Index-Maxminer: a New Maximal Frequent Itemset Mining Algorithm.

Article *in* International Journal of Artificial Intelligence Tools · April 2008

DOI: 10.1142/S021821300800390X · Source: DBLP

3 authors, including:

Wei Song
North China University of Technology
55 PUBLICATIONS **359** CITATIONS
SEE PROFILE

Zhangyan Xu
31 PUBLICATIONS **298** CITATIONS
SEE PROFILE

International Journal on Artificial Intelligence Tools
Vol. 17, No. 2 (2008) 303–320

World Scientific
www.worldscientific.com

INDEX-MAXMINER: A NEW MAXIMAL FREQUENT ITEMSET MINING ALGORITHM

WEI SONG

College of Information Engineering,
North China University of Technology,
Beijing, 100144, China
sgyzfr@yahoo.com.cn

BINGRU YANG

School of Information Engineering,
University of Science and Technology Beijing,
Beijing, 100083, China
bryang_kd@yahoo.com.cn

ZHANGYAN XU

Department of Computer, Guangxi Normal University,
Guilin, 541004, China
xyzwlx72@yahoo.com.cn

Received 10 October 2006
Accepted 22 March 2007

Because of the inherent computational complexity, mining the complete frequent itemset in dense datasets remains to be a challenging task. Mining Maximal Frequent Itemset (MFI) is an alternative to address the problem. Set-Enumeration Tree (SET) is a common data structure used in several MFI mining algorithms. For this kind of algorithm, the process of mining MFI's can also be viewed as the process of searching in set-enumeration tree. To reduce the search space, in this paper, a new algorithm, Index-MaxMiner, for mining MFI is proposed by employing a hybrid search strategy blending breadth-first and depth-first. Firstly, the index array is proposed, and based on bitmap, an algorithm for computing index array is presented. By adding subsume index to frequent items, Index-MaxMiner discovers the candidate MFI's using breadth-first search at one time, which avoids first-level nodes that would not participate in the answer set and reduces drastically the number of candidate itemsets. Then, for candidate MFI's, depth-first search strategy is used to generate all MFI's. Thus, the jumping search in SET is implemented, and the search space is reduced greatly. The experimental results show that the proposed algorithm is efficient especially for dense datasets.

Keywords: Data mining; association rule; maximal frequent itemset; index array; set-enumeration tree.

1. Introduction

Frequent Itemset Mining (FIM) is one of the major problems in many data dining applications. It started as a phase in the discovery of association rules,[1] but has been generalized independent of these to many other patterns. For example, frequent sequences,[2] episodes,[3] and frequent subgraphs.[4]

The FIM problem has been extensively studied in the last years. Several variations to the original Apriori algorithm,[5] as well as completely different approaches, have been proposed.[6–9] Most of the well-studied frequent pattern mining algorithms, including Apriori,[5] ECLAT,[8] and FP-growth,[9] mine the complete set of frequent itemsets. These algorithms may have good performance when the support threshold is high and the pattern space is sparse. However, when the support threshold drops low, the number of frequent itemsets goes up dramatically, the performance of these algorithms deteriorates quickly because of the generation of a huge number of itemsets. Moreover, the effectiveness of the mining of the complete set degrades because it generates numerous redundant itemsets.

Hence, there has been recent interest in mining Maximal Frequent Itemset (MFI). Maximal frequent itemsets are frequent itemsets whose supersets are infrequent and all its subsets are frequent. Given a set of MFI, it is easy to analyze some interesting properties of the database, such as the longest pattern, etc. Thus, they can be used for generating all possible association rules. There are also applications where the MFI is adequate, for example, the combinatorial pattern discovery in biological applications.[10]

Gunopulos *et al.*[11] presented a randomized algorithm for identifying maximal frequent itemsets. Their algorithm works by iteratively attempting to extend a working pattern until failure. A randomized version of the algorithm that does not guarantee every maximal frequent itemset will be returned is evaluated and found to efficiently extract long frequent itemsets. Unfortunately, it remains to be seen how the proposed complete version of the algorithm would perform in practice.

Bayardo[12] introduced MaxMiner which extends Apriori to mine only maximal frequent itemsets. MaxMiner performs a "look-ahead" to do superset frequency pruning. It still needs several passes of the database to find the maximal frequent itemsets.

Pincer Search[13] also extends Apriori through the introduction of a bidirectional search. The algorithm uses a typical Apriori bottom-up search but extends it by incorporating additional pruning through the use of a Maximal Frequent Candidate Sets (MFCS) that approaches the valid search space border and hence the discovery of MFS from the top-down.

DepthProject[14] finds long itemsets using a depth-first search of a lexicographic tree of itemsets, and uses a counting method based on transaction projections along its branches. It returns a superset of the MFI and would require post-pruning to eliminate non-maximal itemsets.

MAFIA[15,16] uses a linked list to organize all frequent itemsets. Since all information contained in the database is compressed into the bitvectors, mining frequent itemsets and candidate frequent itemset generation can be done by bitvector *and*-operations. Pruning techniques, such as Superset Checking and Parent Equivalency Pruning, are also used in the MAFIA algorithm.

GenMax[17] takes a novel approach to maximality testing denoted *progressive focusing* where valid itemsets are first tested against a locally maintained set of MFI's (LMFI). Most nonmaximal valid itemsets are discovered using the local testing therefore reducing the number of subset tests required. GenMax also uses diffsets which become more effective as density increases.

SmartMiner[18] uses a technique to quickly prune candidate frequent itemsets in the itemset lattice. The technique gathers "tail" information for a node in the lattice. The tail information is used to determine the next node to explore during the depth-first mining. Items are dynamically reordered based on the tail information.

MaxDomino[19] uses novel concepts of dominancy factor and collapsibility of transaction for efficiently mining MFI. A top-down with selective bottom-up search strategy is employed. Furthermore, the concept of Longest Maximal (LM) patterns, which is primarily MFI of longest length, is also introduced.

Metamorphosis[20] transforms the dataset to Maximum Collapsible and Compressible (MC^2) format and employs a top down strategy with phased bottom up search for mining MFI.

FPMax[21–23] provides an MFI version of FP-Growth which uses superset checking to construct an auxiliary MFI-Tree. The algorithm proceeds in a similar fashion to FP-Growth, beginning with the initial construction of a FP-Tree and FP-List. However subsequent conditional FP-Tree processing is only conducted where the conditional head, together with all valid items in the head-conditional pattern base (tail), is not a subset of an existing MFI stored within MFI-Tree. If it is not subsumed, the conditional FP-Tree is constructed and, if only consisting of a single path, is appended to MFI-Tree. Analysis proceeds as for FP-Tree, incorporating superset checking optimizations based on FP-structure and processing idiosyncrasies with the final set of MFI's represented in the MFI-Tree.

The Set-enumeration tree[24] and its variations[25–27] are common data structures for several MFI mining algorithms, such as MaxMiner[12] and GenMax,[17] etc. For this kind of algorithm, the process of mining MFI's can also be viewed as the process of searching in set-enumeration tree. To reduce the search space, in this paper, a new algorithm, Index-MaxMiner, for mining MFI is proposed by employing a hybrid search strategy blending breadth-first and depth-first. Firstly, the Index Array is proposed. For a given frequent item, the subsume index of index array presents the co-occurrence of this item with other frequent items that have more support than it. Based on bitmap, an algorithm for computing index array is presented. By adding subsume index to frequent items, Index-MaxMiner discovers the candidate MFI's using breadth-first search at one time, which avoids first-level nodes that would

not participate in the answer set and reduces drastically the number of candidate itemsets. Then, for candidate MFI's, depth-first search strategy is used to generate all MFI's. Thus, the jumping search in set-enumeration tree is implemented, and the search space is reduced greatly. Meanwhile, we investigate in depth the problem of pruning strategies based on index array. The experimental results show that the proposed algorithm is efficient especially over dense datasets.

The remaining of the paper is organized as follows. In Section 2, we briefly revisit the problem definitions of maximal frequent itemset mining and set-enumeration tree. In Section 3, we present the definition of Index Array (IA) and the corresponding algorithm for calculating IA and generating candidate maximal frequent itemsets. In Section 4, we devise algorithm Index-MaxMiner by exploiting the heuristic information provided by IA. A thorough performance study of Index-MaxMiner in comparison with several recently developed efficient algorithms is reported in Section 5. We conclude this study in Section 6.

2. Problem Statement

2.1. *Maximal frequent itemset*

The problem of mining maximal frequent itemsets is formally stated by definitions 1-4 and lemmas 1-2.

Let $\boldsymbol{I} = \{i_1, i_2, \ldots, i_M\}$ be a finite set of items and $\boldsymbol{D}$ be a dataset containing N transactions, where each transaction $t \in \boldsymbol{D}$ is a list of distinct items $t = \{i_1, i_2, \ldots, i_{|t|}\}$, $i_j \in \boldsymbol{I}(1 \leq j \leq |t|)$, and each transaction can be identified by a distinct identifier *tid*.

Definition 1. A set $X \subseteq \boldsymbol{I}$ is called an itemset. An itemset with k items is called a k-itemset.

Definition 2. The support of an itemset X, denoted as $supp(X)$, is defined as the number of transactions in which X occurs as a subset.

Definition 3. For a given $\boldsymbol{D}$, let *min_supp* be the threshold minimum support value specified by user. If $supp(X) \geq min\_supp$, itemset X is called a frequent itemset.

Definition 4. A frequent itemset is called a maximal frequent itemset if it is not a subset of any other frequent itemsets.

According to definitions 3–4, the following lemmas hold.

Lemma 1. *A proper subset of any frequent itemset is not a maximal frequent itemset.*

Lemma 2. *A subset of any frequent itemset is a frequent itemset, a superset of any infrequent itemset is not a frequent itemset.*

Table 1. The example database.

TID	Items	Ordered Items
1	*a b c e f o*	*b f a c e*
2	*a c g*	*g a c*
3	*e i*	*e*
4	*a c d e g*	*d g a c e*
5	*a c e g l*	*g a c e*
6	*e j*	*e*
7	*a b c e f p*	*b f a c e*
8	*a c d*	*d a c*
9	*a c e g m*	*g a c e*
10	*a c e g n*	*g a c e*

An example database is given in Table 1. For convenience we write an itemset $\{a, b, c\}$ as *abc*, and a set of transaction identifiers $\{2, 4, 5\}$ as 245. In the example database, suppose *min_supp*=2, *gace* is a maximal itemset.

2.2. *Set-enumeration tree*

The idea of set-enumeration tree is to expand sets over an ordered and finite item domain as illustrated in Fig. 1 where four items are denoted by their position in the ordering.

The particular ordering imposed on the item domain affects the parent/child relationships in the set-enumeration tree but not its completeness. Set-enumeration trees are not data structures like the hash tree or trie, but instead are used to illustrate how sets of items are to be completely enumerated in a search problem. Note that the tree could be traversed depth-first, breadth-first, or even best-first as directed by some heuristic. Each node n in the set-enumeration tree can be

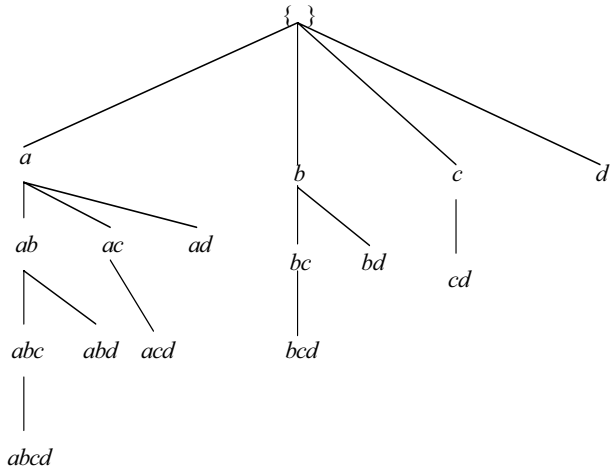


Fig. 1. Complete set-enumeration tree over four items.

represented by two itemsets. The first, called the head and denoted $h(n)$, represents the itemset enumerated by the node. The second itemset, called the tail and denoted $t(n)$, is an ordered set and contains all items not in $h(n)$ that can potentially appear in any sub-node. For example, the node enumerating itemset a in the figure has $h(a) = a$ and $t(a) = bcd$. The ordering of tail items reflects how the sub-nodes are to be expanded. In the case of a static lexical ordering without pruning, the tail of any candidate group is trivially the set of all items following the greatest item in the head according to the item ordering.

3. Index Array and The Generation of Candidate Maximal Frequent Itemsets

3.1. *Index array*

Several MFI mining algorithms, such as MaxMiner[12] and GenMax,[17] etc., can be described using Rymon's generic set-enumeration tree search framework[24] or its variations.[25–27] One of the challenging problems faced by this kind of algorithm is how to extend the itemset enumerated by the current node. That is, for certain node n, how to merge itemset in $h(n)$ and itemsets in $t(n)$. The common ways used to solve this problem are to compute the support of itemsets $h(n)$, $h(n) \cup t(n)$, and $h(n) \cup i$ for all $i \in t(n)$. Based on Lemma 1 and Lemma 2, the supports of itemsets other than $h(n)$ are used for pruning. For example, consider the itemset $h(n) \cup t(n)$. Since $h(n) \cup t(n)$ contains every item that appears in any viable sub-node of n, if it is frequent, then any itemset enumerated by a sub-node will also be frequent but not maximal. Pruning can therefore be implemented by halting sub-node expansion at any candidate node for which $h(n) \cup t(n)$ is frequent. Consider next the itemset $h(n) \cup i$ for some $i \in t(n)$. If $h(n) \cup i$ is infrequent, then any head of a sub-node that contains item i will also be infrequent. Pruning can therefore be implemented by simply removing any such tail item from a candidate group before expanding its sub-node.

However, these kinds of pruning strategies do not always work efficiently. The worst case will lead to $|t(n)|$ redundant operations on merging itemsets and frequency-checking. To reduce the search space, *index array* is proposed. The motivation is to only merge $h(n)$ with itemsets related to $h(n)$ in $t(n)$, so that the resulting itemsets are all frequent. Thus, the number of nodes in the first level of set-enumeration tree is reduced greatly, and the redundant operations on itemset-merging and frequency-checking can be avoided.

The concept of index array is based on the following function.

$$g(X) = \{t \in \boldsymbol{D} | \forall i \in X, i \in t\}.$$

Function g associates with X the transactions related to all items . For example, in the example database in Table 1, $g(bf) = g(b) \cap g(f) = 17$.

Definition 5. An *index array* is an array with size m_1, where m_1 is the number of frequent 1-itemset. Each element of the array corresponds to a tuple (*item*, *sub-*

sume), where *item* is a single item, $subsume(item) = \{j \in \boldsymbol{I} | j \neq item \wedge g(item) \subseteq g(j)\}$. For each element of index array, we call *item* the *representative item*, and $subsume(item)$ the *subsume index*.

The *subsume index* of *item* is an itemset, whose meaning is if $j \in subsume(item)$, the tidset (i.e., sets of identifiers of the transactions which contain a given item) of *item* is the subset of *tidset* of *j*. According to Definition 5, the following theorem holds.

Theorem 1. Let *item* be a frequent itemset, then $X = item \cup subsume(item)$ is also a frequent itemset.

Proof. According to Definition 5, we know that for $\forall i \in subsume(item)$, $g(item) \subseteq g(i)$ holds. Since $g(item) \subseteq g(item)$, we have $g(item) \subseteq g(item \cup subsume(item))$, so $supp(item \cup subsume(item)) \geq supp(item) \geq min\_supp$. That is to say $X = item \cup subsume(item)$ is a frequent itemset. □

Theorem 3.1 shows that, for certain node n enumerated by set-enumeration tree, we can only merge $h(n)$ with items in $subsume(h(n))$, rather than other items in $t(n)$. For example, in the example database in Table 1, $subsume(b)$ is *acef* (items are ordered in increasing order of their supports). If we simply unify b with $t(b) =$ *dfgace*, the result *bdfgace* is not frequent. Then we should unify b with each item in $t(b)$ one by one. We can see from this example that the introduction of index array does reduce the redundant operations on itemset-union and frequency-checking. Based on subsume index, we can easily unify the representative item and frequent itemsets only co-occurrence with it, and identify all candidate maximal frequent itemsets.

3.2. *Generation of candidate maximal frequent itemsets*

The pseudocode for generating candidate maximal frequent itemset is shown in Algorithm 1.

Algorithm 1 starts with the empty set of candidate maximal frequent itemset CMFI (Step 1). Then, the database $\boldsymbol{D}$ is scanned once to determine the frequent single items (Step 2). In Step 3, frequent items are ordered in support ascending order, and the sorted frequent items are assigned to elements of index array as representative items one by one (Steps 4–5). Note that, we sort the frequent items in increasing order of their supports. This is because the lower the support of one item is, the more items could be in its subsume index, the longer are the resulting candidate frequent maximal itemsets. This operation lays foundation for efficient superset checking in Algorithm 2. According to Definition 5, we know that the last ordered item has the highest support. Thus, the calculation of its subsume index can be omitted. In Step 6, the bitmap representation of database $\boldsymbol{D}$ is built. That is for a transaction T, if 1-frequent itemset i is contained by T, then the

bit corresponding to i in T will be set. The set of candidate MFIs are calculated by the main loop (Steps 7-16). New candidate MFIs is formed by intersecting all transactions containing item $index[j].item$ (Steps 9-10). In Step 11, subsume index is obtained. Then, the heuristic information, provided by supports of a frequent item i and items in $subsume(i)$, is used for pruning. This is because we have the following Theorem 3.2.

Algorithm 1 Generating candidate maximal frequent itemset
Input: dataset $\boldsymbol{D}$, *min_supp*
Output: candidate maximal frequent itemset

```
1:  CMFI=∅;        ▷ CMFI is the set of candidate maximal frequent itemsets
2:  Scan database D once. Delete infrequent items;
3:  Sort frequent single items in support ascending order as a_1, a_2, .., a_m;
4:  for each element index[j] of index array do
5:      index[j].item = a_j;
6:  Represent the database D with bitmap;
7:  for each element index[j] in index array do
8:      index[j].subsume = ∅;
9:      candidate=⋂_{t∈g(index[j].item)} t;
10:     CMFI=CMFI∪ candidate;
11:     Store the item corresponding to the i_kth position in candidate
        to index[j].subsume (excluding index[j].item),
        if the value of the i_kth bit in candidate is set;
12:     for each item i in index[j].subsume do
13:         if supp(index[j].item) == supp(i) then
14:             delete index[k] with index[k].item = i;
15:     end for
16: end for
17: write out CMFI;
```

Theorem 2. If item $j \in subsume(i)$ and $supp(i) = supp(j)$, then $i \cup subsume(i) = j \cup subsume(j)$.

Proof. Since $j \in subsume(i)$, according to Definition 5, we know that $g(i) \subseteq g(j)$. Furthermore, $supp(i) = supp(j)$, so $|g(i)| = |g(j)|$. Thus, we have $g(i) = g(j)$.

For $\forall x \in i \cup subsume(i)$, there are two cases:

(i) if $x = i$, we have $g(x) = g(i) = g(j)$, so $g(j) \subseteq g(x)$. That is to say $x \in subsume(j) \subset j \cup subsume(j)$.

(ii) if $x \in subsume(i)$, according to Definition 5, we know that $g(i) \subseteq g(x)$. Since $g(i) = g(j)$, we have $g(j) \subseteq g(x)$. That is to say $x \in subsume(j) \subset j \cup subsume(j)$.

According to the discussions of (i) and (ii), we have $i \cup subsume(i) \subseteq j \cup subsume(j)$.

Similarly, we can also prove that $j \cup subsume(j) \subseteq i \cup subsume(i)$.

Thus, we have $i \cup subsume(i) = j \cup subsume(j)$. □

Example 1. We use example database in Table 1 to illustrate the basic idea of Algorithm 1.

Suppose the support threshold *min_supp* is 2, after the first scan of database, infrequent items are deleted. Then we sort the list of frequent items in support ascending order (If two items have the same supports, they will be sorted according to lexicographic order). The sorted transactions are shown in Table 1. Then the scanned database is represented by bitmap (shown in Table 2).

Table 2. Bitmap representation of example database.

	b	d	f	g	a	c	e
1	1	0	1	0	1	1	1
2	0	0	0	1	1	1	0
3	0	0	0	0	0	0	1
4	0	1	0	1	1	1	1
5	0	0	0	1	1	1	1
6	0	0	0	0	0	0	1
7	1	0	1	0	1	1	1
8	0	1	0	0	1	1	0
9	0	0	0	1	1	1	1
10	0	0	0	1	1	1	1

Then calculate the intersection of transactions that contain certain frequent item one by one. We take frequent item b as example, $candidate = \bigcap_{t \in g(b)} t = 1 \cap 7 = 1010111 \cap 1010111 = 1010111$, where 1 and 7 are *tids*. There are five 1 in *candidate*, since the first bit corresponds to b, the items, corresponds the third, the fifth, the sixth and the last bit, constitute the subsume index of b, that is *face*. Add itemset *bface* into CMFI. Since $supp(b) = supp(f)$ and $f \in subsume(b)$, according to Theorem 3.2, delete the element $index[k]$ in index array with $index[k].item = f$. We can iterate this process similarly, and finally CMFI=$\{bface, dac, gac, ac, e\}$.

4. Index-MaxMiner Algorithm

Algorithm 1 discovers the candidate maximal frequent itemset using breadth-first search at one time. Based on these candidates, the Index-MaxMiner depicted in Algorithm 2 mines all maximal frequent itemsets by depth-first search.

Algorithm 2 starts with the empty set of maximal frequent itemset (Step 1). When CMFI, the set of candidate MFIs, is not empty, the main loop calculates all the MFIs (Steps 2–9). Candidate MFIs are chosen according to the increasing order

of the first item's support (Step 3). For example, the order of itemset *bface* is before *gac*, since $supp(b) < supp(g)$. In Step 4, the selected candidate maximal frequent itemset is deleted from CFMI. Only if no superset exists do we further process the candidate (Step 5). In Step 6, if the support of a candidate maximal frequent itemset equals to the threshold *min_supp*, the candidate MFI will not be expanded. This is because we have the following Theorem 4.1. Depth-First is called when the support of candidate is higher than *min_supp* (Step 8). In Step 10, all MFIs are written out.In procedure Depth-First, we traverse the set-enumeration tree in pure depth-first strategy. Similar to Step 6, Step 13 is also based on Theorem 4.1.

Algorithm 2 Index-MaxMiner algorithm
Input: candidate maximal frequent itemsets generated by Algorithm 1, *min_supp*
Output: maximal frequent itemset

```
1:  MFI=∅;        ▷ MFI is the set of maximal frequent itemsets
2:  while CMFI≠ ∅ do
3:      candidate ← min_≺(CMFI);        ▷ select the first itemset in CMFI
4:      CMFI ← CMFI \ candidate;        ▷ delete candidate in CMFI
5:      if there is no superset of candidate in MFI do
6:          if supp(candidate) ==min_supp then
7:              MFI=MFI ∪ candidate;
8:          else Depth-First(candidate);
9:  end while
10: write out MFI;
 :  Procedure Depth-First(node candidate)
11: for each i ∈ CMFI do
12:     candidate=candidate ∪i;
13:     if supp(candidate)==min_supp then
14:         MFI=MFI ∪ candidate;
15:     if supp(candidate) > min_supp then
16:         Depth-First(candidate);
17: end for
18: if candidate is a leaf and candidate.head is not in MFI then
19:     MFI=MFI ∪ candidate;
20: end Procedure
```

Note that, although our algorithm is in view of the idea of set-enumeration tree, the memory-based tree does not need to be phyisically built. This is the main difference between Index-MaxMiner and the FP-Growth-inspired algorithms, such as FPMax. Furthermore, for FPMax, the FP-Array of FPMax should be constructed recursively. While our index array is only for frequent single items.

Theorem 3. Let CM be a candidate maximal frequent itemset in CMFI with $supp(CM) = min\_supp$, then there exists no item $i \in$ CMFI, such that $CM \cup i$ is a maximal frequent itemset.

Proof. We prove the theorem by contradiction. Suppose there exists $i \in$ CMFI, such that $CM \cup i$ is a maximal frequent itemset. Since function g (described in Section 3.1) is monotonous decreasing, and $CM \subseteq CM \cup i$, we have $g(CM \cup i) \subseteq g(CM)$, i.e., $supp(CM \cup i) \leq supp(CM)$. Assume $supp(CM \cup i) = supp(CM)$, this means $g(CM \cup i) = g(CM)$. According to Algorithm 1, we can easily find that $i \in CM$. Thus, $supp(CM \cup i) < supp(CM) = min\_supp$. So we can draw the conclusion that there exists no item $i \in$ CMFI, such that $CM \cup i$ is a maximal frequent itemset. □

Theorem 4 (Algorithm Correctness). Index-MaxMiner enumerates all maximal frequent itemsets.

Proof. To prove Theorem 4.2, we should prove for $\forall S \in$ MFI, there do exist $R \in$CMFI, such that $R \subseteq S$, where MFI and CMFI are the set of maximal frequent itemsets and the set of candidate maximal frequent itemsets.

We prove the theorem by contradiction. Suppose the above hypothesis does not hold. That is, there exists $S_0 \in$ MFI, for $\forall R \in$CMFI, $R \not\subseteq S_0$ holds. Let $S_0 = b_1 b_2 \ldots b_k$, where $supp(b_i) \leq supp(b_j)$, $1 \leq i < j \leq k$; denote $index[l_i].subsume = sub_i$(where $index[l_i].item = b_i$) and $\bar{S}_0 = S_0 \cup sub_1 \cup sub_2 \cup \ldots \cup sub_k$. Since $\forall x \in sub_i (1 \leq i \leq k) \Rightarrow g(b_i) \subseteq g(x)$, so we have $g(b_i) \subseteq g(sub_i)$, consequently

$$\begin{aligned} g(\bar{S}_0) &= g(S_0 \cup sub_1 \cup sub_2 \cup \cdots \cup sub_k) \\ &= g(b_1) \cap g(b_2) \cap \ldots \cap g(b_k) \cap g(sub_1) \cap g(sub_2) \cap \cdots \cap g(sub_k) \\ &= g(b_1) \cap g(b_2) \cap \cdots \cap g(b_k) \\ &= g(S_0) \end{aligned}$$

So $supp(\bar{S}_0) = supp(S_0)$. Since $S_0 \subseteq \bar{S}_0$, and S_0 is maximal frequent itemset, we have $\bar{S}_0 = S_0$. So $R_i = b_i \cup sub_i (1 \leq i \leq k) \subseteq S_0$. Furthermore, we can easily found that $R_i \in$ CMFI according to Algorithm 1. Thus, for $\forall S \in$ MFI, there do exist $R \in$CMFI, such that $R \subseteq S$. Because Algorithm 2 uses the classical depth-first search as many MFI mining algorithms do, we can draw the conclusion that Index-MaxMiner correctly identifies all and only the maximal frequent itemsets. □

Example 2. We illustrate Index-MaxMiner algorithm on example database in Table 1.

After the processing of Algorithm 1, CMFI={*bface*, *dac*, *gac*, *ac*, *e*}. Select itemset *bface* from CMFI, then CMFI = {*dac*, *gac*, *ac*, *e*}. Since $supp(bface) = min\_supp$, according to Theorem 4.1, further expansion is redundant. Thus, it is

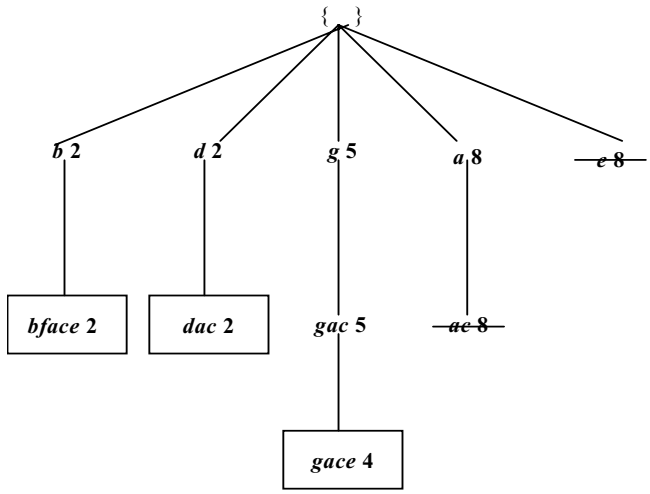


Fig. 2. Search space of Algorithm 1 and Algorithm 2 over example database.

added into MFI directly. Then MFI = {*bface*}. Itemset *dac* is processed similarly. Thus, CMFI = {*gac*, *ac*, *e*}, MFI = {*bface*, *dac*}. Select *gac* from CMFI, then CMFI = {*ac*, *e*}. As $supp(gac) = 5 > min\_supp$, it is necessary to expand it by depth-first search. For the remaining candidates in CMFI, because $ac \subset gac$, *gac* can only unify *e*. Thus, *gace* is generated. Since *gace* is a leaf node, and $supp(gace) = 4 > min\_supp$, add *gace* into MFI. Thus, MFI = {*bface*, *dac*, *gace*}. For the next candidate *ac*, as $ac \subset gace$ and *gace* is in MFI, *ac* will be ignored. Itemset *e* will also be skipped similarly. Thus, CMFI= $\emptyset$, the algorithm stops. The result obtained by Index-MaxMiner is MFI = {*bface*, *dac*, *gace*}.

The search space of Algorithm 1 and Algorithm 2 over example database is shown in Fig. 2. Boxes indicate maximal frequent itemsets and the number following the enumerated itemset is support. From Fig. 2, we can easily find that Index-MaxMiner implements the jumping traversal of complete set-enumeration tree over *bdfgace*, which reduces the search space greatly.

5. Performance Evaluation

We compared the performances of Index-MaxMiner with well-known state-of-the-art algorithms GenMax[17] and FPMAX.[21–23] GenMax is downloaded from the author's homepage http://www.cs.rpi.edu/ zaki/software/, and FPMAX is publicly available from the FIMI repository page http://fimi.cs.helsinki.fi. In this sets of experiments we confirmed the conclusion made at the FIMI 2003 workshop,[28] that there are no clear winners with all databases. Indeed, algorithms that were shown to be winners with some databases were not the winners with others. Some algorithms quickly lose their lead once the support level becomes smaller.

Table 3. Characteristics of datasets used for experiment evaluations.

Datasets	# Items	# Records	Avg. Length
Chess	75	3,196	37
Connect	129	65,557	43
Pumsb	2113	49,046	74
Pumsb*	2088	49,046	50.5
T10I4D100K	870	100,000	11
T40I10D100K	942	100,000	40.5

5.1. *Test environment and datasets*

We chose several real and synthetic datasets for testing the performance of Index-MaxMiner. All datasets are taken from the FIMI repository page http://fimi.cs.helsinki.fi. The connect and chess datasets are derived from their respective game steps. While the PUMS datasets contain census data. Pumsb* is the same as pumsb without items with 80% or more support. Typically, these real datasets are very dense, i.e., they produce many long frequent itemsets even for very high values of support. We also chose a few synthetic datasets, which have been used as benchmarks for testing previous association mining algorithms. These datasets mimic the transactions in a retailing environment. Usually the synthetic datasets are sparse when compared to the real sets. Table 3 shows the characteristics of these datasets. The experiments were conducted on a Windows XP PC equipped with a Pentium 1.5GHz CPU and 1GB of RAM memory.

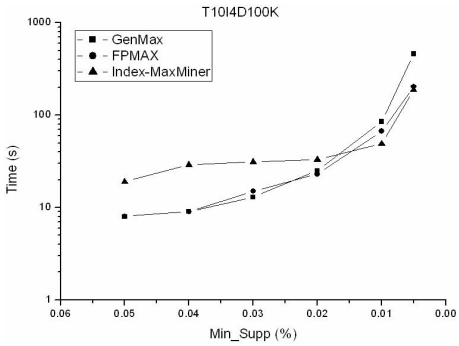
5.2. *The runtime*

Both Fig. 3 and Fig. 4 use total running time as the performance metric. Because all of the datasets are relatively small (the largest dataset is only 20MB), the time to load and prepare the data is negligible and, therefore, the total running time reflects the algorithmic performance only.

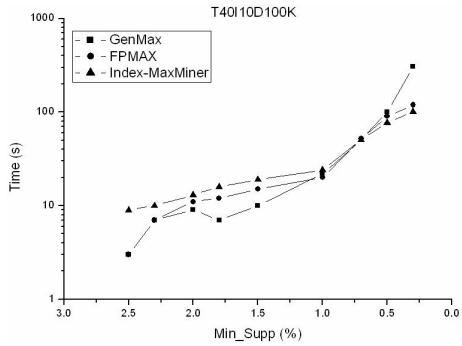
Figure 3 shows the results of comparing Index-MaxMiner with GenMax and FPMAX on sparse data.

On the sparse artificial datasets, GenMax demonstrates the best performance of the three algorithms for higher supports and is around two times faster than Index-MaxMiner. However, note that, as the support drops and the itemsets become longer, Index-MaxMiner passes GenMax in performance to become the fastest algorithm. It is clear that Index-MaxMiner performs better when the itemsets are longer.

The dense datasets in Fig. 4 support the idea that Index-MaxMiner usually runs the fastest on longer itemsets. For most supports on the dense datasets, Index-MaxMiner has the best performance. Index-MaxMiner runs around two to five times faster than GenMax on Connect, Pumsb, and Pumsb* and over five to 10 times faster on Chess. Index-MaxMiner outperforms FPMAX in most cases, but not all cases.

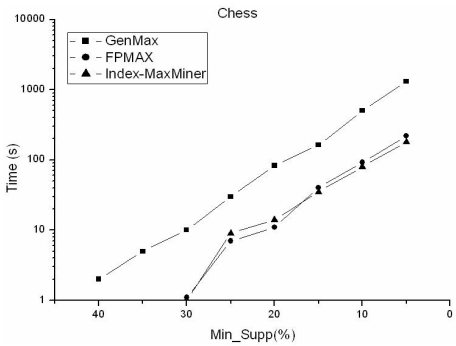


(a) T10I4D100K

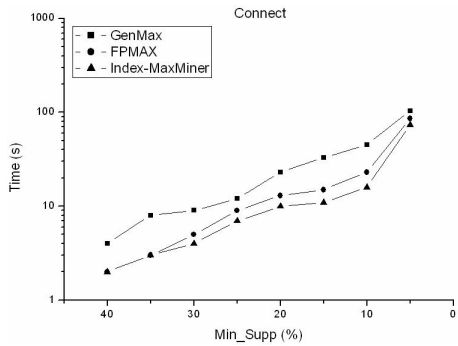


(b) T40I10D100K

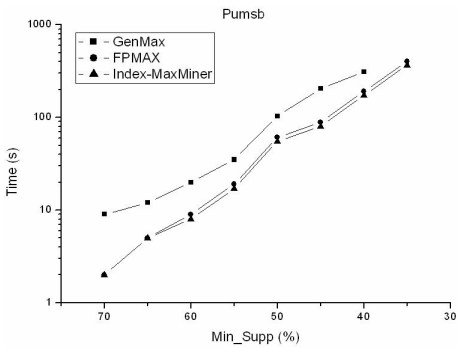
Fig. 3. Execution times (in seconds) required by GenMax, FPMAX, and Index-MaxMiner to mine various publicly available sparse datasets as a function of the minimum support threshold.



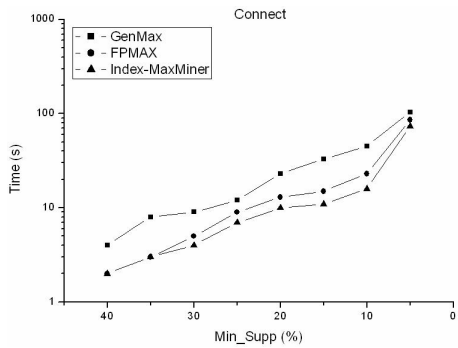
(a) Chess



(b) Connect

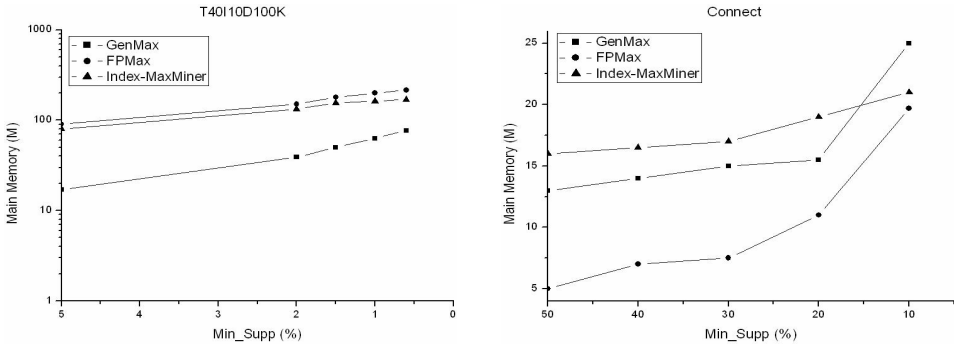


(c) Pumsb



(d) Pumsb*

Fig. 4. Execution times (in seconds) required by GenMax, FPMAX, and Index-MaxMiner to mine various publicly available dense datasets as a function of the minimum support threshold.



(a) T40I10D100K (b) Connect

Fig. 5. Amount of memory required by GenMax, FPMAX, and Index-MaxMiner for mining a sparse (a), and a dense (b) dataset, as a function of the support threshold.

5.3. *Memory consumption*

Figure 5 shows the peak memory consumption of the algorithms on the synthetic and real datasets.

For T40I10D100K, a sparse dataset with large average transaction length and large average itemset length, we can see that the memory consumption of both FPMax and Index-MaxMiner are high, while GenMax consumes the least memory. The question of this situation on a synthetic dataset can be answered as follows: For FPMax, the FP-tree for a sparse dataset and the recursively constructed FP-trees will be big and bushy because there are not many shared common prefixes among the FIs in the transactions. For Index-MaxMiner, the percentage of merged CMFIs by using index array is relatively low. That is the length of generated candidate maximal frequent itemsets could be short. This leads to the increase of the depth of recursive searching in set-enumeration tree. Thus, the effect of using index array is not notable.

For Connect, Because of the compactness of FP-tree data structure for dense datasets, FPMax consumes far less memory than for sparse data sets. In Fig. 5(b), the curves for GenMax overlap the curves for Index-MaxMiner for high minimum support. However, when the minimum support is low, GenMax consumes much more memory than Index-MaxMiner.

6. Conclusions

Set-enumeration tree is a common data structure used in several MFI mining algorithms. For this kind of algorithm, the process of mining MFI's can also be viewed as the process of searching in set-enumeration tree. To reduce the search space of the tree, in this paper, a new algorithm, Index-MaxMiner, for mining MFI is proposed by employing a hybrid search strategy blending breadth-first and depth-first. We have introduced a novel index array technique that allows using

set-enumeration tree more efficiently when mining maximal frequent itemsets. By adding subsume index to frequent items, Index-MaxMiner discovers the candidate MFI's using breadth-first search at one time, which avoids first-level nodes that would not participate in the answer set and reduces drastically the number of candidate itemsets. Then, for candidate MFI's, depth-first search strategy is used to generate all MFI's. Thus, the jumping search in SET is implemented, and the search space is reduced greatly. The experimental results show that our technique greatly reduces the time spent on traversing set-enumeration tree, and works especially well for dense data sets.

Our algorithm can not perform well, until there is enough memory to keep the bitmap representation and index array of input dataset. However, real world datasets may be huge. So to design efficient out-core algorithm for mining maximal frequent itemset is our future work. Furthermore, existing algorithms require to store the possibly huge set of MFIs mined so far to be searched for superset-checking. Although the introduction of progressive focusing (Ref. 17) eased this problem to a certain extent. It is still a challenging issue for MFI mining. Can we give some possible solution to reduce the spatial complexity is our another future work.

Acknowledgments

The authors are indebted to the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of P. R. China (60675030).

References

1. R. Agrawal, T. Imielinski and A. Swami, Mining associations between sets of items in massive databases, in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, eds. P. Buneman and S. Jajodia (ACM Press, Washington, 1993), p. 207-216.
2. R. Agrawal and R. Srikant, Mining sequential patterns, in *Proceedings of the Eleventh International Conference on Data Engineering*, eds. P. S. Yu and A. L. P. Chen (IEEE Computer Society, Taipei, 1995), p. 3-14.
3. H. Mannila, H. Toivonen and A.I. Verkamo, Discovery of frequent episodes in event sequences, *Data Mining and Knowledge Discovery*, 1(3): p. 259-289, 1997.
4. A. Inokuchi, T. Washio and H. Motoda, An apriori based algorithm for mining frequent substructures from graph data, in *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, eds. D. A. Zighed, H. J. Komorowski and J. M. Zytkow (Springer-Verlag, Lyon, 2000), p. 13-23.
5. R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases, in *Proceedings of the 20th International Conference on Very Large Data Bases*, eds. J. B. Bocca, M. Jarke and C. Zaniolo (Morgan Kaufmann, Chile, 1994), p. 487-499.
6. S. Brin, R. Motwani, J. D. Ullman and S. Tsur, Dynamic itemset counting and implication rules for market basket data, in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ed. J. Peckham (ACM Press, Arizona, 1997), p. 255-264.

7. E. Han, G. Karypis and V. Kumar, Scalable parallel data mining for association rules, in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ed. J. Peckham (ACM Press, Arizona, 1997), p. 277-288.
8. M. J. Zaki, S. Parthasarathy, M. Ogihara and W. Li, New algorithms for fast discovery of association rules, in *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, eds. D. Heckerman, H. Mannila and D. Pregibon (AAAI Press, California, 1997), p. 283-296.
9. J. Han, J. Pei and Y. Yin, Mining frequent patterns without candidate generation, in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, eds. W. Chen, J. F. Naughton and P. A. Bernstein (ACM Press, Texas, 2000), p. 1-12.
10. L. Rigoutsos and A. Floratos, Combinatorial pattern discovery in biological sequences: the teiresias algorithm, *Bioinformatics*, 14(1): p. 55-67,1998.
11. G. Gunopulos, H. Mannila and S. Saluja, Discovering all most specific sentences by randomized algorithms, in *Proceedings of the 6th International Conference on Database Theory*, eds. F. N. Afrati and P. G. Kolaitis (Springer-Verlag, Delphi, 1997), p. 215-229.
12. R. J. Bayardo, Efficiently mining long patterns from databases, in *Proceedings ACM SIGMOD International Conference on Management of Data*, eds. L. M. Haas and A. Tiwary (ACM Press, Seattle, 1998), p. 85-93.
13. D. I. Lin and Z. M. Kedem, Pincer-Search: a new algorithm for discovering the maximum frequent set, in *Proceedings of 6th International Conference on Extending Database Technology*, eds. H. J. Schek, F. Saltor, I.Ramos and G. Alonso (Springer-Verlag, Valencia, 1998), p. 105-119.
14. R. C. Agarwal, C. C. Aggarwal and V. V. V. Prasad, Depth first generation of long patterns, in *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining*, eds. R. Ramakrishnan and S. Stolfo (ACM Press, Boston, 2000), p. 108-118.
15. D. Burdick, M. Calimlim and J. Gehrke, MAFIA: a maximal frequent itemset algorithm for transactional databases, in *Proceedings of the 17th International Conference on Data Engineering*, (IEEE Computer Society, Heidelberg, 2001), p. 443-452.
16. D. Burdick, M. Calimlim, J. Flannick, J. Gehrke and T. Yiu, MAFIA: a maximal frequent itemset algorithm, *IEEE Transaction on Knowledge and Data Engineering*, 17(11): p. 1490-1504, 2005.
17. K. Gouda and M. J. Zaki, Efficiently mining maximal frequent itemsets, in *Proceedings of the 2001 IEEE International Conference on Data Mining*, eds. N. Cercone, T. Y. Lin and X.Wu (IEEE Computer Society, California, 2001), p. 163-170.
18. Q. Zou, W. W. Chu and B. Lu, SmartMiner: a depth first algorithm guided by tail information for mining maximal frequent itemsets, in *Proceedings of the 2002 IEEE International Conference on Data Mining*, eds. N. Cercone, T. Y. Lin and X. Wu (IEEE Computer Society, Maebashi City, 2002), p. 570-577.
19. K. Srikumar and B. Bhasker, Efficiently mining maximal frequent sets in dense databases for discovering association rules, *Intelligent Data Analysis*, 8(2): p. 171-182, 2004.
20. K. Srikumar and B. Bhasker, Metamorphosis: mining maximal frequent sets in dense domains, *International Journal on Artificial Intelligence Tools*, 14 (3): p. 491-505, 2005.
21. G. Grahne and J. Zhu, Efficiently using prefix-trees in mining frequent itemsets, in *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, eds. B. Goethals and M. J. Zaki (CEUR-WS.org, 2003).

22. J. Zhu and G. Grahne, Reducing the main memory consumptions of FPmax* and FPclose, in *Proceedings of the ICDM 2004 Workshop on Frequent Itemset Mining Implementations*, eds. R. J. Bayardo, B. Goethals and M. J. Zaki (CEUR-WS.org, 2004).
23. G. Grahne and J. Zhu, Fast algorithms for frequent itemset mining using FP-trees, *IEEE Transaction on Knowledge and Data Engineering*, 17(10): p. 1347-1362, 2005.
24. R. Rymon, Search through systematic set enumeration, in *Proceedings of 3rd International Conference on Principles of Knowledge Representation and Reasoning*, eds. B. Nebel, C. Rich and W. R. Swartout (Morgan Kaufmann, Cambridge, 1992), p. 539-550.
25. R. C. Agarwal, C. C. Aggarwal and V. V. V. Prasad, A tree projection algorithm for generation of frequent item sets, *Journal of Parallel and Distributed Computing*, 61(3): p. 350-371, 2001.
26. M. Kubat, A. Hafez, V. V. Raghavan, J. R. Lekkala and W. K. Chen, Itemset trees for targeted association querying, *IEEE Transaction on Knowledge and Data Engineering*, 15(6): p. 1522-1534, 2003.
27. Y. Li and M. Kubat, Searching for high-support itemsets in itemset trees, *Intelligent Data Analysis*, 10(2): p. 105-120, 2006.
28. B. Goethals, M. J. Zaki, Advances in frequent itemset mining implementations introduction to FIMI03, in *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, eds. B. Goethals and M. J. Zaki (CEUR-WS.org, 2003).