2.D100K	4	96.4	70.4	60.5
T20.I6.D100K	4	1829.0	1505.6	1446.0
T20.I8.D100K	10	1763.5	1480.2	1378.4
T20.I10.D100K	10	1687.5	1332.0	1279.5

In real applications, it is generally very difficult to pick an optimal support threshold. If the guess is too large, then the conventional approach would not get the proper results, and therefore our approach is definitely much better. We evaluate the effect of guessing a non-optimal threshold that is too small for the FP-tree method. We decrease ξ_{opt} by different factors and use the resulting values as the non-optimal thresholds for the algorithm. Figure 15 (a) shows the increase in the number of itemsets for each size k and the total response time when the non-optimal threshold gets smaller. Figure 15 (b) shows that when user sets a non-optimal threshold, the BOMO algorithm can greatly outperform the improved FP-tree method.



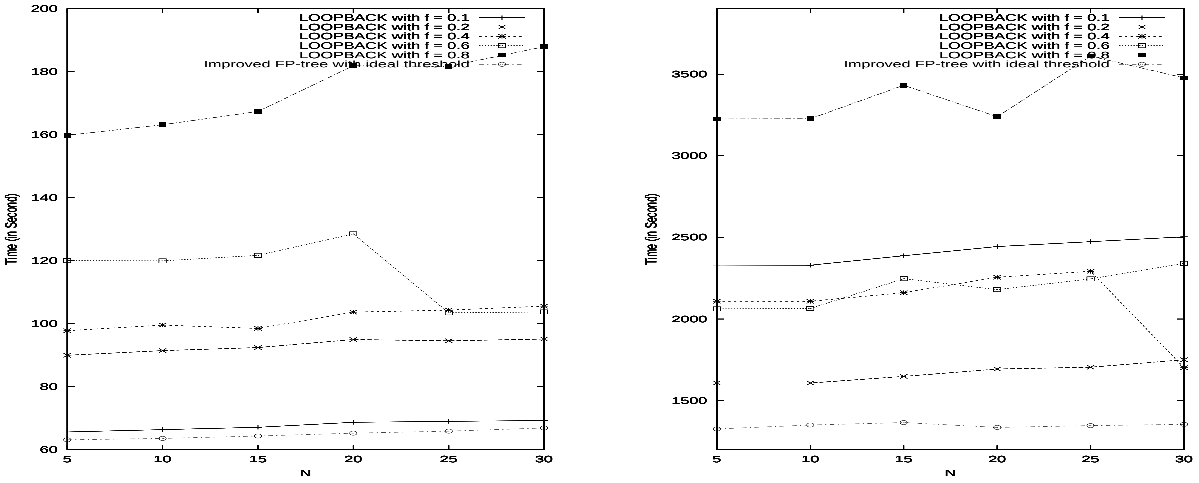
(a) number of itemsets for k_{max}=4, N=30

(b) response time for k_{max}=4

Figure 15: Effect of guessing small thresholds, synthetic dataset: T5.I2.D100K.

We test the use of different decrease factor, f, for LOOPBACK as shown in Figure 16. In each loop-back, we decrease the threshold by a certain factor (decrease factor) which is ranged from 0.1 to 0.8. In general, the smaller the f, the faster our algorithm is as the number of loop-back is reduced. However, if f becomes too small, the difference between the new and old thresholds (ξ_{new} and ξ_{old}) in each loop becomes large, the number of itemsets with supports that fall between these two thresholds increases and therefore the pruning effect of ξ becomes insignificant. This explains why there is an increase in execution time if f is too small as shown in Figure 16.

Near the end of Section 3.1, we discussed the different ordering of conditional FP-tree construction from the header table. There we propose an ordering starting from the middle. It is of interest to examine some other ordering also starting from the middle. Here we compare the one from Section 3.1, we call BOMO with an ordering of (middle, middle + 1, middle -1, middle+2, middle-2, ...) "BOMO with special order". The results are shown



(a) T5.I2.D100K with k_{max}=4

(b) T20.I8.D100K with k_{max}=10

Figure 16: Different decrease rate, synthetic datasets.

in Figure 17. The two orderings are comparable, and there is no significant winner. For all other experiments, we apply the ordering at Section 3.1

5.2 Constraint-based Mining

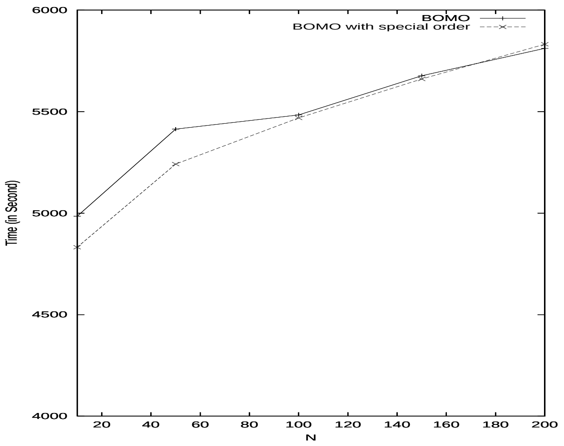
We make use of the effectiveness of BOMO for the implementation of the single and double FP-trees algorithms.

Real Data: The real census dataset provided in 1990 [22] is used. The dataset has 23 attributes, 63 items and 126229 transactions. This is the same set of data used in the previous work in [25]. To generate support specifications, we group the items from the same attribute into a bin, giving 23 bins $B_1, B_2, \ldots, B_{23}$ [25]. Let V_i be a bin variable. We specify constraint IC_i in the closed interpretation:

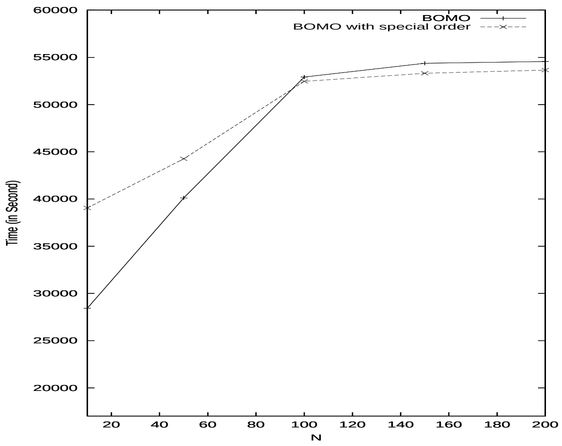
$$IC_i(V_1, V_2, \ldots, V_k) = N_i, \text{ where } 0 < k \leq k_{max}. \tag{4}$$

We specify N_i in two ways: (1) N_i is specified by user. (2) N_i = number of itemsets matching IC_i with supports $\geq \theta_i(V_1, V_2, \ldots, V_k)$, where $\theta_i(V_1, V_2, \ldots, V_k) = min(\gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k), 1)$, $\gamma > 1$, $S(V_j)$ is the smallest support of the items in the bin represented by V_j.

Synthetic Data: We use the same synthetic data generator as stated in Section 5.1. We set



(a) Real dataset: BMS-POS, with $k_{max} = 10$



(b) Synthetic dataset: T10.I4.D1M, with $k_{max} = 10$

Figure 17: Execution time for different orderings.

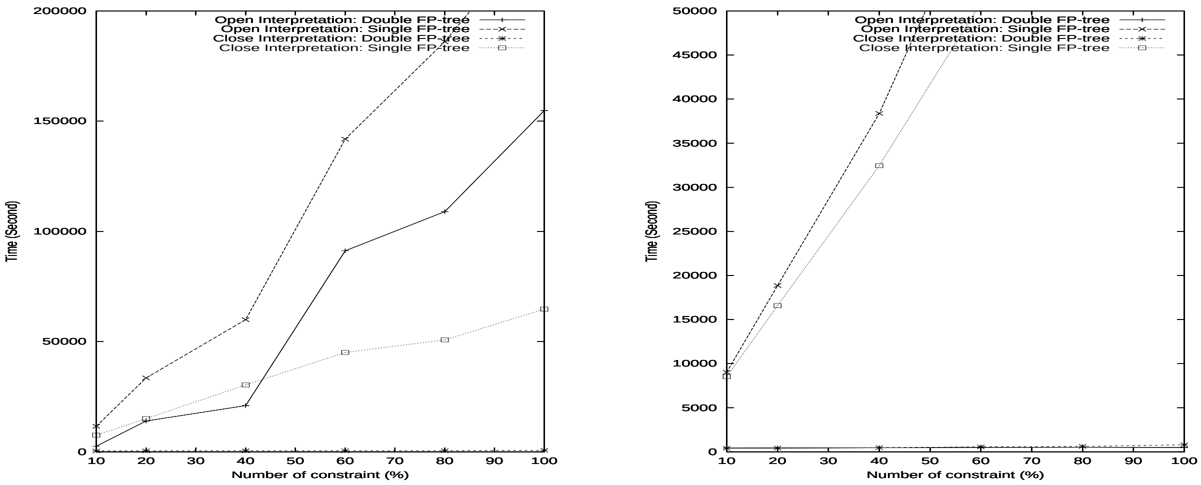
the number of transactions = 100K, number of items = 500, average length of transactions = 10, and the default setting for all other parameters. We partitioned the support range into 4 intervals such that B_i contains the items with support in the i-th interval and each B_i contains approximately the same number of items [24]: $\mid B_1 \mid = 122$, $\mid B_2 \mid = 122$, $\mid B_3 \mid = 122$, and $\mid B_4 \mid = 124$. Table 7 shows the constraints in the closed interpretation, where N_i is determined by Equation (4).

Table 7: Constraints for synthetic data in the closed interpretation.

C_i	N_i	C_i	N_i
$IC_1(B_1)$	122	$IC_9(B_2)$	122
$IC_2(B_1, B_2)$	14884	$IC_{10}(B_2, B_3)$	1233
$IC_3(B_1, B_2, B_3)$	453348	$IC_{11}(B_2, B_3, B_4)$	13209
$IC_4(B_1, B_2, B_3, B_4)$	36131	$IC_{12}(B_2, B_4)$	11372
$IC_5(B_1, B_2, B_4)$	120005	$IC_{13}(B_3)$	122
$IC_6(B_1, B_3)$	14884	$IC_{14}(B_3, B_4)$	8243
$IC_7(B_1, B_3, B_4)$	43492	$IC_{15}(B_4)$	124
$IC_8(B_1, B_4)$	13176		

We consider the real dataset. We vary the constraint set size by randomly choosing 10%, 20%, 40%, ..., 100% constraints out of the original set. We set N_i using Equation 4 with $\gamma = 5$, and $k_{max} = 5$. The results are shown in Figure 18(a). The Double FP-trees approach outperforms the Single FP-tree approach with a great margin.

As expected, the matching in the closed interpretation is faster than that in the open interpretation. An itemset can match at most one constraint in the closed interpretation while it can match several constraints in the open interpretation. Moreover, we can apply Lemma 3 for the closed interpretation, and we cannot do so for the open interpretation since any item in the base of a conditional FP-tree can both be included and excluded during matching.



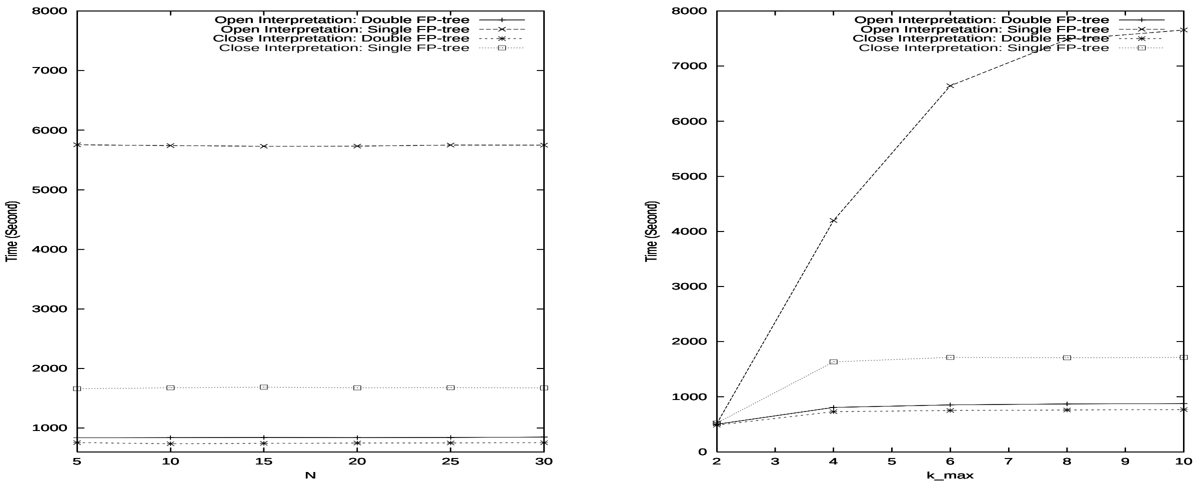
(a) Different N using Equation 4

(b) Uniform $N = 5$

Figure 18: Real dataset.

Next, we vary the size of the constraint set but have an uniform N for all the constraints. We set $N = 5$ for k_{max} up to 5. Again, Double FP-trees outperforms Single FP-tree. Figure 18(b) shows that the Double FP-trees approach requires only 400 – 700 seconds for the mining while the Single FP-tree approach may require more than 100000 seconds. From the result, we can see that the use of constraint FP-tree allows an efficient matching between itemsets and constraints. Moreover, we can build less number of conditional FP-trees with smaller sizes by making use of a local and larger ξ_{min}, instead of a global minimum ξ_{min} as in the Single FP-tree approach, for pruning when building conditional FP-trees.

With the synthetic data, we also test the performance against different N and k_{max} values. Figure 19(a) shows the result for N = 5, 10, ..., 30. Since the constraint set is much smaller than that of the real data, the mining time required for different N values of the



(a) Uniform N = 5, 10, ..., 30

(b) N = 5, different k_{max}

Figure 19: Synthetic dataset.

same approach does not vary much. However, using a tree structure for constraint matching benefits our Double FP-tree approach to a certain extent as shown in the figure.

Figure 19(b) shows the result of different k_{max}. We set k_{max} = 2, 4, ..., 10. The Double FP-trees method gives a close to linear performance in both interpretations.

In conclusion, the Double FP-tree algorithm is much more scalable than the exhaustive Single FP-tree method. It is highly effective no matter the number of constraints is large like that in the real dataset or small like that in the synthetic dataset.

Mining with Maximum Support Thresholds: We vary the maximum support threshold ξ_{max} in the following ways: (1) For real data, ξ_{max} = 20%, 30%, 40%, 50%, or 100%. (2) For synthetic data, ξ_{max} = 5%, 6%, 7%, 8%, 9%, or 100%. The cases where ξ_{max} = 100% are cases where we do not have any overhead in handling maximum support thresholds, as there is essentially no threshold. From experiment, we find that the performances of the Double FP-trees algorithm with different ξ_{max} are similar. For the real dataset with N_i specified by Equation 4, the execution time varies from 590 to 650 seconds. For the synthetic dataset with $N = 5$, the time varies only from 750 to 770 seconds, which is less than 10%. Therefore the overhead in handling the maximum support threshold is very small, and mining interesting itemsets with maximum support thresholds can be considered as an alternative when users, who may have some special knowledge of the data, want to find interesting but non-trivial itemsets.

6 Conclusion

In this paper, we propose two algorithms for mining N-most interesting itemsets. We allow users to control the number of results. Experiments show that our proposed method outperforms the previous Itemset-Loop algorithm by a large margin and it is also comparable to the FP-tree algorithm, even when given an ideal threshold. For thresholds that are too small for the original FP-tree algorithm, our proposed method can have a much superior performance. For thresholds that are too large, the original FP-tree algorithm will not give a proper answer, in fact it may not return any itemsets.

With the efficient BOMO algorithm, we then consider the constraint-based itemsets mining. We define item constraints which allow users to specify the particular set of items they are interested. This caters for the case where particular users would only like to look for certain interesting patterns. We propose the Double FP-trees approach for mining constraint-based interesting itemsets, and show by experiments that it is highly efficient.

Acknowledgements: The authors thank the anonymous reviewers for their helpful comments which have led to great enhancements on the paper. This research is supported by the RGC Earmarked Research Grant CUHK 4436/99E and 4179/01E.

References

[1] Charu C. Aggarwal and Philip S. Yu. Mining large itemsets for association rules. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 23–31, March 1998.

[2] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Canada*, 1998.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 487–499, 1994.

[4] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 255–264, 1997.

[5] IBM Almaden Research Center. Synthetic Data Generation Code for Associations and Sequential Patterns. http://www.almaden.ibm.com/cs/quest.

[6] Y.L. Cheung and A. Fu. An fp-tree approach for mining n-most interesting itemsets. In *Proceedings of the SPIE Conference on Data Mining*, 2002.

[7] Ada Wai-chee Fu, Renfrew Wang-wai Kwong, and Jian Tang. Mining N-most Interesting Itemsets. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS)*, 2000.

[8] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.

[9] Jiawei Han, J. Pei, and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 2000.

[10] Minnesota Population Center in University of Minnesota. IPUMS-98. http://www.ipums.umn.edu/usa/samples.html.

[11] M. Kamber, J. Han, and J. Y. Chiang. Mining partial periodicity using frequent pattern trees. In *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pages 207–210, 1997.

[12] M. Klemettinen, H. Mannila, P. Ronkainen, Toivonen H., and A.I. Verkamo. Finding interesting rules from large sets of discovered associated rules. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 401–408, 1994.

[13] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 220–231, 1997.

[14] C. Leung, L. Lakshmanan, and R. Ng. Exploiting succinct constraints using fp-trees. In *ACM SIGKDD Explorations (Special Issue on Constraints in Data Mining) Vol 4, Issue 1*, volume 4, pages 40–49, June 2002.

[15] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 175–186, 1995.

[16] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th ICDT Conf.*, 1999.

[17] J. Pei and J. Han. Constrained frequent pattern mining: A pattern-growth view. In *ACM SIGKDD Explorations (Special Issue on Constraints in Data Mining) Vol 4, Issue 1*, volume 4, pages 31–39, June 2002.

[18] J. Pei, J. Han, and Mao R. Closet: an efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'00)*, May 2000.

[19] J. Pei, Jiawei Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.

[20] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large database. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 432–443, 1995.

[21] Pradeep Shenoy, Jayant R. Haritsa, S. Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. Turbo-charging Vertical Mining of Large Databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 1–12, 2000.

[22] C. Silverstein, S. Brin, R. Motwani, and J Ullman. Scalable techniques for mining causal structures. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, pages 594–605, 1998.

[23] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pages 67–73, 1997.

[24] K. Wang, Y. He, and J. Han. Pushing support constraints into frequent itemset mining. In *School of Computing, National University of Singapore*, 2000.

[25] Ke Wang, Yu He, and Jiawei Han. Mining frequent itemsets using support constraints. In *Proceedings of the 26th VLDB Conference*, 2000.

[26] Zijian Zheng, Ron Kohavi, and Llew Mason. Real world performance of association rule algorithms. In *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pages 401–406, 2001.