High Performance Mining of Maximal Frequent Itemsets

Gösta Grahne and Jianfei Zhu
Concordia University
{grahne, j_zhu}@cs.concordia.ca

Abstract

Mining frequent itemsets is instrumental for mining association rules, correlations, multi-dimensional patterns, etc. Most existing work focuses on mining *all* frequent itemsets. However, since any subset of a frequent set also is frequent, it is sufficient to mine only the set of *maximal* frequent itemsets. In this paper, we study the performance of two existing approaches, GENMAX and MAFIA, for mining maximal frequent itemsets. We also develop an extension, called FPMAX, of the well known FP-growth method. Since one cannot expect that one single approach will be suitable for all types of data, we analyze the behaviour of the three approaches GENMAX, MAFIA, and FPMAX, under various types of data. We validate our conclusions through careful experimentation with synthetic data, in which the parameters influencing the data characteristics are easily tunable.

We then turn the conclusions into prediction of the performance of each of the three methods for specific data characteristics. We test these predictions of real datasets, and find that they are valid in most cases.

1 Introduction

The space of items in a transactional database gives rise to a subset lattice. The itemset lattice is a conceptualization of the search space when mining frequent itemsets. There are then basically two types of algorithms to mine frequent itemsets, *breadth-first* algorithms and *depth-first* algorithms. The breadth-first algorithms, such as APRIORI [4, 5] and its variants [11], apply a bottom-up level-wise search in the itemset lattice. Candidate itemsets with $k+1$ items are only generated from frequent itemsets with k items. For each level, all candidate itemsets are tested for frequency by scanning the database. On the other hand, depth-first algorithms such as *FP-growth* [9] search the lattice bottom-up in "depth-first" way (one should perhaps say "height-first" way). From a singleton itemset $\{i\}$, successively larger candidate sets are generated by adding one element at a time.

The drawback of mining *all* frequent itemsets is that if there is a large frequent itemset with size ℓ, then almost all 2^ℓ candidate subsets of the items might be generated. However, since frequent itemsets are upward closed, it is sufficient to discover only all *maximal frequent itemsets* (MFI's). A frequent itemset is called *maximal* if it has no superset that is frequent. Thus mining frequent itemsets can be reduced to mining a "border" in the itemset lattice, as introduced in [10]. All itemsets above the border are infrequent, the others that are below the border are all frequent.

Bayardo [6] introduces MAXMINER which extends APRIORI to mine only "long" patterns (maximal frequent itemsets). To reduce the search space, MAXMINER performs not only *subset* infrequency pruning such that a candidate itemset that has an infrequent subset will not be considered, but also a "lookahead" to do *superset* frequency pruning. For any frequent itemset X, find all single items j, such that $j \notin X$, and $X \cup \{j\}$ is frequent. Suppose these j-items forms set Y. If $X \cup Y$ is frequent, we can conclude that any its subset also is frequent. Though superset frequency pruning reduces the search time dramatically, MAXMINER still needs many passes to get all long patterns.

DEPTHPROJECT by Agarwal, Aggrawal, and Prasad [3] also mines only long patterns. It performs a mixed depth-first/breadth-first traversal of the itemset lattice. In the algorithm, both subset infrequency pruning and superset frequency pruning are used. The database is represented as a bitmap. Each row in the bitmap is a bitvector corresponding to a transaction, each column corresponds to an item. The number of rows is equal to the number of transactions, and the number of columns is equal to the number of items. A row has a 1 in the ith position if corresponding transaction contains the item i, and a 0 otherwise. Figure 1 (a) shows an example for bitmap representation of a transaction

database. The count of an itemset is the number of rows that have 1's in all corresponding positions. For instance, the count of BCD is 3 since row 2, 4 and 5 have 1's in position B, C and D. By carefully designed counting methods, the algorithm significantly reduces the cost for finding support counts. Experimental results in [3] show that DepthProject outperforms MaxMiner by at least an order of magnitude.

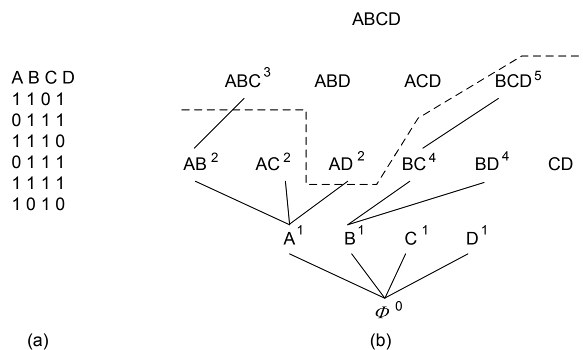


Figure 1: Bitmap representation and depth-first search

In [7], Burdick, Calimlim, and Gehrke extend the idea in DepthProject and give an algorithm called Mafia to mine maximal frequent itemsets. Similar to DepthProject, their method also uses a bitmap representation, where the count of an itemset is based on the column in the bitmap (the bitmap is called "vertical bitmap"). As an example, in Figure 1 (a), the bitvectors for items B, C, and D are 111110, 011111, and 110110, respectively. To get the bitvectors for any itemset, we only need to apply the bitvector *and*-operation $\otimes$ on the bitvectors of the items in the itemset. For above example, the bitvector for itemset BC is $111110 \otimes 011111$, which equals 011110, while the bitmap for itemset BCD can be calculated from the bitmaps of BC and D, i.e., $011110 \otimes 110110$, which is 010110. The count of an itemset is the number of 1's in its bitvector. Mafia is a depth-first algorithm. Figure 1 (b) shows the the sequence of itemsets tested for frequency given a minimum support of 50% on dataset in Figure 1 (a). The testing order is indicated by the number on the top-right side of the itemsets. Besides subset infrequency pruning and superset frequency pruning, some other pruning techniques are also used in Mafia. As an example, the support of an itemset $X \cup Y$ equals the support of X, if and only if $x \otimes Y = X$ This is the case if the bitvector for Y has a 1 in every position that the bitvector for X has 1. The last condition is easy to test. This allows us to conclude without counting that $X \cup Y$ also is frequent. The technique is called *Parent Equivalence Pruning* in [7].

Genmax, proposed by Gouda and Zaki [12], takes a novel approach to maximality testing. Most methods, including MaxMiner, use a variant of the algorithm in [8] and find the maximal elements among n sets in time $O(\sqrt{n} \log n)$. Gouda and Zaki use a novel technique called *progressive focusing.* This technique, instead of comparing an newly found frequent itemsets (FI's) with all maximal frequent itemsets found so far, maintains a set of *local* maximal frequent itemsets, LMFI's. The newly found FI is firstly compared with itemsets in LMFI. Most non-maximal FI's can be detected by this step, thus reducing the number of subset tests. Genmax also uses a vertical representation of the database. However, for each itemset, Genmax stores a *transaction identifier set*, or TIS, rather than a bitvector. The cardinality of an itemset's TIS equals its support. The TIS of itemset $X \cup Y$ can be calculated from the intersection of the TIS's of X and Y. Experimental results show that Genmax outperforms other existing algorithms on some types of datasets. For more information, see [12].

1.1 Contributions

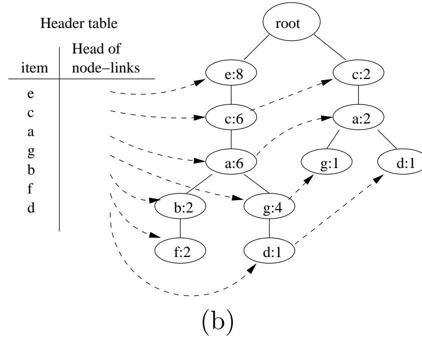
In this paper, we first introduce Fpmax, an extension of the *FP-growth* method, for mining MFI's only. During the mining process, an FP-tree (a trie structure) is used to store the frequency information of the whole dataset. To test if a frequent itemset is maximal, another trie structure, called a *Maximal Frequent Itemset tree* (MFI-tree), is utilised to keep track of all maximal frequent itemsets. This structure makes Fpmax effectively reduce the search time and the number of subset testing operations.

Since it is not to be expected that one single approach will be suitable for all types of data, we analyze the behaviour of algorithms Mafia, Genmax and Fpmax, under various types of data. We validate our analysis through careful experimentation with synthetic data, in which the parameters influencing the data characteristics are easily tunable. We then turn the conclusions into predictions of the performance of each of the three methods for specific data characteristics. By testing these predictions of real dataset, we find that they are valid in most cases.

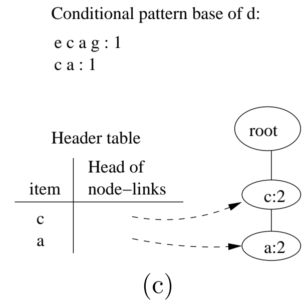
Experimental results also show that Fpmax is competitive. For certain types of data, it outperforms Mafia and Genmax. Fpmax also has a very good scalability.

$a\ b\ c\ e\ f\ o$
$a\ c\ g$
$e\ i$
$a\ c\ d\ e\ g$
$a\ c\ e\ g\ l$
$e\ j$
$a\ b\ c\ e\ f\ p$
$a\ c\ d$
$a\ c\ e\ g\ m$
$a\ c\ e\ g\ n$

(a)



(b)



(c)

Figure 2: An Example FP-tree (minsup=2)

1.2 Overview

The rest of the paper is organized as follows. Section 2 gives a brief introduction of the FP-tree and the *FP-growth* method of [9], and then introduces the trie structure for storing MFI's. The FPMAX algorithm is also given in this section. In section 3 we analyze the influence of data characteristics on the performance of MAFIA, GENMAX and FPMAX. Experimental results on synthetic dataset are given to validate our analysis. After we draw our conclusions for predicting the performance of the three methods, experimental results on real datasets are given and we found the conclusions are valid in most cases. Conclusions and future work are given in Section 4.

2 Discovering MFI's by FPMAX

2.1 FP-tree and *FP-growth* method

APRIORI and its variants repeatedly scan the database and check the frequency of candidate itemsets by pattern-matching. This is costly especially if there are prolific frequent patterns, long patterns, or quite low minimum support thresholds.

In the aforementioned FP-growth method [9], a novel data structure, the FP-tree (Frequent Pattern tree), is used. The FP-tree is a compact data structure for storing all necessary information about frequent itemsets in a database. Every branch of the FP-tree represents a frequent itemset, and the nodes along the branch are ordered decreasingly by the frequency of the corresponding item, with leaves representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping itemsets are represented by sharing prefixes of the corresponding branches.

The FP-tree has a header table associated with it. Single items and their count are stored in the header table in decreasing order of frequency. A row in the header table also contains the head of a list that links all the corresponding nodes of the FP-tree.

Compared with APRIORI and its variants which need several database scans, the *FP-growth* method only needs two database scans when mining all frequent itemsets. In the first scan, all frequent items are found. The second scan constructs the first FP-tree which contains all frequency information of the original dataset. Mining the database then becomes mining the FP-tree. Figure 2 (a) shows a database example. After the first scan, all frequent items are inserted in the hearder table of an initial FP-tree. Figure 2 (b) shows the first FP-tree constructed from the second scan.

The FP-growth method relies on the following principle: if X and Y are two itemsets, the support of itemset $X \cup Y$ in the database is exactly that of Y in the restriction of the database to those transactions containing X. This restriction of the database is called the *conditional pattern base* of X. Given an item in the header table, the growth method constructs a new FP-tree corresponding to the frequency information in the sub-dataset of only those transactions that contain the given item. Figure 2(c) shows the conditional pattern base and the FP-tree for item $\{d\}$. This step is applied recursively, and it stops when the resulting smaller FP-tree contains only one single path. The complete set of frequent itemsets is generated from all single-path FP-trees. More details about the construction of FP-tree and FP-growth method can be found in [9].

2.2 FPMAX: Mining MFI's

We extend the FP-growth method and get algorithm FPMAX described in Figure 3. Like FP-growth, algorithm FPMAX is also recursive. In the initial call, an FP-tree is constructed from the first scan of the database. A linked list *Head* contains the

items that form the conditional base of the current call. Before recursively calling Fpmax, we already know that the set containing all items in *Head* and the items in the FP-tree is not a subset of any existing MFI. If there is only one single path in the FP-tree, this single path together with *Head* is an MFI of the database. In line 2, we use the MFI-tree data stucture to keep track of all MFI's. If the FP-tree is not a single-path tree, then for each item in the header-table, the item is appended to *Head*, and line 7 calls function *subset_checking* to check if the new *Head* together with all frequent items in the *Head*-conditional pattern base is a subset of any existing MFI. If not, Fpmax will be called recursively. The data structure MFI-tree and function *subset_checking* will be explained shortly.

```
Procedure FPMAX(T)
Input:  T:  an FP-tree
Global:
  MFIT: an MFI-tree.
  Head:  a linked list of items.
Output:  The MFIT that contains all
MFI's
Method:
1. if T only contains a single path P
2.    insert Head ∪ P into MFIT
3. else for each i in Header-table of T
4.    Append i to Head;
5.    Construct the Head-pattern base
6.    Tail = {frequent items in base}
7.    subset_checking(Head ∪ Tail);
8.    if Head ∪ Tail is not in MFIT
9.     construct the FP-tree T_Head;
10.    call FPMAX(T_Head);
11.   remove i from Head.
```

Figure 3: Algorithm Fpmax

Since Fpmax is a depth-first algorithm, it's straightforward to prove the following lemma.

Lemma 1 *In* Fpmax, *any frequent itemset cannot be a subset of any frequent set generated later, i.e., it is either maximal frequent set or a subset of some existing MFI's.*

By this lemma and the correctness of *FP-growth* method, we can conclude that Fpmax returns all and only the maximal frequent itemsets in the given dataset.

By replacing line 2 in Figure 3 with "insert Head ∪ P and its support into MFIT" and minor change of data structure MFI-tree, Fpmax can return all MFI's and their supports.

2.3 MFI-Tree

How should we test if a frequent itemset is maximal or not? In [8], an algorithm is introduced for extracting all maximal elements in a set of sets. If there are n sets, then getting all maximal sets takes at least $O(\sqrt{n}\log n)$ time. In Fpmax, a frequent itemset can be a subset only of an already discovered MFI. In other words, if a frequent itemset is not a subset of any existing MFI, it is a new MFI. Therefore, a special structure can be used to do the subset-testing more efficiently. We introduce the *Maximal Frequent Itemset tree* (MFI-tree) as the special data structure to store all MFI's.

The MFI-tree resembles an FP-tree. It has a root labelled with "root". Children of the root are item prefix subtrees. Each node in the subtree has two fields: item-name and node-link. All nodes with same item-name are linked together. The node-link points to the next node with same item-name. A header table is constructed for items in the MFI-tree, the item order in the table is same as the item order in the first FP-tree constructed from the first scan of the database. Each entry in the header table consists of two fields, item-name and head of a node-link. The node-link points to the first node with the same item-name in the MFI-tree.

In Fpmax, a newly discovered frequent itemset is inserted into the MFI-tree, unless it is a subset of an itemset already in the tree. Due to the lack of space, we omit the algorithm for constructing MFI-tree here. We take the FP-tree in Figure 2 as an example to see how algorithm Fpmax and the construction of the MFI-tree works.

For the database in Figure 2(a), after the construction of the first FP-tree by FP-growth method, the method is called recursively for each frequent item i. In this example, the FP-trees corresponding to all $\{i\}$-conditional pattern base each contain only a single branch, and therefore the recursion stops. In the following table we summarize the $\{i\}$-conditional pattern base for each frequent item i and its corresponding conditional FP-tree.

item	condi. pattern base	conditional FP-tree
d	$\{(ecag:1),(ca:1)\}$	$\{(c:2,a:2)\}$
f	$\{(ecab:2)\}$	$\{(e:2,c:2,a:2,b:2)\}$
b	$\{(eca:2)\}$	$\{(e:2,c:2,a:2)\}$
g	$\{(eca:4),(ca:1)\}$	$\{(c:5,a:5,e:4)\}$
a	$\{(ec:6),(c:2)\}$	$\{(c:8,e:6)\}$
c	$\{(e:6)\}$	$\{(e:6)\}$
e	$\emptyset$	$\emptyset$

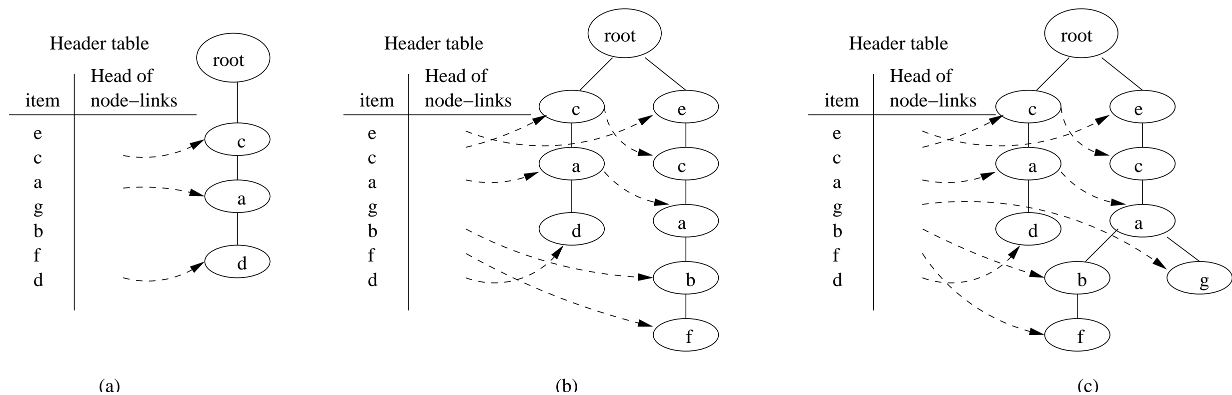


Figure 4: Construction of Maximal Frequent Itemset Tree

Figure 4 illustrates the construction of the MFI-tree for the example of Figure 2. Start from the first FP-tree T in Figure 2(b), by calling FPMAX(T). Since the T contains more than one path, a bottom-up search has to be done. For item d, its conditional FP-tree only has one single path, so we get the first frequent itemset $\{c, a, d\}$. Obviously this set is maximal, so it is inserted into the MFI-tree directly (Figure 4 (a)). Note that in Figure 4 the header table of the MFI-tree is the same as that of the FP-tree in Figure 2, constructed from the database. For item f, the only f-conditional frequent itemset is $\{e, c, a, b, f\}$, and since there is no link-chain for f, this set is also maximal. We then insert $\{e, c, a, b, f\}$ into the MFI-tree (Figure 4 (b)). For item b, the only $\{b\}$-conditional itemset is $\{e, c, a, b\}$, and by calling *subset_checking*, we determine that $\{e, c, a, b\}$ is a subset of an existing MFI, so it will not be inserted into the MFI-tree. Next, the $\{g\}$-conditional frequent itemset $\{e, c, a\}$ will be inserted into the MFI-tree (Figure 4 (c)). No itemsets in the conditional bases for $\{a\}$, $\{c\}$, or $\{e\}$ are maximal, no new MFI's will be inserted into the MFI-tree. Every branch of the MFI-tree forms an MFI. Thus the MFI's are $\{c, a, d\}$, $\{e, c, a, b, f\}$, $\{e, c, a, g\}$.

2.4 Implementation of subset testing

In FPMAX, function *subset_checking* is called to check if $Head \cup Tail$ is a subset of some MFI in the MFI-tree. If $Head \cup Tail$ is a subset of some MFI, then any frequent itemset generated from the FP-tree corresponding to $Head$ could not be maximal, and thus we can stop mining MFI's for $Head$. By calling *subset_checking*, we do superset frequency pruning.

Note that before and after calling *subset_checking*, if $Head \cup Tail$ is not subset of any MFI, we still don't know if $Head \cup Tail$ is frequent or not. By constructing the FP-tree for $Head$ from the conditional pattern base of $Head$, if the FP-tree only has a single path, we can conclude that $Head \cup Tail$ is frequent. Since $Head \cup Tail$ was not a subset of any previously discovered MFI, it's a new MFI and will be inserted to the MFI-tree.

To do subset testing, one possibility is to always compare a set with the MFI's in the MFI-tree. However, we can do better. We found that most frequent sets are subsets of the latest MFI inserted into the MFI-tree. Therefore, each time we insert a new MFI into the MFI-tree, we keep a copy of this most resent MFI, any new frequent set will be compared with the copy first. Only if the new set is not subset of the copy, the new set will be compared with the MFI's in the MFI-tree.

By using the header-table in the MFI-tree, a set S is not necessarily compared with all MFI's in the MFI-tree. First, S is sorted according to the order of items in header table. Suppose the sorted S is $\langle i_1, i_2, \ldots, i_n \rangle$. From the header table, we find the node list for i_n. For each node in the the list, we test if S is a subset of the ancestors of that node. Note that both sets are ordered according to the header table, so this subset test can be done in linear time. The function *subset_checking* returns *false* if $\{i_1, i_2, \ldots, i_{n-1}\}$ is not a subset of any set in the MFI-tree.

3 Data Characteristics and Performance

Previously, in [3], Agarwal et al. have shown that DEPTHPROJECT achieves more than one order of magnitude speedup over MaxMiner [6]. In [7], the performance numbers of MAFIA show that MAFIA outperforms DEPTHPROJECT by a factor of three to

five. In the latest paper about mining MFI's, Gouda and Zaki [12] claim that MAFIA is the current best method for mining a *superset* of all MFI's, and that GENMAX is the current best method for enumerating the *exact* set of MFI's.

In the present study, we wish to reach an understanding of how the data characteristics influence the performance of MAFIA, GENMAX, and our new algorithm FPMAX. We first analyze the mining time used by the three algorithms.

We can divide the mining task in two parts. The first part consists of mining a superset of FI's, and the second part is for pruning out non-maximal FI's. For FPMAX, the time resources in the first part are invested in the construction of an FP-tree for concisely representing the database, and then extracting FI's from the FP-tree using the FP-growth method.

In the second part of the mining task, in order to extract the maximal FI's, the FPMAX algorithm has to perform a large number of subset tests. Suppose there are n items in header table. Then we know there are at most $C_n^{\lfloor n/2 \rfloor}$ maximal frequent itemsets. If we construct an MFI-tree for all these MFI's, the tree has height $\lfloor n/2 \rfloor$. In the first level, there are $C^1_{\lceil n/2 \rceil+1}$ nodes, in the second level, there are $C^2_{\lceil n/2 \rceil+2}$ nodes, in the ith level, there are $C^i_{\lceil n/2 \rceil+i}$ nodes, and in the last level, the $\lfloor n/2 \rfloor$th level, there are $C_n^{\lfloor n/2 \rfloor}$ nodes. Thus, the total number of nodes in the tree is

$$\sum_{i=1}^{\lfloor n/2 \rfloor} C^i_{\lceil n/2 \rceil+i} \tag{1}$$

This is also an upper bound on the number of subset tests needed in constructing the MFI-tree. Similar observations apply to the size of the FP-tree.

In the first part of the mining task both GENMAX and MAFIA, construct a column-wise representation of the bitmap representation of the database. To extract the FI's from the columns, MAFIA has to compute a number of bitvector *and*-operations, and GENMAX does TIS intersections. If there are n items in the dataset, in the worst case, if the length of all MFI's is $n/2$, by a similar analysis as above, the total number of bitvector operations or TIS intersections could be equal to (1). However, a dense dataset (most columns have 1's) and a sparse dataset (most columns have 0's) having the same number of maximal FI's, will require the same number of bitvector operations or TIS intersections.

Now let's see how the parameters of the synthetic data generator at [1] influence the performance of the three algorithms. The adjustable parameters include

- the average size of the transactions, also called *average transaction length*, ATL, and
- the average size of the maximal potentially large itemsets, also called *average pattern length*, APL.

We can think of the ATL as influencing the density of the dataset. The APL, on the other hand, determines the *average length* of MFI's that the dataset will contain. Thus, a long ATL generates a dense dataset, and a long APL gives long average MFI's. This gives us four categories of data.

1. **Short ATL, short APL.** In this dataset we can expect that each transaction will be fairly short, and that the MFI's will also be short. For FPMAX, this could result in a costly bushy FP-tree.

   Now, if the minimum support is high, there might be only relatively few maximal FI's. This means that FPMAX spent a considerable time effort to construct an FP-tree, from which only a small set of MFI's will be extracted. In this case, we can expect GENMAX and MAFIA to be more efficient, since the ATL will not influence the time computing bitvector operations and set intersections.

   However, in the case where the minimum support is low, there might be numerous maximal FI's. Then the time FPMAX invested in the FP-tree will pay off, since the size of the output (the MFI's represented as an MFI-tree) will also be large. Now we expect FPMAX to outperform the other two algorithms.

2. **Short ATL, long APL.** Here we expect that the transactions in the dataset are short, while the average length of the MFI's is close to ATL. For FPMAX, this will result in a small FP-tree. Contrary to the first case, we can now efficiently extract the MFI-tree from the small FP-tree. We can also extract a relatively large MFI-tree from the small FP-tree. GENMAX and MAFIA on the other hand, will still need to do all the bitvector operations and set intersections they did in a dataset with short APL. Thus, we can expect that FPMAX is the algorithm of choice for data with short ATL and long APL.

3. **Long ATL, short APL.** Now the transactions in the dataset are long. For FPMAX, it will result in a very bushy and tall FP-tree. This will require more space and time than in the case of short ATL. Now, if the minimum support is

high and we get small size MFI-tree, FPMAX will not be efficient, GENMAX or MAFIA will perform better. On the other hand, if the minimum support is low and we have a large output, FPMAX may outperform GENMAX and MAFIA.

4. **Long ATL, long APL.** For FPMAX, both the FP-tree and the MFI-tree could be very large, which means we need more time to construct the FP-tree, and we need more comparisons to construct the MFI-tree. For this type of data, if the cost of bitvector *and*-operations in MAFIA is less than that of TIS intersections in GENMAX, MAFIA could be the best, otherwise, GENMAX is the best.

3.1 Experiments on Synthetic data

To test the accuracy of the analysis above, we run three algorithms on synthetic datasets. The source codes for MAFIA and GENMAX were provided by their authors, and in particular the source code for MAFIA is the latest version that mine exact MFI's without post-processing. We use the application from [1] to generate synthetic datasets. For all datasets in this section, the number of transactions was fixed at 100,000, and the number of items was fixed at 1000. All experiments were performed on a 1Ghz Pentium III with 512 MB of memory running RedHat Linux 7.3. All timings in the figures are CPU time.

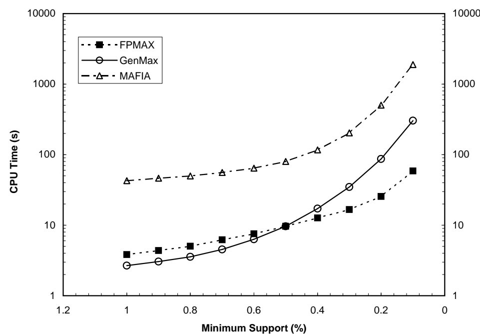


Figure 5: ATL=20, APL=20

The results of the first set of experiments are shown in Figure 5. We ran the algorithms on a short ATL of 20, and short APL also of 20. We see that FPMAX and GENMAX outperform MAFIA five to ten times. GENMAX is faster than FPMAX when the minimum support is high, and when minimum support is low, FPMAX is faster. For minimum support 0.1%, FPMAX is five times faster than GENMAX, while for minimum support 1%, GENMAX is only about one and a half times faster than FPMAX.

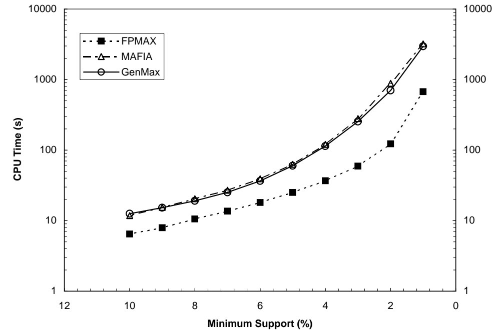


Figure 6: ATL=20, APL=100

The synthetic data for the second set of experiments, displayed in Figure 6, has a short (20) ATL and a long (100) APL. In this case, the performance of MAFIA and GENMAX is almost the same. FPMAX is clearly the most efficient on this dataset. FPMAX outperforms the other two by factor of at least two, both for high and low minimum support.

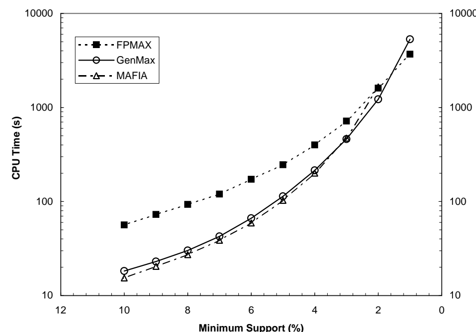


Figure 7: ATL=100, APL=20

Figure 7 shows a totally different figure for algorithms on dataset with a long ATL (100) and a short APL (20). We can see that FPMAX is slower than the other algorithms most of the time. On this type of data, MAFIA or GENMAX is the best for high support, while FPMAX tends to be faster than the other two for low support.

We also run the algorithms on the dataset with long (100) ATL and long (100) APL. On this type of data, from Figure 8 we can see that MAFIA seems to be the best, although GENMAX performs well too. FPMAX seems to be slow at all time, even when the support is low.

For the next two experiments we fixed a low minimum support of 1%.

Figure 9 shows the result for the datasets generated by fixing ATL to 20 and varying APL from 20 to 100. In these experiments, FPMAX has better performance than the other two algorithms. MAFIA

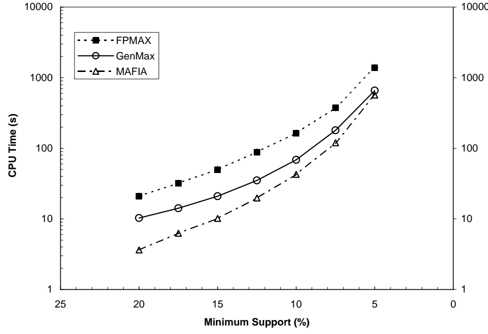


Figure 8: ATL=100, APL=100

and GENMAX have same tendency, although GENMAX is faster than MAFIA.

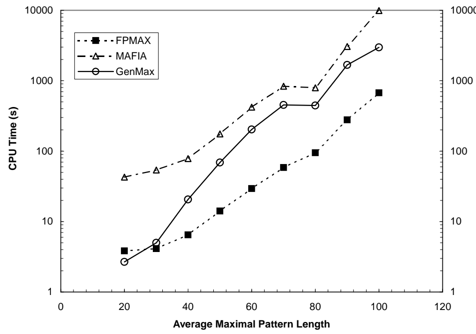


Figure 9: ATL=20, MinSup=1%

Then, we generated the datasets for the second set of experiments by fixing APL to 20 and varying APL from 20 to 100. Figure 10 shows the result. We can see that FPMAX is slightly faster than GENMAX, while MAFIA is distinctly slower than the other two algorithms.

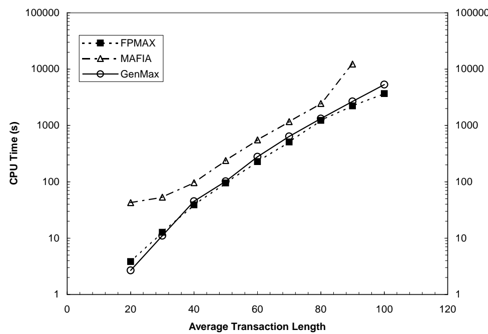


Figure 10: APL=20, MinSup=1%

The results of our experiments are summarized in Figure 11, which gives the best algorithm for the various types of data.

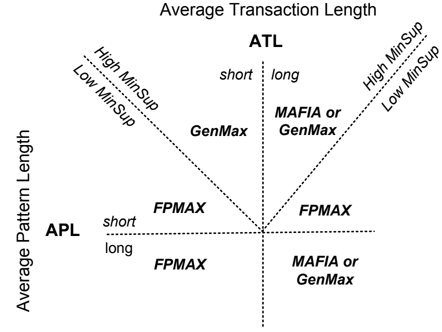


Figure 11: Best algorithms for different types of data.

3.2 Experiments on Real Datasets

Next, we ran the programs on real datasets downloaded from [2]. We used datasets *chess*, *connect-4*, *mushroom*, and *pumsb**. The *chess* and *connect-4* datasets are compiled from game state information. The *mushroom* dataset consists of records describing the characteristics of various mushroom species, and the *pumsb** dataset is produced from census data of Public Use Microdata Sample (PUMS). All these real datasets are used in [6]. Many other papers [3, 7, 12] also use these datasets to test and compare their algorithms. These real datasets are all very dense, so a large number of MFI's can be mined even for very high values of support.

Figures 12 to 15 show the performance of the three algorithms on these real datasets. Figure 12 shows the experimental results on *mushroom*. Here FPMAX outperforms the other algorithms, for all levels of minimum support. In dataset *mushroom*, the average transaction length is 23, and the average MFI's length ranges from 8 to 19 for minimum support 10% to 0.1%. We can categorize this dataset as having short ATL, and long APL. Figure 11 shows that FPMAX has the best performance.

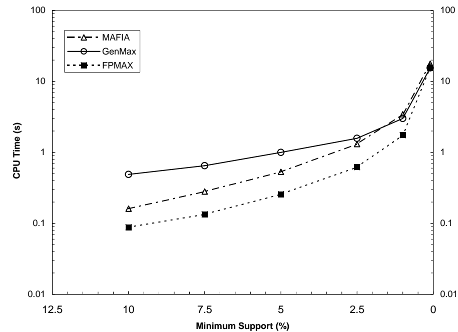


Figure 12: dataset *mushroom*

The results for the *chess* dataset is shown in Figure 13. The ATL of the dataset is 37 while the av-

erage length of MFI's is up to 12, which means both ATL and APL are long, so Fpmax is not expected to perform well on this dataset. Also, here Mafia needs more work in the bitverctor *and*-operations, so Genmax is the best algorithm for *mushroom* dataset.

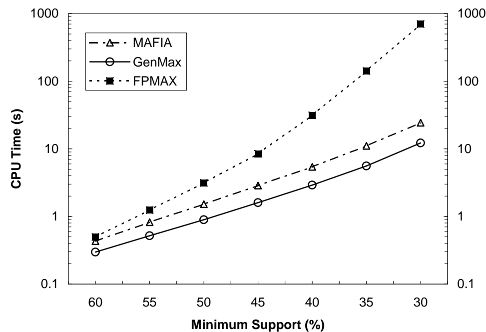


Figure 13: dataset *chess*

In the dataset *connect-4*, though the ATL of this dataset is 43, which is fairly long, and the average length of the MFI's is 9 to 21 for minimum support 90% to 10%, which also is fairly long, Fpmax is nevertheless the best algorithm for high minimum support, and Genmax is the best for low minimum support. This result doesn't fit the rule in Figure 11. We conjecture that *connect-4* has a skewed distribution, different from the binomial or exponential distributions that are used to generate the synthetic datasets [5]. By checking the size of FP-tree and the number of subset tests needed in constructing the MFI-tree, we found that they are both far smaller than those for synthetic dataset with ATL equal to 40.

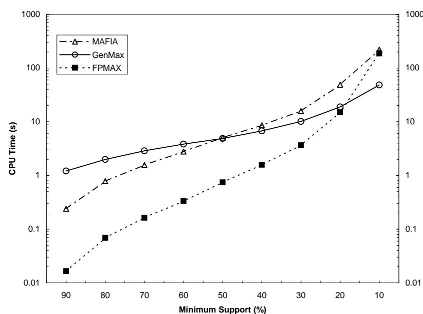


Figure 14: dataset *connect-4*

The *pumsb** dataset is also skewed, long (50) ATL and long APL (average length of MFI's is 7 to 14 for minimum support 35% to 10%). As we can see from Figure 15, for high minimum support, Mafia is the most efficient, followed by Fpmax, and then Genmax.

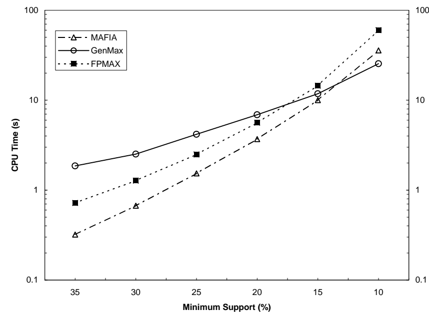


Figure 15: dataset *pumsb**

Based on the experiments with datasets *connect-4* and *pumsb**, it seems that the predictions in Figure 11 do not hold in the case when both ATL and APL are long.

3.3 Scalability of the algorithms

To test the scalability of three algorithms, we also run the programs on both synthetic and real datasets, while varying the number of transactions in the datasets.

For the synthetic datasets, we set ATL to 10, and APL to 4, and varied the number of transactions from 100,000 to 1,000,000. We chose a minimum support of 0.05%, because for this level Fpmax and Genmax use almost the same amount of CPU time. Figure 16 shows that the mining time increases almost linearly for all three algorithms, while Mafia and Genmax show a steeper increase than Fpmax.

The steeper increase for Mafia and Genmax in Figure 16 is not accidental. For synthetic datasets, if we increase the number of transactions and keep other parameters unchanged, we can expect more similar transactions, while the number of MFI's will not increase much. For Fpmax, adding transactions similar to the existing ones will not increase the sizes of the FP-tree and MFI-tree much, while it does increase the cost of set intersections because the sets now become long. In the extreme case, if we increase the dataset by adding transactions equal to those that are already in the dataset, we can expect that the CPU time for Fpmax will remain unchanged, while it will increase for Genmax and Mafia. Figure 17 shows the result on dataset which is generated by duplicating the real dataset *connect-4* two to ten times. From the figure, we can see that the line for Fpmax is flat while the CPU time for the other two algorithms increase rapidly. From Figure 17 we can also see that bitvector *and*-operations in Mafia needs more time than TIS intersections in Genmax.

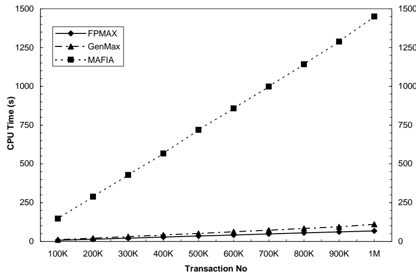


Figure 16: Scaled Datasets

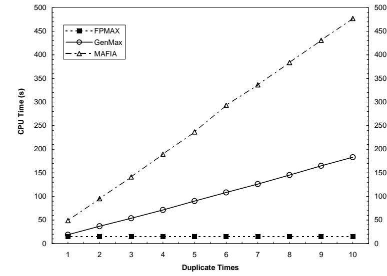


Figure 17: Duplicated Datasets

4 Conclusions

This paper studies the performance of algorithms for mining maximal frequent itemsets. We first reviewed two existing algorithms, GENMAX and MAFIA. We then give our new algorithm, FPMAX, which is an extension of FP-growth method [9]. A trie structure, the MFI-tree, is also introduced as a part of FPMAX to keep track of all MFI's.

In order to understand the performance on datasets with different characteristics, we analyzed the likely behaviour of GENMAX, MAFIA and FPMAX. Numerous experiments on synthetic datasets were done to validate our analysis.

From the experiments, we also see that FPMAX outperforms GENMAX and MAFIA in many cases, especially for datasets with short average transaction length and long average pattern length. The scalability of FPMAX is also studied, and found to be good.

Though the experimental results on real datasets also show that our conclusions are valid to some degree, they don't seem to hold for some skewed real datasets. We are currently undertaking further analysis and experiments in order to obtain an understanding of the impact of the skewness of the data.

References

[1] `http://www.almaden.ibm.com/cs/quest/syndata.html`.

[2] `http://www.almaden.ibm.com/cs/people/bayardo/resources.html`.

[3] Ramesh C. Agarwal, Charu C. Aggarwal and V. V. V. Prasad, Depth first generation of long patterns, In *Knowledge Discovery and Data Mining*, pages 108-118, 2000.

[4] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceeding of Special Interest Group on Management of Data*, pages 207–216, 1993.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceeding of Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, Sept. 1994.

[6] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceeding of Special Interest Group on Management of Data*, pages 85–93, Seattle, WA, June 1998.

[7] Doug Burdick, Manuel Calimlim, and Johannes Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Heidelberg, Germany, April 2001.

[8] D. Yellin. An algorithm for bynamic subset and intersection testing. In *Theoretical Computer Science*, Vol. 129: 397-406, 1994.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceeding of Special Interest Group on Management of Data* , pages 1–12, Dallas, TX, May 2000.

[10] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. In *Data Mining and Knowledge Discovery*, Vol. 1, 3(1997), pages 241-258.

[11] H. Toivonen. Sampling large databases for association rules. In *Proceeding of Int. Conf. Very Large Data Bases*, pages 134–145, 1996

[12] K. Gouda, M.J. Zaki. Efficiently Mining Maximal Frequent Itemsets In *1st IEEE International Conference on Data Mining (ICDM)*, pages 163–170, San Jose, November 2001.