APPLYING BIT-VECTOR PROJECTION APPROACH FOR EFFICIENT MINING OF N-MOST INTERESTING FREQUENT ITEMSETS

Zahoor Jan, Shariq Bashir, A. Rauf Baig
FAST-National University of Computer and Emerging Sciences, Islamabad
A. K. Brohi Road H-11/4, Islamabad
zahoor_jan2003@yahoo.com, shariq.bashir@nu.edu.pk, rauf.baig@nu.edu.pk

ABSTRACT
Real world datasets are sparse, dirty and contain hundreds of items. In such situations, discovering interesting rules (results) using traditional frequent itemset mining approach by specifying a user defined input support threshold is not appropriate. Since without any domain knowledge, setting support threshold small or large can output nothing or a large number of redundant uninteresting results. Recently a novel approach of mining N-most interesting itemsets is proposed, which discovers only top N interesting results without specifying any user defined support threshold. However, mining N-most interesting itemsets are more costly in terms of itemset search space exploration and processing cost. Thereby, the efficiency of mining process highly depends upon the itemset frequency (support) counting, implementation techniques and projection of relevant transactions to lower level nodes of search space. In this paper, we present a novel N-most interesting itemset mining algorithm (N-MostMiner) using the bit-vector representation approach which is very efficient in terms of itemset frequency counting and transactions projection. Several efficient implementation techniques of N-MostMiner are also present which we experienced in our implementation. Our different experimental results on benchmark datasets suggest that the N-MostMiner is very efficient in terms of processing time as compared to currently best algorithm BOMO.

KEY WORDS
Data Mining, Association Rules Mining, Frequent Itemset Mining, N-most interesting itemset mining, Bit-vector representation approach, Bit-vector Projection.

1 Introduction

Since the introduction of association rules mining by Agrawal et al. [1], it has now become one of the main pillars of data mining and knowledge discovery tasks and has been successfully applied in many interesting association rules mining problems such as sequential pattern mining [2], emerging pattern mining [8], classification [13], maximal and closed itemset mining [3, 5, 14]. Using the support-confidence framework presented in [1], the problem of mining the complete association rules from transactional dataset is divided into two parts – (a) finding complete frequent itemsets with support (an itemset's occurrence in the dataset) greater than minimum support threshold, (b) generating association rules from frequent itemsets with confidence greater than minimum confidence threshold. In practice, the first phase is the most time-consuming task, which requires the heaviest frequency counting operation for each candidate itemset.

Let *TDS* be our transactional dataset and *I* be a set of distinct items in the *TDS*. Each individual transaction *t* in *TDS* consists a subset of single items such as $t_I \subseteq I$. We call *X* an itemset, if it contains $X \subseteq I$. Let *min-sup* be our minimum support threshold, we call an itemset *X* frequent if its support *(support(X))* is greater than *min-sup*; otherwise infrequent. By following the Apriori property [1] an itemset *X* cannot be a frequent itemset, if one of its subset is infrequent. The frequent itemsets mining algorithms take a transactional dataset *(TDS)* and *min-sup* as an input and output all those itemsets which appear in at least *min-sup* number of transactions in *TDS*. However, the real datasets are sparse, dirty and contain hundreds of items. In such situations, users face difficulties in setting this *min-sup* threshold to obtain their desired results. If *min-sup* is set too large, then there may be a small number of frequent itemsets, which does not give any desirable result. If the *min-sup* is set too small, then there may be a large number of redundant short uninteresting itemsets, which not only takes a large processing time for mining but also increases the complexity of filtering un-interesting itemsets. In both situations, the ultimate goal of mining interesting frequent itemsets is undermined. We refer the readers [7] for further reading about the problem of setting this user defined *min-sup* threshold without any previous domain knowledge about the dataset.

For handling such situations, Fu et al. in [9] presents a novel technique of mining only N-most interesting frequent itemsets without specifying any *min-sup* threshold. The problem of mining N-most interesting frequent itemsets of size *k* at each level of $1 \leq k \leq k_{max}$, given *N* and k_{max} can be considered from the following definitions.

TDS	The given transactional dataset
k_{max}	Upper bound on the size of interesting

	itemsets to be found
ξ	Current support threshold for all the itemsets
ξ_k	Current support threshold for the k-itemsets

Definition 1: *k-itemsets:* Is a set of items containing k items.

Definition 2: *N-most interesting k-itemsets:* Let *I* be the set of sorted itemsets by descending support. Let *S* be the support of the N-th *k*-itemset in the set *I*. The N-most interesting *k*-itemsets in the set *I* are those itemsets having *support* $\geq S$.

Definition 3: *The N-most interesting itemsets:* Is the union of the N-most interesting *k*-itemsets for each $1 \leq k \leq k_{max}$.

1.1 Motivation behind our Work

As clear from the Definition 2 described above, the Apriori property presented by Agrawal et al. in [1] can not be applied in mining N-most interesting itemset algorithm for pruning un-interesting itemsets. Since the superset of any uninteresting *k*-itemset may be the N-most interesting itemset of any level *h such that* $1 \leq h \leq k_{max}$. Therefore, a large area of itemset search space is explored as compared to traditional frequent itemset mining approach. In our different computational experiments on several sparse and dense benchmark datasets, we found that the efficiency of mining N-most interesting itemsets highly depends upon two main factors. (1) Dataset representation approach used for frequency counting [10]. (2) Projection of relevant transactions to lower level nodes of search space. The projection of relevant transactions at any node *n*, are those transactions of dataset which contain the node *n*'s itemset as subset [11]. Therefore, to increase the efficiency of mining N-most interesting itemsets, in this paper we present a novel efficient algorithm (N-MostMiner) using Bit-vector dataset representation approach. The major advantage of using Bit-vector dataset representation approach is that, it optimizes the itemset frequency counting cost with a factor of 1/32, if we represent 32 rows per single vertical bit-vector region [16]. Before our work, Bit-vector dataset representation approach has been successfully applied in many complex association rules problems such as maximal frequent itemsets mining [6], sequential patterns mining [4] and fault-tolerant frequent itemsets mining [12]. In addition to dataset representation approach, this paper also presents a novel bit-vector projection technique which we named as projected-bit-regions (PBR). The main advantage of using PBR in N-MostMiner is that, it consumes a very small processing cost and memory space for projection. Our different experiments on benchmark datasets suggest that mining N-most interesting itemsets using N-MostMiner is fast and efficient than the currently best algorithm of N-most interesting itemsets mining BOMO [7].

2 N-MostMiner: Algorithmic Description

In this section, we describe our N-most interesting itemset mining algorithm (N-MostMiner) with its several techniques used for fast itemset frequency counting and projection. For *k*-itemset representation at any node, the N-MostMiner uses the vertical bit-vector representation approach [6]. In a vertical bit-vector representation, there is one bit for each transaction of dataset. If item *i* appear in transaction *j*, then the bit *j* of bit-vector *i* is set to one; otherwise the bit is set to zero. Figure 1 (b) shows the vertical bit-vector representation of Figure 1 (a) dataset. To count the frequency of an *k*-itemset e.g. *{AB}* we need to perform a bitwise-AND operation on bit-vector *{A}* and bit-vector *{B}*, and resulting ones in bit-vector *{AB}* denotes the frequency of *k*-itemset *{AB}*.

Transaction	Items
01	A B C
02	A B I
03	B
04	C I
05	A B D
06	A B C D
07	A

(a)

Transaction	A	B	C	D	I
01	1	1	1	0	0
02	1	1	0	0	1
03	0	1	0	0	0
04	0	0	1	0	1
05	1	1	0	1	0
06	1	1	1	1	0
07	1	0	0	0	0

(b)

Figure 1: (a) A sample transactional dataset. (b) Bit-vector representation of sample dataset.

2.1 Itemset Generation in N-MostMiner

Let < be some lexicographical order of items in Transactional Dataset (TDS) such that for every two items *a* and *b*, $a \neq b$: $a < b$ *or* $a > b$. The search space of mining N-most interesting itemset mining can be considered as a lexicographical order [15], where root node contains an empty itemset, and each lower level *k* contains all the *k*-itemsets. Each node of search space is composed of head and tail elements. Head denotes the itemset of node, and items of tail are the possible extensions of new child itemsets. For example with four items *{A, B, C, D},* in Figure 2 root's head is empty *⟨()⟩* and tail is composed with all of items *⟨(A, B, C, D)⟩*, which generates four possible child nodes *{head ⟨(A)⟩: tail ⟨(BCD)⟩}, {head ⟨(B)⟩: tail ⟨(CD)⟩}, {head ⟨(C)⟩: tail ⟨(D)⟩}, {head ⟨(D)⟩: tail ⟨{}⟩}*. At any node *n*, the candidate N-most k-itemset or child nodes of *n* are generated by performing join operation on *n's* head itemset with each item of *n's* tail, and checked for frequency or support counting. This itemset search space can be traverse either by depth first order or breadth first search. At each node, infrequent items from tail are removed by dynamic reordering heuristic [5] by comparing their support with all the itemsets support (ξ) and k-itemsets support (ξ_k) thresholds [7]. In addition to this, our algorithm also ordered the tail items by decreasing support which keeps the search space as small as possible [5].

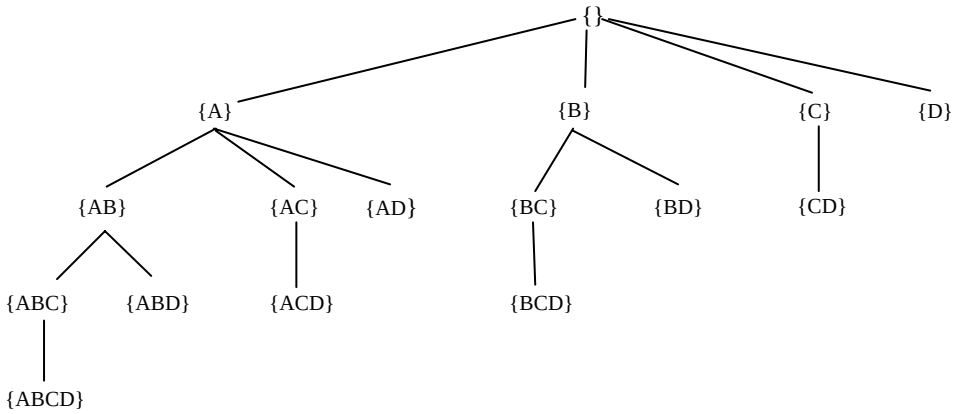


Figure 2: Lexicographic tree of four items ⟨A, B, C, D⟩.

2.2 *k*-Itemset Frequency Calculation

To check, whether any tail item X of node n at level k is N-most interesting k-itemset or not, we must check its frequency (support) in *TDS*. Calculating itemset frequency in bit-vector representation requires applying bitwise-$\wedge$ operation on *n.head* and X bit-vectors, which can be implemented by a loop; which we call **simple loop**, where each iteration of **simple loop** apply bitwise-$\wedge$ operation on some region of *n.head* with X bit-vectors. Since 32-bit CPU supports 32-bit $\wedge$ per operation, hence each region of X bit-vector is composed of 32-bits (represents 32 transactions). Therefore calculating frequency of each itemset by using **simple loop** requires applying bitwise-$\wedge$ on all regions of *n*.head with X bit-vectors. However, when the dataset is sparse, and each item is presented in few transactions, then counting itemset frequency by using **simple loop** and applying bitwise-$\wedge$ on those regions of bit-vectors which contain zero involves many unnecessary counting operations. Since the regions which contain zero, will contribute nothing to the frequency of any itemset, which will be superset of k-itemset. Thereby, removing these regions from head bit-vectors (using projection) in earlier stages of search space is more beneficial and useful.

```
(1) for each region index ℓ in projected-bit-regions of head
(2)     AND-result = bitmap-X [ℓ] ∧ bitmap-head [ℓ]
(3)     Support-X = Support-X + number of ones(AND-result)
```

Figure 3: Pseudo code of Bit-vector projection using **PBR** technique.

2.3 Bit-vector Projection Using Projected-Bit-Regions (PBR)

For efficient projection of bit-vectors, the goal of projection should be such as, to bitwise-$\wedge$ only those regions of head bit-vector ⟨*bitmap(head)*⟩ with tail item X bit-vector ⟨*bitmap(X)*⟩ that contain a value greater than zero and skip all others. Obviously for doing this, our counting procedure must be so powerful and have some information which guides it, that which regions are important and which ones it can skip. To achieve this goal, we propose a novel bit-vector projection technique **PBR** (Projected-Bit-Regions). With projection using **PBR**, each node Y of search space contains an array of valid region indexes $PBR_{\langle Y \rangle}$ which guides the frequency counting procedure to traverse only those regions which contain an index in array and skip all others. Figure 3 shows the code of itemset frequency calculation using **PBR** technique. In Figure 3, the line 1 is retrieving a valid region index ℓ in ⟨*bitmap (head)*⟩, while the line 2 is applying a bitwise-$\wedge$ on ⟨*bitmap (head)*⟩ with ⟨*bitmap (X)*⟩ on region ℓ.

One main advantage of bit-vector projection using **PBR** is that, it consumes a very small processing cost for its creation, and hence can be easily applied on all nodes of search space. The projection of child nodes at any node n can be created either at the time of frequency calculation if pure depth first search is used, or at the time of creating head bit-vector if dynamic reordering is used. The strategy of creating $PBR_{\langle X \rangle}$ at node n for tail item X is as; when the **PBR** of ⟨*bitmap(n)*⟩ are bitwise-$\wedge$ with ⟨*bitmap(X)*⟩ a simple check is perform on each bitwise-$\wedge$ result. If the value of result is greater than zero, then an index is allocated in $PBR_{\langle n.head \cup X \rangle}$. The set of all indexes which contain a value greater than zero makes the projection of $\{n.head \cup X\}$ node.

2.4 Memory Requirement

Some other advantages of projection using **PBR** are that, it is a very scalable approach and consumes very small amount of memory during projection and can be applicable on very large sparse datasets. Scalability is achieved as; we know that by traversing search space in depth first order; a single tree path is explored at any time. Therefore a single **PBR** array for each level of path needs to remain in memory. As a preprocessing step a **PBR** array for each level of maximum path is created and cached in memory. At k-itemset generation time different

paths of search space (tree) can share this maximum path memory and do not need to create any extra projection memory during itemset mining.

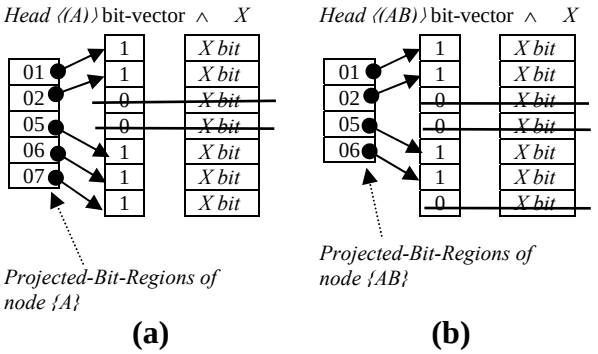


Figure 4: (a) Itemset *(AB ∪X)* frequency calculation using **PBR.** (b) Itemset *(AB ∪X)* frequency calculation using **PBR**.

2.5 Projection Example

Let the second column of Figure 1 (a) shows the transactions of our sample *TDS*. Let the minimum thresholds of ξ *and* ξ_k for all $(1 \le k \le k_{max})$ are equal to 2. For ease of example explanation we will make two assumptions – (a) Each region of bit-vectors contain only a single bit (b) A static alphabetical order will be follow instead of ascending frequency order.

At the start of algorithm, the bit-vector of each single item in the dataset is created. Then, the N-most interesting itemset mining process from the root node is started. At root node, head itemset is *{}*, therefore the PBR of each tail item *X = ⟨(A, B, C, D, E, I)⟩* is created by traversing all *⟨bitmap(X)⟩* regions, and indexing only those which contain a value greater than 0. For example *⟨bitmap (A)⟩* contains a value greater than zero in bit regions *⟨(01, 02, 05, 06, 07)⟩*; thereby these are its PBR *indexes*. Since we are traversing search space in depth first order and item *{A}* is first in alphabetical order. Therefore, root node creates a child node with *head ⟨(A)⟩* and *tail ⟨(B, C, D, E, I)⟩* and iterates to *node {A}*. At *node {A}*, frequency of each tail item *X* is calculated by applying bitwise-∧ on *⟨bitmap (A)⟩* with *⟨bitmap(X)⟩* on $PBR_{\langle A \rangle}$. Figure 4 (a) shows the frequency calculation process of itemset *⟨(X ∪ head.{A})⟩* by using PBR. After frequency calculation tail items *⟨(B: 4, C: 2, D: 2)⟩* are found locally frequent at *node {A}*. Since item *{B}* in *tail.{A}* is next in alphabetical order, therefore the *node {A}* creates a new child node with *head ⟨(AB)⟩* and *tail ⟨(C, D)⟩. PBR = ⟨(01, 02, 05, 06)⟩* of child *node {AB}* at *node {A}* are created by applying bitwise-∧ on *⟨(bitmap (A)⟩* with *⟨(bitmap (B)⟩* on $PBR_{\langle A \rangle}$. The mining process then moves to *node {AB}* with *head ⟨(AB)⟩* and *tail ⟨(C, D)⟩.*

Similarly at *node {AB}*, frequency of each tail item *X* is calculated by applying bitwise-∧ on *⟨bitmap (AB)⟩* with *⟨bitmap(X)⟩* on $PBR_{\langle AB \rangle}$. Figure 4 (b) is showing the frequency calculation process at *node {AB}*. After frequency calculation tail items *⟨(C: 2, D: 2)⟩* are found locally frequent at *node {AB}*. Since item *{C}* in *tail.{AB}* is next in alphabetical order, therefore the mining process creates a child node with *head ⟨(ABC)⟩* and *tail ⟨(D)⟩. PBR = ⟨(01, 06)⟩* of child *node {ABC}* at *node {AB} are* created by applying bitwise-∧ on *⟨bitmap (AB)⟩* with *⟨(bitmap (C)⟩* on $PBR_{\langle AB \rangle}$. Afterward, the mining process iterates to *node {ABC}* with *head ⟨(ABC)⟩* and *tail ⟨(D)⟩.*

At *node {ABC}*, all tail items are infrequent; therefore the mining process stops and backtracks at *node {AB}* and mine other items of its *tail ⟨(D)⟩.* Since *item {D}* is next in alphabetical order, so the mining process creates a new child *node {ABD}* with *head ⟨(ABD)⟩* and *tail ⟨({})⟩.* At *node {ABD},* tail is empty so the mining process stops and backtracks at *node {A}*. Similarly, the mining process mines other items of *node {A}* tail in same fashion. After mining all tail items of *node {A}*, the mining process then backtracks at root node, and mines other tail items of root in depth first order. Due to the lack of space we could not give a detailed example, but the basic idea for mining other itemsets is similar.

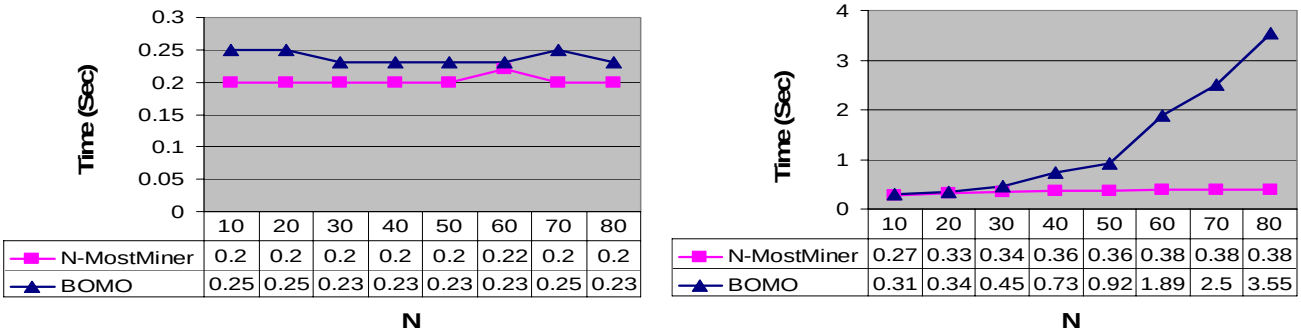
N-MostMiner (Node n)

```
(1)  for each item X in n.tail
(2)    for each region index ℓ in PBR⟨n⟩
(3)      AND-result = bit-vector[ℓ] ∧ head_bit_vector of
         n [ℓ]
(4)      Support[X] = Support[X] +
         number_of_ones(AND-result)

(5)  Remove infrequent items from n.tail, if support less than
     ξk or ξ
(6)  Reorder them by decreasing support

(7)  for each item X in n.tail
(8)    m.head = n.head ∪X
(9)    m.tail = n.tail – X
(10)     for each region index ℓ in PBR⟨n⟩
(11)   AND-result = bit-vector[ℓ] ∧ head-bit-vector[ℓ]
(12)         If AND-result > 0
(13)             Insert ℓ in PBR⟨m⟩
(14)             head bit-vector of m [ℓ] = AND- result
(15)   N_most_k_itemset = N_most_k_itemset ∪ n.head
(16)   if N_most_k_itemset == Nth itemset
(17)     increase the support ξk by assigning minimum
         value support of N_most_k_itemset
(18)   N_MostMiner (m)
```

Figure 5: Pseudo code of N-MostMiner

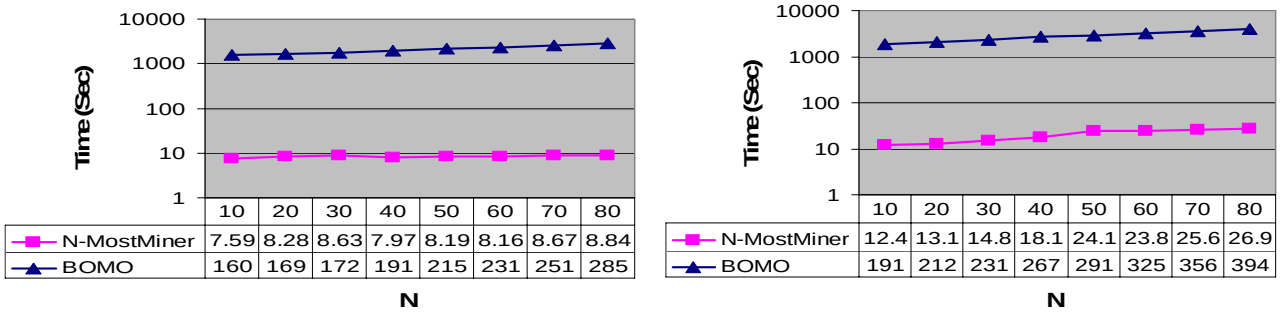
2.6 Adjusting threshold ξ_k and ξ

At the start of algorithm, we set the minimum support threshold of all the *k*-itemsets ξ_k $(1 \le k \le k_{max})$ to zero. With $\xi_k = 0$ we would blindly include any *k*-itemsets in our current set of N-most k-itemset. Once the algorithm encountered any Nth *k*-itemset for some level $k \le k_{max}$, we



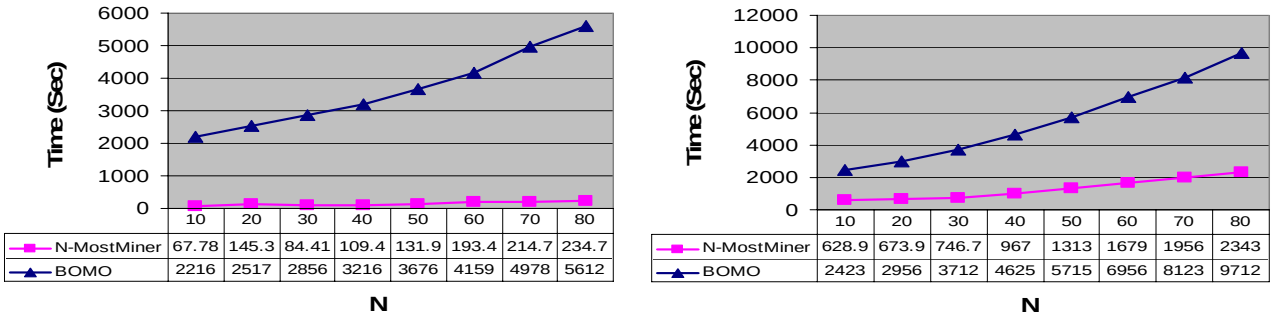
(a) $k_{max} = 5$ (b) $k_{max} = 10$

Figure 6: Performance results on Mushroom dataset



(a) $k_{max} = 5$ (b) $k_{max} = 10$

Figure 7: Performance results on T40I10D100K dataset



(a) $k_{max} = 5$ (b) $k_{max} = 10$

Figure 8: Performance results on BMS-POS dataset

can safely set the minimum support threshold of ξ_k by assigning to it the minimum value among the support of the N-most *k*-itemset discovered so far. Next only that *k*-itemset will be added in the N-most *k*-itemsets result, which contains the support greater than ξ_k. Once the algorithm finds such *k*-itemset, the support threshold of ξ_k is adjusted again by assigning to it the minimum value among the support of the N-most *k*-itemsets.

In the mining phase, we set the initial threshold value of ξ (all the itemsets support) to be zero. During mining, the algorithm increases ξ by assigning to it the minimum value among the supports of the Nth most frequent *k*-itemset discovered so far for $\xi = min\ (\xi_1, \xi_2, \ldots\ \xi_{kmax})$. Figure 5 shows the pseudo code of N-MostMiner.

3 Experiments

In this section we show our performance results of N-MostMiner on a number of benchmark datasets. For experimental purpose we used the original source code of BOMO, which is freely available at http://www.cse.cuhk.edu.hk/~kdd/program.html. All the source code of N-MostMiner is written in C language. The experiments are performed on 1.5 GHz processor with main memory of size 256 MB, running windows XP 2005 professional. For experiments, we used the benchmark datasets available at http://fimi.hi.cs.datasets. These datasets are frequently used in many frequent

itemset mining algorithms. Due to the lack of space we can't show our detailed experiments that we have performed on all the available datasets. Table 1 shows the description of datasets that we used in our experiments. We perform over computational experiments using the different N-most values 10, 20, 30, 40, 50, 60, 70 and 80 under two k_{max} thresholds values, k_{max} = 5 ($1 \leq k \leq 5$) and k_{max} = 10 ($1 \leq k \leq 10$). The performance measure is the execution time of the algorithms under different N and k_{max} threshold values. As clear from the Figures (6, 7 and 8) results the N-MostMiner outperforms the BOMO, on almost all levels of mining thresholds on all types of sparse and dense datasets. This is due to its fast frequency counting of *k*-itemset, projection and efficient implementation techniques. Whereas the BOMO requires several indirect memory accesses during each *k*-itemset frequency counting, which slows down the whole mining process.

Dataset	Items	Average Transactional Length	Records
T40I10D100K	1000	40	100,000
Mushroom	119	23	8,124
BMS-POS	1658	7.5	515,597

Table 1: Datasets used in the our computational experiments

4 Conclusion

Mining N-most interesting itemsets are more costly in terms of processing time due to large area of itemset search space exploration as compare to traditional frequent itemset approach. Due to large number of candidate itemsets generation, the efficiency of mining N-most interesting itemsets algorithm highly depends upon the two main factors – (a) Dataset representation approach for fast frequency counting – (b) Projection of relevant transactions to lower level nodes of search space. In this paper we present a novel N-most interesting itemset mining algorithm (N-MostMiner) using bit-vector representation approach, which is very efficient in terms of *k*-itemset frequency counting. For projection we presents a novel bit-vector projection technique PBR (projected-bit-regions), which is very efficient in terms of processing time and memory requirement. Several efficient implementation techniques of N-MostMiner are also present, which we experienced in our implementation. Our experimental results on benchmark datasets suggest that mining N-most interesting itemsets using N-MostMiner is highly efficient in terms of processing time as compared to currently best algorithm BOMO. This shows the effectiveness of our algorithm.

Reference

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *In Proceedings of international conference on Very Large Data Bases (VLDB),* pp. 487-499, Sept. 1994.

[2] R. Agrawal and R. Srikant, "Mining Sequential Patterns", *In proceedings of international conference on Data Engineering (ICDE)*, pp. 3-14, Mar. 1995.

[3] R. Agrawal, C. Agrawal, and V. Prasad, "Depth first generation of long patterns", *In SIGKDD*, 2000.

[4] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, "Sequential Pattern Mining Using Bitmaps", *In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, July, 2002.

[5] R. J. Bayardo, "Efficiently mining long patterns from databases", *In SIGMOD 1998*: 85-93.

[6] D. Burdick, M. Calimlim, and J. Gehrke, "Mafia: A maximal frequent itemset algorithm for transactional databases", *In proceedings of International Conference of Data Engineering*, pp. 443-452, 2001.

[7] Y. L. Cheung, A. W. Fu, "An FP-tree Approach for Mining N-most Interesting Itemsets", *In Proceedings of the SPIE Conference on Data Mining*, 2002.

[8] G. Dong, J. Li, "Efficient mining of emerging patterns: Discovering trends and differences", *In proceedings of 5th ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD'99)*, San Diego, CA, USA, pp. 43-52, 1999.

[9] A. W. C. Fu, R. W. W. Kwong, and J. Tang, "Mining N-most Interesting Itemsets", *In proceedings of international symposium on Methodologies for Intelligent Systems (ISMIS)*, 2000.

[10] Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations, B. Goethals and M.J. Zaki, eds., CEUR Workshop Proc., vol. 80, Nov. 2003, http://CEUR-WS.org/Vol-90.

[11] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation", *In proceedings of SIGMOD*, pages 1–12, 2000.

[12] J. L. Koh, P. Yo, "An Efficient Approach for Mining Fault-Tolerant Frequent Patterns based on Bit Vector Representations", *In proceedings of 10th International Conference DASFAA 2005*, Beijing, China, April 17-20, 2005.

[13] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining", *In* proceedings of *KDD'98*, New York, NY, Aug. 1998.

[14] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules", *In 7th international conference on Database Theory*, January 1999.

[15] R. Rymon, "Search through Systematic Set Enumeration", *In proceedings of third international conference on Principles of Knowledge Representation and Reasoning*, 1992, pp. 539 –550.

[16] T. Uno, M. Kiyomi, H. Arimura, "LCM ver 3: Collaboration of Array, Bitmap and Prefix Tree for Frequent Itemset Mining", *In 1st International Workshop on Open Source Data Mining (in conjunction with SIGKDD-2005),* 2005.