MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases

Doug Burdick, Manuel Calimlim, Johannes Gehrke
Department of Computer Science, Cornell University

Abstract

We present a new algorithm for mining maximal frequent itemsets from a transactional database. Our algorithm is especially efficient when the itemsets in the database are very long. The search strategy of our algorithm integrates a depth-first traversal of the itemset lattice with effective pruning mechanisms.

Our implementation of the search strategy combines a vertical bitmap representation of the database with an efficient relative bitmap compression schema. In a thorough experimental analysis of our algorithm on real data, we isolate the effect of the individual components of the algorithm. Our performance numbers show that our algorithm outperforms previous work by a factor of three to five.

1 Introduction

The association rule problem is a very important problem in the data-mining field with numerous practical applications, including consumer market-basket analysis, inferring patterns from web page access logs, and network intrusion detection [9, 15, 18]. The association rule model and the support-confidence framework were originally proposed by Agrawal et al. [2, 3].

Let **I** be a set of items (we assume in the remainder of the paper without loss of generality **I** = {1, … , N}). We call $X \subseteq$ **I** an itemset, and we call *X* a *k-itemset* if the cardinality of itemset *X* is *k*. Let database **T** be a multiset of subsets of **I**, and let *support(X)* be the percentage of itemsets *Y* in **T** such that $X \subseteq Y$. Informally, the support of an itemset measures how often *X* occurs in the database. If *support(X)* ≥ *minSup*, we say that *X* is a *frequent* itemset, and we denote the set of all frequent itemsets by **FI**. If *X* is frequent and no superset of *X* is frequent, we say that *X* is a *maximally* frequent itemset, and we denote the set of all maximally frequent itemsets by **MFI**.

The process for finding association rules has two separate phases [3]. In the first phase, we find the set of frequent itemsets (**FI**) in the database T. In the second step, we use the set **FI** to generate "interesting" patterns, and various forms of interestingness have been proposed [8, 9, 17, 22, 27, 29, 30]. In practice, the first phase is the most time-consuming [3]. Smaller alternatives to **FI** that still contain enough information for the second phase have been proposed including the set of frequent closed itemsets **FCI** [20, 21, 33]. An itemset *X* is *closed* if there does not exist an itemset *X'* such that $X' \supset X$ and *t(X) = t(X')*, with *t(Y)* defined as the set of transactions that contain itemset *Y*. It is straightforward to see that the following relationship holds: **MFI** ⊆ **FCI** ⊆ **FI**.

The set **MFI** is orders of magnitude smaller than the set **FCI**, which itself is orders of magnitude smaller than the set **FI**. Wherever there are very long patterns (patterns containing many items) are present in the data, it is often impractical to generate the entire set of frequent itemsets or closed itemsets [7]. Also, there are applications where the set of maximal patterns is adequate, such as combinatorial pattern discovery in biological applications [23].

There is much research on methods for generating all frequent itemsets efficiently [4, 5, 6, 10, 11, 13, 19, 25, 26, 28, 31] or just the set of maximal frequent itemsets [1, 7, 12, 16, 32]. When the frequent patterns are long (more than 15 to 20 items), **FI** and even **FCI** become very large and most traditional methods count too many itemsets to be feasible. Straight Apriori-based algorithms count all of the 2^k subsets of each k-itemset they discover, and thus do not scale for long itemsets. Other methods use "lookaheads" to reduce the number of itemsets to be counted. However, most of these algorithms use a breadth-first approach, i.e. finding all k-itemsets before considering (k+1) itemsets. This approach limits the effectiveness of the lookaheads, since useful longer frequent patterns have not yet been discovered. Recently, the merits of a depth-first approach have been recognized [1].

The database representation is also an important factor in the efficiency of generating and counting

itemsets. *Generating* the itemset $Z = (X \bigcup Y)$ refers to creating $t(Z) = t(X) \cap t(Y)$, and *counting* is the process of determining *support(Z)* in **T**. Most previous algorithms use a horizontal row layout, with the database organized as a set of rows and each row representing a transaction. The alternative *vertical* column layout associates with each item *X* a set of transaction identifiers (tids) for the set *t(X)* [4,10,11,26]. The vertical representation allows simple and efficient support counting (see Section 4).

In this paper, we take a systems approach to the problem of finding maximal large itemsets. We propose an efficient algorithm called **MAFIA** (**MA**ximal **F**requent **I**temset **A**lgorithm) that integrates a variety of old and new algorithmic ideas into a practical algorithm. MAFIA assumes that the entire database (and all data structures used for the algorithm) completely fit into main memory. With the size of current main memories reaching gigabytes and growing, many moderate-sized to large databases will soon become completely memory-resident. Considering the computational complexity involved in finding long patterns even in small databases, this assumption is not very limiting in practice. Since all algorithms for finding association rules, including algorithms that work with disk-resident databases, are CPU-bound, we believe that our study sheds light on the most important performance bottleneck.

In a thorough experimental evaluation, we first quantify the effect of each individual component on the performance of the algorithm. We then compare the performance of MAFIA against DepthProject, the most efficient previously known algorithm for finding maximal frequent itemsets [1]. Our results using some of the standard machine learning benchmark datasets indicate that MAFIA outperforms DepthProject by a factor of three to five on average.

The organization of the paper is as follows. Section 2 discusses the conceptual ideas of the itemset lattice and subset tree we base our approach on. In Section 3, we describe the basic depth-first algorithm and methods to prune the search space. Section 4 describes the database representation and counting methods, including compression techniques that significantly speed up the counting process. We present an analysis of the components of our algorithm and compare its performance to the DepthProject Algorithm in Section 5. We conclude in Section 6 with a discussion of future work.

2 Preliminaries

In this section, we describe the conceptual framework of the item subset lattice. Assume there is a total lexicographic ordering $\leq_L$ of the items **I** in the database. If item *i* occurs before item *j* in the ordering, then we denote this by $i \leq_L j$. This ordering can be used to enumerate the item subset lattice, or a partial ordering over the powerset **S** of items **I**. We define the partial order $\leq$ on $S1, S2 \in \mathbf{S}$ such that $S1 \leq S2$ if $S1 \subseteq S2$. If $S1 \not\subset S2$, there is no relationship in the ordering.

Figure 1 is a sample of a complete subset lattice for four items. The top element in the lattice is the empty set and each lower level k contains all of the k-itemsets. The k-itemsets are ordered lexicographically on each level and all children are generated from the lexicographically earliest subset in the previous level. Generating children in this manner reduces the lattice to the *lexicographic subset tree* originally presented by Rymon [24] and adopted by both Agarwal [1] and Bayardo [7].

The itemset identifying each node will be referred to as the node's *head*, while possible extensions of the node are called the *tail*. In a pure depth-first traversal of the tree, the tail contains all items lexicographically larger than any element of the head. With a *dynamic reordering* scheme, the tail contains only the frequent extensions of the current node. Notice that all items that can appear in a subtree are contained in the subtree root's head *union* tail (*HUT*), a set formed by combining all elements of the head and tail.

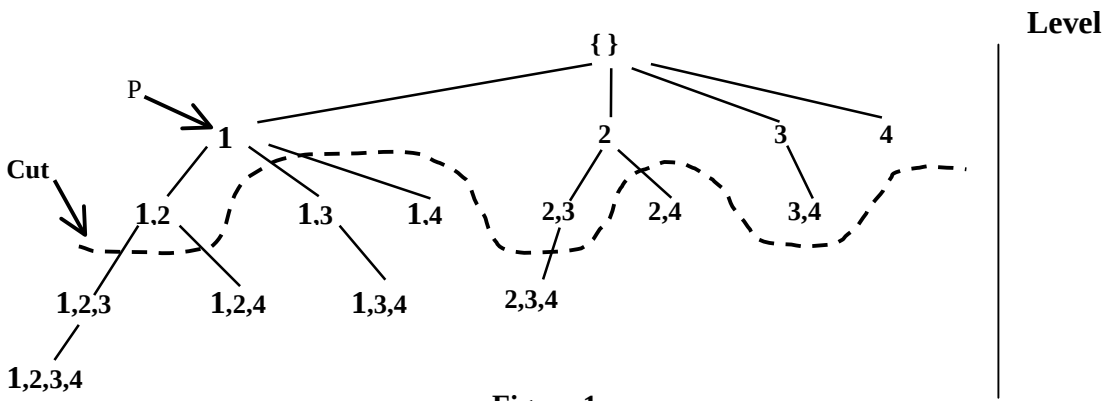


Figure 1

The problem of mining the frequent itemsets can be viewed as finding a cut through this lattice such that all elements above the cut are frequent itemsets, and all elements (subsets) below are infrequent (see Figure 1). The frequent itemsets above the cut constitute the *positive border*, while the infrequent itemsets below the cut form the *negative border*. With a simple traversal without pruning, we need to count the supports of all elements above and including the negative border. For example, in Figure 1, a cut in the lattice has been drawn in and all of the itemsets shown need to be counted except for {1,2,3,4} and {1,3,4}, since they are both below the negative border.

Consider node ***P*** in Figure 1 and the cut through the lattice. ***P***'s head is {1} and the tail (before processing) is the set {2,3,4}. Therefore, ***P***'s HUT is {1,2,3,4}. Using dynamic reordering, ***P***'s children {1,2}, {1,3}, {1,4} will be counted first. Based on the cut in Figure 1, only {1,2} is frequent; items 3 and 4 will be trimmed out of the tail of ***P*** and no new itemsets need to be counted in the subtree rooted at ***P***. In the rest of the search tree, itemsets {2,3}, {2,4}, and {3,4} would also be counted. On the other hand, a pure depth-first traversal would do extra work and compute {1,2,3}, {1,2,4}, and {2,3,4} in addition to the itemsets counted using dynamic reordering. Thus, as the size of the tree grows, dynamic reordering will trim out many branches of the search tree.

2.1 Related Work

Given this conceptual framework, we can describe the most recent approaches to the maximal frequent itemset problem. As a baseline, Apriori traverses the lattice in a pure breadth-first manner, discovering all frequent nodes at level k before moving to level (k+1); Apriori finds support information by explicitly generating and counting each node [3]. MaxMiner performs a breadth-first traversal of the search space as well, but also performs *lookaheads* to prune out branches of the tree [7]. The lookaheads involve superset pruning, using apriori in reverse (all subsets of a frequent itemset are also frequent). In general, lookaheads work better with a depth-first approach, but MaxMiner uses a breadth-first approach to limit the number of passes over the database.

DepthProject performs a mixed depth-first traversal of the tree, along with variations of superset pruning [1]. Instead of a pure depth-first traversal, DepthProject uses dynamic reordering of children nodes. With dynamic reordering, the size of the search space can be greatly reduced by trimming infrequent items out of each node's tail. Also proposed in DepthProject is an improved counting method and a projection mechanism to reduce the size of the database.

The other notable maximal pattern methods are based on graph-theoretic approaches. MaxClique and MaxEclat [32] both attempt to divide the subset lattice into smaller pieces ("cliques") and proceed to mine these in a bottom-up Apriori-fashion with a vertical data representation. However, the algorithms both rely on a pre-processing step having been performed that limits future mining flexibility. Pincer-Search also assumes a pre-processing step has taken place before the algorithm executes [16].

The VIPER algorithm has shown a method based on a vertical layout can sometimes outperform even the optimal method using a horizontal layout [26]. It uses a vertical bitvector with compression to store intermediate data during algorithm execution, while counting is performed using a vertical tid-list approach. However, VIPER returns the entire set **FI** and would not be appropriate for finding the set **MFI** if the patterns are very long. Other vertical mining methods for finding **FI** are presented by Holsheimer [14] and Savasere et al. [25]. The benefits of using the vertical tid-list were also explored by Ganti et al. [11]. An analysis of the impact of different database representations on performance can be found by Dunkel et al. [10].

3 Algorithmic Descriptions

In this section, we describe the different components of MAFIA (Figure 5) and the various pruning methods used to reduce the search space. First, we describe a simple depth-first traversal with no pruning. We use this algorithm to motivate the pruning and ordering improvements introduced in Sections 3.2 and 3.3.

3.1 Simple DFS

In the simple algorithm (see Figure 2), we traverse the lexicographic tree in pure depth-first order. At each node **n**, each element in the node's tail is generated and counted as a possible 1-extension. If the support of {**n**'s head} $\cup$ {1-extension} is less than *minSup*, then we can stop by the Apriori principle, since any itemset from that possible 1-extension would have an infrequent subset. If none of the 1-extensions leads to a frequent itemset, the node is a leaf.

When we reach a leaf in the tree, we have a candidate for entry into the MFI. However, a frequent superset of the itemset may have already been discovered. Therefore, we need to check if a superset of the candidate itemset is already in the MFI. If no superset exists, then we add the candidate itemset to the MFI. It is important to note that with the depth-first traversal, we never have to worry about removing subsets from the MFI. This is due to the fact that itemsets already inserted into the MFI will be lexicographically ordered earlier.

Algorithmic Descriptions and Pseudocode

Overview: A simple DFS of the space. Lookups in the **MFI** will only guarantee maximality and do not conduct any pruning.

```
Pseudocode: Simple (Current node C, MFI) {
  For each item i in C.tail {
    newNode = C ∪ i
    if newNode is frequent
      Simple(newNode, MFI) }
  if (C is a leaf and C.head is not in MFI)
    Add C.head to MFI
}
```

Figure 2

Overview: For each child generated, the transaction sets of the child and parent are compared. If they match, the parent can trim the tail by moving that child from the tail to the head.

```
Pseudocode: PEP (Current node C, MFI) {
  For each item i in C.tail {
    newNode = C ∪ i
    if (newNode.support == C. support)
      Move i from C.tail to C.head
    else if newNode is frequent
      PEP (newNode, MFI) }
  if (C is a leaf and C.head is not in MFI)
    Add C.head to MFI
}
```

Figure 4

Overview: A superset check is performed with *{head}* ∪ *{tail}* by exploring the leftmost branch.

```
Pseudocode: FHUT (node C, MFI, Boolean IsHUT) {
  For each item i in C.tail {
    newNode = C ∪ I
    IsHUT = whether i is the leftmost child in the tail
    if newNode is frequent
      FHUT (newNode, MFI, IsHUT) }
  if (C is a leaf and C.head is not in MFI)
    Add C.head to MFI
  if (IsHUT and all extensions are frequent)
    Stop exploring this subtree and go back up tree to
      when IsHUT was changed to True
}
```

Figure 6

Overview: Lookups in the MFI check whether the *{head}* ∪ *{tail}* is frequent for superset pruning.

```
Pseudocode: HUTMFI (Current node C, MFI) {
  name HUT = C.head ∪ C.tail;
  if HUT is in MFI
    Stop searching and return
  For each item i in C.tail {
    newNode = C ∪ I
    if newNode is frequent
      HUTMFI(newNode, MFI) }
  if (C.head is not in MFI)
    Add C.head to MFI
}
```

Figure 3

Overview: All of the components from Figures 2-4, and 6 are included along with dynamic reordering of children.

```
Pseudocode: MAFIA (C, MFI, Boolean IsHUT) {
  name HUT = C.head ∪ C.tail;
  if HUT is in MFI
    Stop generation of children and return
Count all children, use PEP to trim the tail, and reorder
  by increasing support.
  For each item i in C.trimmed_tail {
    IsHUT = whether i is the first item in the tail
    newNode = C ∪ I
    MAFIA(newNode, MFI, IsHUT) }
  if (IsHUT and all extensions are frequent)
    Stop search and go back up subtree
  if (C is a leaf and C.head is not in MFI)
    Add C.head to MFI
}
```

Figure 5

Overview: Compress bitmap **X** and all bitmaps in the tail to a smaller bitmap the size of *support(X)*

```
Pseudocode: Project(Bitmap X, node's TAIL) {
  For (each item I in TAIL) {
    Create empty bitmap I'
    For each transaction T
      if bit T of X is on
        Append bit T of I to I' }
  Create X' – a bitmap filled with ones and size of
  support(X)
  Return set of projected bitmaps I' and X'
}
```

Figure 7

3.2 Pruning Away the Tree

The simple depth-first traversal is ultimately no better than a comparable breadth-first approach, since exactly the same search space is generated and counted. To realize performance gains, we must prune out parts of the search space.

3.2.1 Parent Equivalence Pruning (PEP)

One method of pruning involves comparing the transaction sets of each parent/child pair (see Figure 4). Let x be the node **n**'s head and y be an element in the node **n**'s tail. If $t(x) \subseteq t(y)$, then any transaction containing x also contains y. This guarantees that any frequent itemset z containing x but not y has the frequent superset $(z \cup y)$. Since we only want the maximal frequent itemsets, it is not necessary to count itemsets containing x and not y. Therefore, we can move item y from the tail to the head. For node **n**, $x = x \cup y$ and item y is removed from **n**'s tail.

3.2.2 FHUT

Another type of pruning is superset pruning. We observe that at node **n**, the largest possible frequent itemset contained in the subtree rooted at **n** is **n**'s HUT (head union tail) as observed by Bayardo [7]. If **n**'s HUT is discovered to be frequent, we never have to explore any subsets of the HUT and thus can prune out the entire subtree rooted at node **n**. We refer to this method of pruning as *FHUT* (**F**requent **H**ead **U**nion **T**ail) pruning.

FHUT can be computed by exploring the leftmost path in the subtree rooted at each node. In fact, since the depth-first algorithm already explores the leftmost path, no extra computation is necessary (see Figure 6). A disadvantage of FHUT versus HUTMFI (Section 3.2.3) is that the leftmost path contains the most items of any path in the subtree. Thus, while the subtree rooted at that node is pruned, a significant portion of the tree is still generated and counted.

3.2.3 HUTMFI

There are two methods for determining whether an itemset x is frequent: direct counting of the support of x, and checking if a superset of x has already been declared frequent; FHUT uses the former method. The latter approach determines if a superset of the HUT is in the MFI. If a superset does exist, then the HUT must be frequent and the subtree rooted at the node corresponding to X can be pruned away.

We call this type of superset pruning *HUTMFI* (see Figure 3). Note that HUTMFI does not expand any children to check for successful superset pruning, unlike FHUT where the leftmost branch of the subtree is explored. Therefore, in general, HUTMFI is preferable to FHUT pruning.

3.3 Dynamic Reordering

The benefit of dynamically reordering the children of each node based on support instead of following the lexicographic order is significant. However, dynamic reordering requires counting the support of all the extensions of a node and thus, would not be a pure depth-first traversal of the space.

Note that most of the elements of a node's tail will *not* be frequent extensions, and these same infrequent items appear in many tails below. An algorithm that trims the tail to only frequent extensions at a higher level will save a lot of computation. For example in Figure 1, if the itemset {1,4} is counted and found to be infrequent, then item 4 can be trimmed from the tail of all nodes in that subtree, and the itemsets {1,2,4}, {1,3,4} need never be counted.

Of particular note, PEP can be applied much faster with dynamic reordering. Since PEP depends on the support of each child relative to the parent, we can move all elements for which PEP holds from the tail to the head at once, quickly reducing the size of the tail.

The order of the tail elements is also an important consideration. Ordering the tail elements (possible children) by increasing support will keep the search space as small as possible. This heuristic was first used by Bayardo [7].

All of the components of the algorithm fit together nicely and help trim the search space. However, certain components are more effective than others. The interaction of the different pruning mechanisms and the effect on performance is analyzed in Section 5.1.

4 Representation of the Database

4.1 Representation and Counting Support

We chose to use a vertical bitmap representation for the database. In a vertical bitmap, there is one bit for each transaction in the database. If item **i** appears in transaction **j**, then bit **j** of the bitmap for item **i** is set to one; otherwise, the bit is set to zero. This naturally extends to itemsets. Let **X** be the itemset corresponding to the head of a node. Let *onecount(**X**)* be the number of ones in the vertical bitmap for **X**. Note that the count of ones is exactly the support of itemset **X**. Let *bitmap(**X**)* correspond to the vertical bitmap that represents the transaction set for the itemset **X**. For each element **Y** in the node's tail, $t(X) \cap t(Y)$ is simply computed as the bitwise-AND of *bitmap(**X**)* and *bitmap(**Y**)*.

As the fundamental step in generating each new node in the lexicographic tree, support counting must be highly optimized. This necessity motivated the vertical bitmap representation. We adapted a two-phase byte counting method for generating and counting the extensions to an

itemset. Offline, we compute and store the number of 1's of a particular byte value for all 256 possible byte values, e.g. byte value "2" has 1 one, "3" has 2 ones, and "255" has 8 ones. Generation of new itemset bitmaps involves bitwise-ANDing bitmap(**X)** with a bitmap for 1-itemset **Y** and storing the result in bitmap (**X** ⋃ **Y**). Next, for each byte in bitmap (**X** ⋃ **Y**), we lookup the number of 1's in the byte using the pre-computed table. Summing these lookups gives the support of (**X** ⋃ **Y**).

4.2 Compression And Projected Bitmaps

The weakness of a vertical representation is the sparseness of the bitmaps, especially at the lower support levels. Since every transaction has a bit in vertical bitmaps, there are many zeros since both the absence and presence of the itemset in a transaction need to be represented. However, in cases where itemsets appear in a significant number of transactions, the vertical bitmap is the smallest representation of the information we need possible and all methods to make the representation smaller involve some form of lossless compression and CPU-computation.

An alternative representation would be a vertical tid-list for each item, where each item has the list of transactions it appears in associated with it. Even though only the presence of items is represented, it is guaranteed to be a more expensive representation in terms of space if the support of an item is greater than (1/32) or about 3%. In a vertical tid-list representation, we need an entire word (32 bits in most architectures) to represent the presence of an item versus the single bit of the vertical bitvector approach. Thus, if the item appears in more than (1/32) of the transactions, this is a less efficient representation.

Given the generating and counting method presented above, this run of zeros present a source of inefficiency since we are performing wasted operations over regions containing useless information (regions of only 0's in **X**'s bit vector).

Note that we only need information about transactions containing the itemset **X** to count the support of the subtree rooted at node **N**. If transaction **T** does not contain the itemset **X** (**X**'s bit vector has a 0 in bit **T**), then it will not provide useful information for counting supports of **N**'s children and can be ignored in all subsequent operations. So, conceptually we can remove the bit for transaction T from **X** and all items in **N**'s tail. This is a form of compression of the vertical bitmaps to speed up calculations. The pseudocode describing the process can be found in Figure 7.

In the projected bit vector for **X** all positions have value 1, since each position corresponds to a transaction needed for computation in the subtree rooted at node **N**. We are guaranteed to consider every element in **N**'s tail at least once when traversing the subtree rooted at **N**. The projection operation is performed when the support of itemset **X** drops below a certain ***rebuilding_threshold***, expressed as a percentage of bitmap **X**'s overall size (in bits).

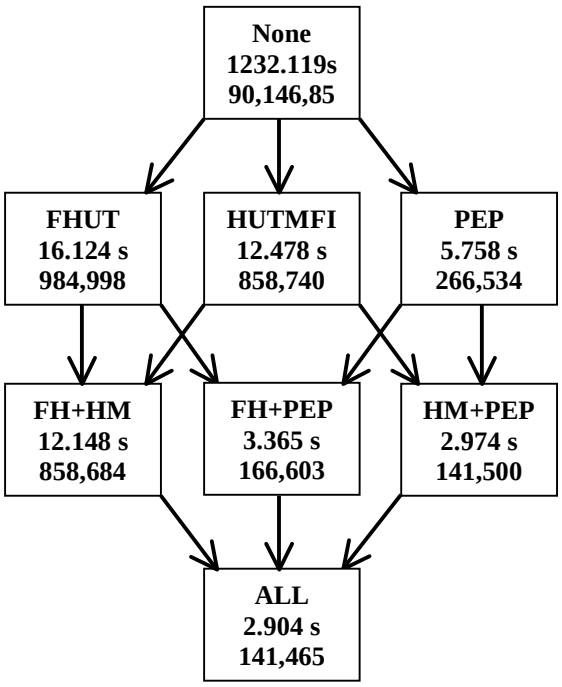
However, there is a tradeoff on the value selected for ***rebuilding_threshold***. As the ***rebuilding_threshold*** decreases, the cost of projection rises, since many more nodes at the lower levels of the tree will need to be projected. On the other hand, the size of the projected bitmaps is guaranteed to be smaller and thus the cost of subsequent generation and counting in that subtree is significantly reduced. In practice, we found that there is only a small difference in choosing a value for ***rebuilding_threshold*** between 0.25 and 1. Below 0.25, the cost of rebuilding exceeds the savings of using projected bitmaps, but between 0.25 and 1, the cost of projection is balanced by the smaller vertical bitmaps. The reason for this result is that the size of the projected bitmaps when built immediately (***rebuilding_threshold*** = 1) depends only on the support of the frequent 1-itemsets they are projected against. In the real datasets considered, many 1-itemsets have low supports, and thus, any ***rebuilding_threshold*** above the supports will yield the same projected bitmaps.

5 Experimental Results

The experiments were conducted on a 933 Mhz Pentium III with 512 MB of memory running Microsoft Windows 2000 Professional. All code was compiled using Microsoft Visual C++ 6.0. MAFIA was tested on three datasets that have been shown to generate long patterns [1,7]: connect-4, mushroom, and chess. In general, at the higher supports, the pattern length varies between 5-12 items, while at lower supports the patterns contain over 20 items. A detailed comparison of MAFIA on these datasets was conducted, since MAFIA yields the greatest gains when mining long patterns from a database.

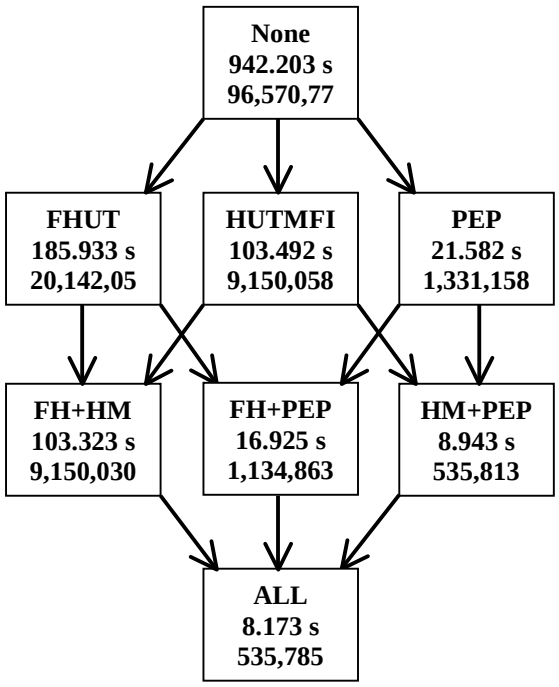
5.1 Algorithmic Component Analysis

First however, we present a full analysis of each component of the MAFIA algorithm (see Section 3 for algorithmic details). There are three types of pruning used to trim the tree: FHUT, HUTMFI, and PEP. FHUT and HUTMFI are both forms of superset pruning and thus will tend to "overlap" in their efficacy for reducing the search space. In addition, dynamic reordering will significantly reduce the size of the search space by removing infrequent items from each node's tail.



Mushroom at 1% support
Scaled 10x vertically
With Reordering

Figure 8



Mushroom at 1% support
Scaled 10x vertically
Without Reordering

Figure 9

Figures 8 and 9 show the effects of each component of the MAFIA algorithm on the mushroom dataset at 1% minimum support. The number of transactions was increased by repeating all transactions in the database by a certain scaling factor. We call this form of scaling *vertical scaling*. In this case, mushroom was scaled *ten* times vertically. Note that vertical scaling will not change the search space, and will only affect the time taken for counting the support of itemsets.

The components of the algorithm are represented in a *cube* format, where the running times and number of lattice nodes visited during the MAFIA search for all possible combinations are shown. The top of the cube shows the time for a simple traversal where the full search space is explored, while the bottom of the cube corresponds to all three pruning methods being used. Two separate cubes (with and without dynamic reordering) rather than one giant cube are presented for readability.

Note that all pruning components yield some savings in running time, but that certain components are more effective than others. In particular, HUTMFI and FHUT yield very similar results, since they use the same type of superset pruning, but with different methods of implementation. The efficient MFI lookups that HUTMFI uses to check for frequency explain why HUTMFI outperforms FHUT (see Section 3). It is also interesting to see that adding FHUT when HUTMFI is already performed yields very little savings, i.e. from HM to HM+FH or from HM+PEP to ALL, the running times do not significantly change. HUTMFI checks for the frequency of a node's HUT by looking for a frequent superset in the MFI, while FHUT will explore the leftmost branch of the subtree rooted at that node. Apparently, there are very few cases where a superset of a node's HUT is not in the MFI, but the HUT is frequent.

PEP has the biggest effect of the three pruning methods. All of the running time of the algorithm occurs at the lower levels of the tree where the border between frequent and infrequent itemsets exists, and since PEP is most likely to trim out large sections at the lower levels, this pruning yields the greatest results. Dynamically reordering the tail also has dramatic savings (cf. Figure 8 with Figure 9). It is interesting to note that without PEP, dynamic reordering runs nearly an order of magnitude faster than the static ordering, while with PEP, it is "only" 3-5 times faster. Since both PEP and reordering remove elements from a node's tail, it is not surprising that they overlap in their efficacy.

5.2 Comparison With DepthProject

We tested the algorithm on "real" datasets containing long patterns that have been used in earlier work [1,7].

These datasets are publicly available from the UCI Machine Learning Repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). At the lowest supports tested, the longest patterns in these databases have over 20 items, making any algorithm that examines all possible subsets of these patterns (or a significant portion thereof) infeasible. This makes the task of finding the patterns computationally intensive despite the small size of the databases. For some of the experiments the databases were scaled vertically by concatenating copies of the database together. This only affects the time counting takes since the bitmaps (compressed or not) are longer and the search space examined remains constant.

Figures 10 – 12 illustrate the results of comparing MAFIA to our implementation of the DepthProject method, the state-of-the-art method for finding maximal patterns [1]. The x-axis is the user-specified minimum support, while the y-axis uses a logarithmic scale to show the running time.

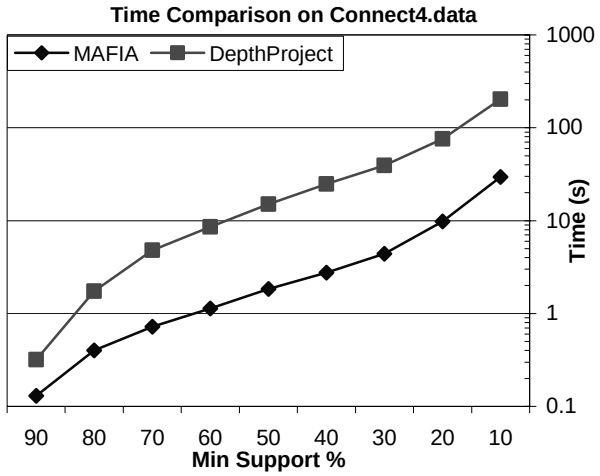


Figure 10

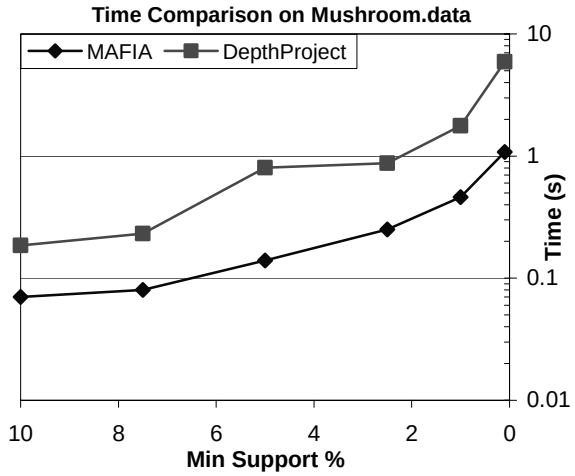


Figure 11

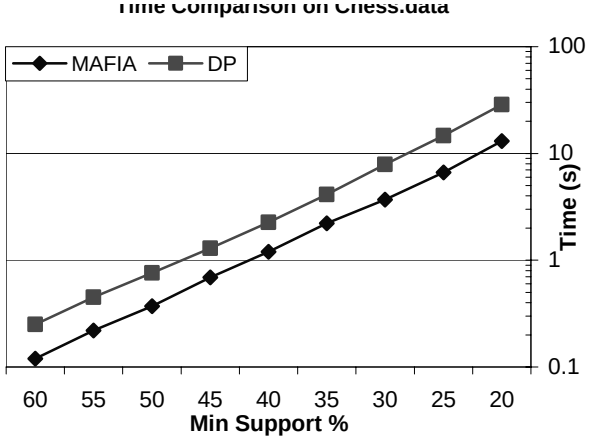


Figure 12

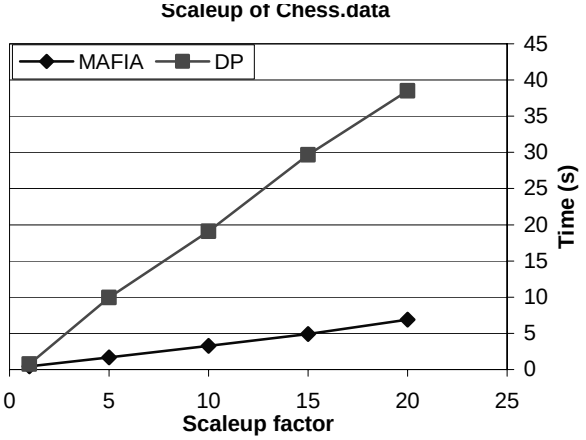


Figure 13

DEPTH	1	2	3	4	5	6	7	8
Connect-4	1.23	4.22	16.46	61.07	134.8	169.06	79.98	50.43
Mushroom	1	12.64	51.81	73.15	72.92	56.81	41.29	26.78
Chess	1.33	4.14	5.73	6.51	6.73	6.7	5.97	4.95

Table 1 – Reduction Factor of Nodes Considered Due to PEP Pruning

In Figures 10 and 11, we see MAFIA is approximately four to five times faster than DepthProject on both the Connect-4 and Mushroom datasets for all support levels tested (down to 10% support in Connect-4 and 0.1% in Mushroom). For Connect-4, the increased efficiency of itemset generation and support counting in MAFIA versus DepthProject explains the improvement. Connect-4 contains an order of magnitude more transactions than the other two datasets (67,557 transactions), amplifying the MAFIA advantage in generation and counting.

For Mushroom, the improvement is best explained by how often parent-equivalence pruning (PEP) holds, especially for the lowest supports tested. The dramatic effect PEP has on reducing the number of itemsets generated and counted is shown in Table 1. The entries in the table are the reduction factors due to PEP (in the presence of all other pruning methods) for the first eight levels of the tree. The reduction factor is defined as (# itemsets counted at depth k without PEP) / (# itemsets counted at depth k with PEP). In the first four levels, Mushroom has the greatest reduction in number of itemsets generated and counted. This leads to a much greater reduction in the overall search space than for the other datasets, since the reduction is so great at highest levels of the tree.

In Figure 12, we see that MAFIA is only a factor of two better than DepthProject on the dataset Chess. The extremely low number of transactions in Chess (3196 transactions) and the small number of frequent 1-items at low supports (only 54 at lowest tested support) muted the factors that MAFIA relies on to improve over DepthProject. Table 1 shows the reduction in itemsets using PEP for Chess was about an order of magnitude lower compared to the other two data sets for all depths.

To test the counting conjecture, we ran an experiment that vertically scaled the Chess dataset and fixed the support at 50%. This keeps the search space constant while varying only the generation and counting efficiency differences between MAFIA and DepthProject. The result is shown in Figure 13. We notice both algorithms scale linearly with the database size, but MAFIA is about five times faster than DepthProject. Similar results were found for the other datasets as well. Thus, we see MAFIA scales very well with the number of transactions.

5.3 Effects Of Compression

To isolate the effect of the compression schema on performance, experiments with varying *rebuilding-threshold* values we conducted. The most interesting result is on a scaled version of Connect-4, displayed in Figure 14. The Connect-4 dataset was scaled vertically three times, so the total number of transactions is approximately 200,000. Three different values for *rebuilding-threshold* were used: 0 (corresponding to no compression), 1 (compression immediately, and all subsequent operations performed on compressed bitmaps), and an optimized value determined empirically. We see for higher supports above 40% compression has a negligible effect, but at the lowest supports compression can be quite beneficial, e.g. at 10% support compression yields an improvement factor of 3.6.

However, the small difference between compressing immediately and finding an optimal compression point is not so easily explained. The greatest savings here is only 11% at the lowest support of Conenct-4 tested. We performed another experiment where the support was fixed and the Connect-4 dataset was scaled vertically. The results appear in Figure 15. The x-axis shows the scale up factor while the y-axis displays the running times. We can see that the optimal compression scales the best. For many transactions (over 10^6), the optimal *rel-threshold* outperforms compressing everywhere by approximately 35%. In any case, both forms of compression scale much better than no compression.

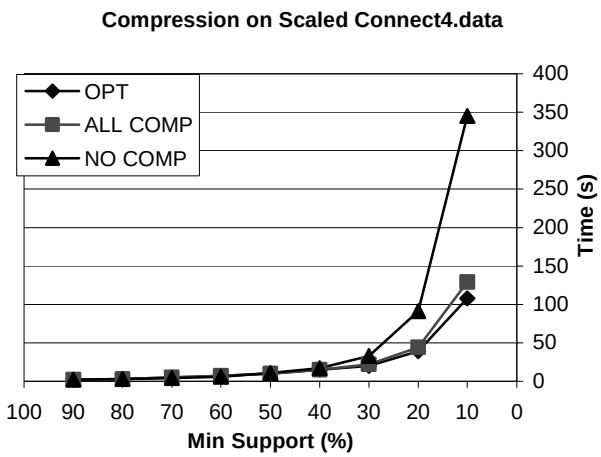


Figure 14

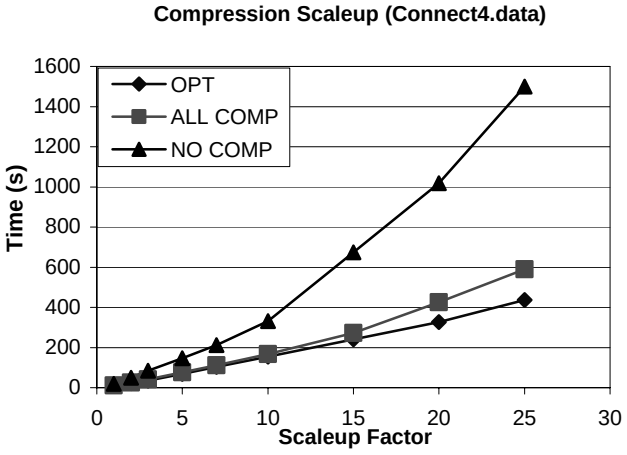


Figure 15

6 Conclusions

We presented MAFIA, an algorithm for finding maximal frequent itemsets. Our experimental results demonstrate that MAFIA consistently outperforms DepthProject by a factor of three to five on average. The breakdown of the algorithmic components showed parent-equivalence pruning and dynamic reordering were quite beneficial in reducing the search space while relative compression/projection of the vertical bitmaps dramatically cuts the cost of counting supports of itemsets and increases the vertical scalability of MAFIA.

Acknowledgements. We thank Ramesh Agarwal and Charu Aggarwal for discussing DepthProject and giving us advise on its implementation. We also thank Jayant R. Haritsa for his insightful comments on the MAFIA algorithm and Jiawei Han for providing us the executable of the FP-Tree algorithm. This research was partly supported by an IBM Faculty Development Award and by a grant from Microsoft Research.

References:

[1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad: A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing* (special issue on high performance data mining), (to appear), 2000.

[2] R. Agrawal, T. Imielinski, and R. Srikant: Mining association rules between sets of items in large databases. SIGMOD, May 1993.

[3] R. Agrawal, R. Srikant: Fast Algorithms for Mining Association Rules. *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, Sept. 1994.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo: Fast Discovery of Association Rules. *Advances in Knowledge Discovery and Data Mining*, Chapter 12, AAAI/MIT Press, 1995.

[5] C. C. Aggarwal, P. S. Yu: Mining Large Itemsets for Association Rules. Data Engineering Bulletin 21(1): 23-31 (1998)

[6] C. C. Aggarwal, P. S. Yu: Online Generation of Association Rules. ICDE 1998: 402-411

[7] R. J. Bayardo: Efficiently mining long patterns from databases. SIGMOD 1998: 85-93.

[8] R. J. Bayardo and R. Agrawal. Mining the Most Interesting Rules. *SIGKDD 1999:* 145-154.

[9] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):255, 1997.

[10] B. Dunkel and N. Soparkar: Data Organization and access for efficient data mining. ICDE 1999.

[11] V. Ganti, J. E. Gehrke, and R. Ramakrishnan: DEMON: Mining and Monitoring Evolving Data. ICDE 2000: 439-448

[12] D. Gunopulos, H. Mannila, and S. Saluja: Discovering All Most Specific Sentences by Randomized Algorithms. ICDT 1997: 215-229

[13] J. Han, J. Pei, and Y. Yin: Mining Frequent Patterns without Candidate Generation. SIGMOD Conference 2000: 1-12.

[14] M. Holsheimer, M. L. Kersten, H. Mannila, and H.Toivonen: A Perspective on Databases and Data Mining. KDD 1995: 150-155.

[15] W. Lee and S. J. Stolfo: Data mining approaches for intrusion detection. *Proceedings of the 7th USENIX Security Symposium*, 1998.

[16] D. I. Lin and Z. M. Kedem: Pincer search: A new algorithm for discovering the maximum frequent sets. *Proc. of the 6th Int'l Conference on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.

[17] J.-L. Lin, M.H. Dunham: Mining Association Rules: Anti-Skew Algorithms. ICDE 1998: 486-493

[18] B. Mobasher, N. Jain, E. H. Han, and J. Srivastava: Web mining: Pattern discovery from world wide web transactions. Technical Report TR-96050, Department of Computer Science, University of Minnesota, Minneapolis, 1996

[19] J. S. Park, M.-S. Chen, P. S. Yu: An Effective Hash Based Algorithm for Mining Association Rules. SIGMOD Conference 1995: 175-186

[20] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal: Discovering frequent closed itemsets for association rules. ICDT '99: 398-416, Jerusalem, Israel, January 1999.

[21] J. Pei, J. Han, and R. Mao: CLOSET: An efficient algorithm for mining frequent closed itemsets. Proc. of ACM SIGMOD DMKD Workshop, Dallas, TX, May, 2000.

[22] R. Rastogi and K. Shim: Mining Optimized Association Rules with Categorical and Numeric Attributes. ICDE 1998, Orlando, Florida, February 1998.

[23] L. Rigoutsos and A. Floratos: Combinatorial pattern discovery in biological sequences: The Teiresias algorithm. *Bioinformatics* 14, 1 (1998), 55-67.

[24] R. Rymon: Search through Systematic Set Enumeration. Proc. Of Third Int'l Conf. On Principles of Knowledge Representation and Reasoning, 539 –550. 1992

[25] A. Savasere, E. Omiecinski, and S. Navathe: An efficient algorithm for mining association rules in large databases. 21st VLDB Conference, 1995.

[26] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah: Turbo-charging Vertical Mining of Large Databases. SIGMOD Conference 2000: 22-33

[27] R. Srikant, R. Agrawal: Mining Generalized Association Rules. VLDB 1995: 407-419.

[28] H. Toivonen: Sampling Large Databases for Association Rules. VLDB 1996: 134-145.

[29] K. Wang, Y. He, J. Han: Mining Frequent Itemsets Using Support Constraints. VLDB 2000: 43-52

[30] G. I. Webb: OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:45-83, 1996.

[31] L. Yip, K. K. Loo, B. Kao, D. Cheung, and C.K. Cheng: Lgen. A Lattice-Based Candidate Set Generation Algorithm for I/O Efficient Association Rule Mining. PAKDD '99, Beijing, 1999.

[32] M. J. Zaki: Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, pp 372-390, May/June 2000.

[33] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. RPI Technical Report 99-10, 1999.