Data Mining and Knowledge Discovery, 11, 1–20, 2005

GenMax: An Efficient Algorithm for Mining Maximal Frequent Itemsets

KARAM GOUDA karam_g@hotmail.com
Department of Mathematics, Faculty of Science, Benha, Egypt

MOHAMMED J. ZAKI zaki@cs.rpi.edu
Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180, USA

Abstract. We present GenMax, a backtrack search based algorithm for mining maximal frequent itemsets. GenMax uses a number of optimizations to prune the search space. It uses a novel technique *called progressive focusing* to perform maximality checking, and *diffset propagation* to perform fast frequency computation. Systematic experimental comparison with previous work indicates that different methods have varying strengths and weaknesses based on dataset characteristics. We found GenMax to be a highly efficient method to mine the exact set of maximal patterns.

Keywords: maximal itemsets, frequent itemsets, association rules, data mining, backtracking search

1. Introduction

Mining frequent itemsets is a fundamental and essential problem in many data mining applications such as the discovery of association rules, strong rules, correlations, multi-dimensional patterns, and many other important discovery tasks. The problem is formulated as follows: Given a large data base of set of items transactions, find all frequent itemsets, where a frequent itemset is one that occurs in at least a user-specified percentage of the data base.

Many of the proposed itemset mining algorithms are a variant of Apriori (Agrawal et al., 1996), which employs a bottom-up, breadth-first search that enumerates every single frequent itemset. In many applications (especially in dense data) with long frequent patterns enumerating all possible 2^m-2 subsets of a m length pattern *(m* can easily be 30 or 40 or longer) is computationally unfeasible. Thus, there has been recent interest in mining *maximal* frequent patterns in these "hard" dense databases. Another recent promising direction is to mine only closed sets (Zaki, 2000; Zaki and Hsiao, 2002); a set is closed if it has no superset with the same frequency. Nevertheless, for some of the dense datasets we consider in this paper, even the set of all closed patterns would grow to be too large. The only recourse is to mine the maximal patterns in such domains.

In this paper we introduce *GenMax*, a new algorithm that utilizes a backtracking search for efficiently enumerating all maximal patterns. GenMax uses a number of optimizations to quickly prune away a large portion of the subset search space. It uses a novel *progressive focusing* technique to eliminate non-maximal itemsets, and uses *diffset propagation* for fast frequency checking.

We conduct an extensive experimental characterization of GenMax against state-of-the-art maximal pattern mining methods like MaxMiner (Bayardo, 1998) and Mafia (Burdick et al., 2001). We found that the three methods have varying performance depending on the database characteristics (mainly the distribution of the maximal frequent patterns by length). We present a systematic and realistic set of experiments showing under which conditions a method is likely to perform well and under what conditions it does not perform well. We conclude that while Mafia is good for mining a *superset* of all maximal patterns, GenMax is the method of choice for enumerating the *exact* set of maximal patterns. We further observe that there is a type of data, where MaxMiner delivers the best performance.

2. Preliminaries and related work

The problem of mining maximal frequent patterns can be formally stated as follows: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct items. Let $\mathcal{D}$ denote a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items. The set of all tids is denoted $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$. A set $X \subseteq \mathcal{I}$ is also called an *itemset*. An itemset with k items is called a k-itemset. The set $t(X) \subseteq \mathcal{T}$, consisting of all the transaction tids which contain X as a subset, is called the *tidset* of X. For convenience we write an itemset $\{A, C, W\}$ as ACW, and its tidset $\{1, 3, 4, 5\}$as $t(X) = 1345$.

The *support* of an itemset X, denoted $\sigma(X)$, is the number of transactions in which that itemset occurs as a subset. Thus $\sigma(X) = |t(X)|$. An itemset *is frequent* if its support is more than or equal to some threshold *minimum support* (*min_sup*) value, i.e., if $\sigma(X) >$ *min_sup*. We denote by F_k the set of frequent k-itemsets, and by **FI** the set of all frequent itemsets. A frequent itemset is called *maximal* if it is not a subset of any other frequent itemset. The set of all maximal frequent itemsets is denoted as **MFI**. Given a user specified *miti-sup* value our goal is to efficiently enumerate all patterns in **MFI**.

Example 1. Consider our example database in figure 1. There are five different items, $\mathcal{I} = \{A, C, D, T, W\}$ and six transactions $\mathcal{T} = \{1, 2, 3, 4, 5, 6\}$. The figure also shows the frequent and maximal k-itemsets at two different min_sup values – 3(50%) and 2(30%) respectively.

2.1. Backtracking search

GenMax uses backtracking search to enumerate the **MFI**. We first describe the backtracking paradigm in the context of enumerating all frequent patterns. We will subsequently modify this procedure to enumerate the **MFI**.

Backtracking algorithms are useful for many combinatorial problems where the solution can be represented as a set $\mathcal{I} = \{i_0, i_1, \ldots\}$, where each i_j is chosen from a finite *possible set*, P_j. Initially I is empty; it is extended one item at a time, as the search space is traversed. The length of I is the same as the depth of the corresponding node in the search tree. Given a partial solution of length l, $I_l = \{i_0, i_1, \ldots, i_{l-1}\}$, the possible values for the next item i_t comes from a subset $C_l \subseteq P_l$ called the *combine set*. If $y \in P_l - C_l$, then nodes in the subtree with root node $I_l = \{i_0, i_1, \ldots, i_{l-1}, y\}$ will not be considered by the backtracking

TID	Items	Frequent itemsets Min_Sup = 3 trans	Frequent itemsets Min_Sup = 2 trans	Itemset Size	Maximal itemsets Min_Sup=3 trans	Maximal itemsets Min_Sup = 2 trans
1	ACTW	A, C, D, T, W	A, C, D, T, W	1		
2	CDW	AC, AT, AW, CD, CT, CW, DW, TW	AC, AD, AT, AW, CD, CT, CW, DT, DW, TW	2		
3	ACTW					
4	ACDW	ACT, ACW, ATW,CTW, CDW,	ACD, ACT, ACW, ADW, ATW, CDT, CDW, CTW	3	CDW	CDT
5	ACDTW					
6	CDT					
		ACTW	ACDW, ACTW	4	ACTW	ACDW, ACTW

Figure 1. Mining frequent itemsets.

algorithm. Since such subtrees have been pruned away from the original search space, the determination of C_l is also called *pruning*.

Consider the backtracking algorithm for mining all frequent patterns, shown in figure 2. The main loop tries extending I_l with every item x in the current combine set C_l. The first step is to compute I_{l+1}, which is simply I_l extended with x. The second step is to extract the new possible set of extensions, P_{l+1}, which consists only of items y in C_l that follow x. The third step is to create a new combine set for the next pass, consisting of valid extensions. An extension is valid if the resulting itemset is frequent. The combine set, C_{l+1}, thus consists of those items in the possible set that produce a frequent itemset when used to extend I_{l+1} Any item not in the combine set refers to a pruned subtree. The final step is to recursively call the backtrack routine for each extension. As presented, the backtrack method performs a depth-first traversal of the search space.

Example 2. Consider the full subset search space shown in figure 3. The backtrack search space can be considerably smaller than the full space. For example, we start with $I_0 = \emptyset$ and $C_0 = F_1 = \{A, C, D, T, W\}$. At level 1, each item in C_0 is added to I_0 in turn. For example, A is added to obtain $I_1 = \{A\}$. The possible set for A, $P_1 = \{C, D, T, W\}$ consists of all items that follow A in C_0 However, from figure 1, we find that only AC, AT, and AW are frequent (at min_sup = 3), giving $C_1 = \{C, T, W\}$. Thus the subtree corresponding to the node AD has been pruned.

```
// Invoke as FI-backtrack(∅, F_1, 0)
FI-backtrack(I_l, C_l, l)
1.  for each x ∈ C_l
2.      I_{l+1} = I_l ∪ {x} //also add I_{l+1} to FI
3.      P_{l+1} = {y : y ∈ C_l and y > x}
4.      C_{l+1} = FI-combine (I_{l+1}, P_{l+1})
5.      FI-backtrack(I_{l+1}, C_{l+1}, l + 1)

// Can I_{l+1} combine with other items in C_l?
FI-combine(I_{l+1}, P_{l+1})
1.  C = ∅
2.  for each y ∈ P_{l+1}
3.      if I_{l+1} ∪ {y} is frequent
4.          C = C ∪ {y} //reorder C if required
5.  return C
```

Figure 2. Backtrack algorithm for mining **FI**.

2.2. Related work

Methods for finding the maximal elements include All-MFS (Gunopulos et al., 2003), which works by iteratively attempting to extend a working pattern until failure. A randomized version of the algorithm that uses vertical bit-vectors was studied, but it does not guarantee every maximal pattern will be returned. The Pincer-Search (Lin and Kedem, 1998) algorithm uses horizontal data format. It not only constructs the candidates in a bottom-up manner like Apriori, but also starts a top-down search at the same time, maintaining a candidate set of maximal patterns. This can help in reducing the number of database scans, by eliminating non-maximal sets early. The maximal candidate set is a superset of the maximal patterns, and in general, the overhead of maintaining it can be very high. In contrast GenMax maintains only the current known maximal patterns for pruning.

MaxMiner (Bayardo, 1998) is another algorithm for finding the maximal elements. It uses efficient pruning techniques to quickly narrow the search. MaxMiner employs a breadth-first traversal of the search space; it reduces database scanning by employing a lookahead pruning strategy, i.e., if a node with all its extensions can determined to be frequent, there is no need to further process that node. It also employs item (re)ordering heuristic to increase the effectiveness of superset-frequency pruning. Since MaxMiner uses the original horizontal database format, it can perform the same number of passes over a database as Apriori does.

DepthProject (Agrawal et al., 2000) finds long itemsets using a depth first search of a lexicographic tree of itemsets, and uses a counting method based on transaction projections

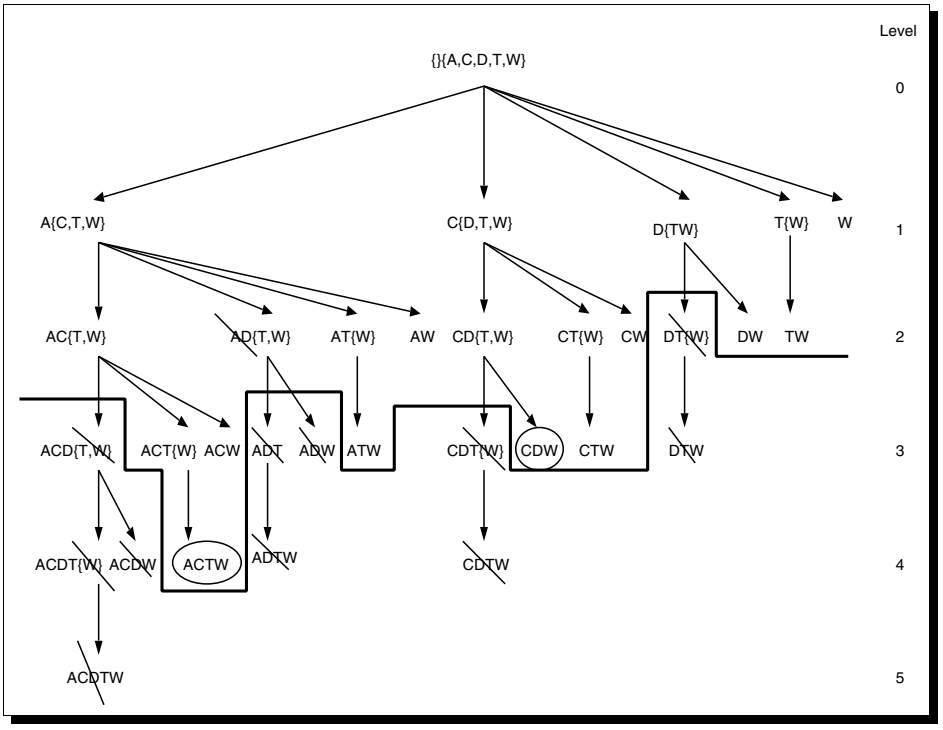

Figure 3. Subset/Backtrack Search Tree *(min_sup* $= 3$): Circles indicate maximal sets and the infrequent sets have been crossed out. Due to the downward closure property of support (i.e., all subsets of a frequent itemset must be frequent) the frequent itemsets form a *border* (shown with the bold line), such that all frequent itemsets lie above the border, while all infrequent itemsets lie below it. Since **MFI** determine the border, it is straightforward to obtain **FI** in a single database scan of **MFI** is known.

along its branches. This projection is equivalent to a horizontal version of the tidsets at a given node in the search tree. DepthProject also uses the look-ahead pruning method with item reordering. It returns a superset of the **MFI** and would require post-pruning to eliminate non-maximal patterns. FPgrowth (Han et al., 2000) uses the novel frequent pattern tree (**FP**-tree) structure, which is a compressed representation of all the transactions in the database. It uses a recursive divide-and-conquer and database projection approach to mine long patterns. Nevertheless, since it enumerates all frequent patterns it is impractical when pattern length is long.

Mafia (Burdick et al., 2001) is the most recent method for mining the **MFI**. Mafia uses three pruning strategies to remove non-maximal sets. The first is the look-ahead pruning first used in MaxMiner. The second is to check if a new set is subsumed by an existing maximal set. The last technique checks if $t(X) \subseteq t(Y)$. If so X is considered together with Y for extension. Mafia uses vertical bit-vector data format, and compression and projection of bitmaps to improve performance. Mafia mines a superset of the **MFI**, and requires a post-pruning step to eliminate non-maximal patterns. In contrast GenMax integrates pruning with mining and returns the exact **MFI**.

Recently there has been a surge of interest in experimentally comparing algorithms for frequent itemset mining, including the problem of computing the **MFI**. We refer the reader to Goethals and Zaki (2003) for more details on the Frequent Itemset Mining Implementations (FIMI) workshop. The FIMI web-page (http://fimi.cs.helsinki.fi/) also contains the open source implementations of many new **MFI** algorithms, as well as new datasets that can be used for testing.

3. GenMax for efficient MFI mining

There are two main ingredients to develop an efficient **MFI** algorithm. The first is the set of techniques used to remove entire branches of the search space, and the second is the representation used to perform fast frequency computations. We will describe below how GenMax extends the basic backtracking routine for **FI**, and then the progressive focusing and diffset propagation techniques it uses for fast maximality and frequency checking.

The basic **MFI** enumeration code used in GenMax is a straightforward extension of **Fl-backtrack**. The main addition is the superset checking to eliminate non-maximal itemsets, as shown in figure 4. In addition to the main steps in **FI** enumeration, the new code adds a step (line 4) after the construction of the possible set to check if $I_{l+1} \cup P_{l+1}$ is subsumed by an existing maximal set. If so, the current and all subsequent items in C_l can be pruned away. After creating the new combine set, if it is empty and I_{l+1} is not a subset of any maximal pattern, it is added to the **MFI**. If the combine set is non-empty a recursive call is made to check further extensions.

```
// Invocation: MFI-backtrack(∅, F_1, 0)
MFI-backtrack(I_l, C_l, l)
1.    for each x ∈ C_l
2.        I_{l+1} = I_l ∪ {x}
3.        P_{l+1} = {y : y ∈ C_l and y > x}
4.*       if I_{l+1} ∪ P_{l+1} has a superset in MFI
5.*           return //all subsequent branches pruned!
6.        C_{l+1} = Fl-combine (I_{l+1}, P_{l+1})
7.*       if C_{l+1} is empty
8.*           if I_{l+1} has no superset in MFI
9.*               MFI= MFI ∪ I_{l+1}
10.   else MFI-backtrack(I_{l+1}, C_{l+1}, l + 1)
```

Figure 4. Backtrack algorithm for mining **MFI**(*indicates a new line not in Fl-backtrack).

3.1. Superset checking techniques

Checking to see if the given itemset I_{l+1} combined with the possible set P_{l+1} is subsumed by another maximal set was also proposed in Mafia (Burdick et al., 2001) under the name HUTMFI. Further pruning is possible if one can determine based just on support of the combine sets if $I_{l+1} \cup P_{l+1}$ will be guaranteed to be frequent. In this case also one can avoid processing any more branches. This method was first introduced in MaxMiner (Bayardo, 1998), and was also used in Mafia under the name FHUT.

3.2. Reordering the combine set

Two general principles for efficient searching using backtracking are that: (1) It is more efficient to make the next choice of a subtree (branch) to explore to be the one whose combine set has the fewest items. This usually results in good performance, since it minimizes the number of frequency computations in Fl-combine. (2) If we are able to remove a node as early as possible from the backtracking search tree we effectively prune many branches from consideration.

Reordering the elements in the current combine set to achieve the two goals is a very effective means of cutting down the search space. The first heuristic is to reorder the combine set in increasing order of support. This is likely to produce small combine sets in the next level, since the items with lower frequency are less likely to produce frequent itemsets at the next level. This heuristic was first used in MaxMiner, and has been used in other methods since then (Agrawal et al., 2000; Burdick et al., 2001; Zaki and Hsiao, 2002). At each level of backtracking search, GenMax reorders the combine set in increasing order of support (this is indicated in figures 2, 7, and 8).

In addition to sorting the initial combine set at level 0 in increasing order of support, GenMax uses another reordering heuristic based on a simple lemma.

Lemma 1. *Let* $IF(x) = \{y : y \in F_1, xy \text{ is not frequent}\}$, *denote the set of infrequent 2-itemsets that contain an item* $x \in F_1$, *and let* $M(x)$ *be the longest maximal pattern containing* x. *Then* $|M(x)| \leq |F_1| - |IF(x)|$.

Proof: Assume there exists a maximal pattern containing x with $|M(x)| > |F_1| - |FI(x)|$. This implies $|M(x)| + |IF(x)| > |F_1|$. But this is a contradiction since $IF(x)$ and $M(x)$ are disjoint, $M(x) \subseteq F_1$ and $IF(x) \subseteq F_1$. Their combined size thus cannot exceed $|F_1|$.

Assuming F_2 has been computed, reordering C_0 in decreasing order of $IF(x)$ (with $x \in C_0$) ensures that the smallest combine sets will be processed at the initial levels of the tree, which result in smaller backtracking search trees. GenMax thus initially sorts the items in decreasing order of $IF(x)$ and in increasing order of support. Then at each subsequent level, GenMax keeps the combine set in increasing order of support.

Example 3. For our database in figure 1 with min_sup = 2, $IF(x)$ is the same of all items $x \in F_1$, and the sorted order (on support) is A, D, T, W, C. Figure 5 shows the backtracking

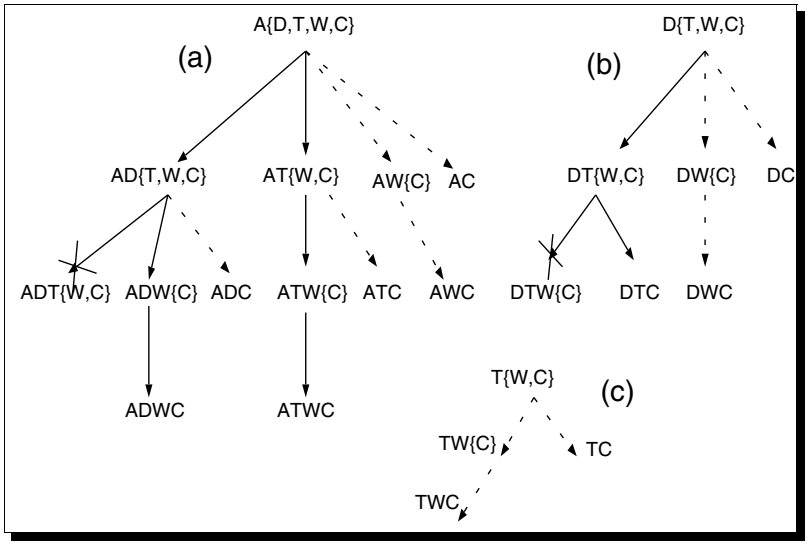

Figure 5. Backtracking trees of Example 2.

search trees for maximal itemsets containing prefix items A and D. Under the search tree for A, figure 5(a), we try to extend the partial solution AD by adding to it item T from its combine set. We try another item W after itemset ADT turns out to be infrequent, and so on. Since GenMax uses itemsets which are found earlier in the search to prune the combine sets of later branches, after finding the maximal set $ADWC$, GenMax skips ADC. After finding $ATWC$ all the remaining nodes with prefix A are pruned, and so on. The pruned branches are shown with dashed arrows, indicating that a large part of the search tree is pruned away.

Theorem 1 (Correctness). *MFI-backtrack returns all and only the maximal frequent itemsets in the given database.*

Proof: The **Fl-backtrack** procedure considers every possible frequent extension $I_l \cup \{x\}$ of the current itemset I_l. It thus generates all frequent itemsets. The **MFI-backtrack** procedure removes additional branches only if the current itemset along with its entire possible set ($I_{l+1} \cup P_{l+1}$) is subsumed by some maximal frequent itemset, since in this case no extension of I_{l+1} can possibly be maximal. Finally, a set is added to the maximal collection only if it is not a subset of any other maximal frequent itemset. The proof of correctness of GenMax thus follows from the correctness of the method, Fl-backtrack, which enumerates all frequent itemsets, combined with the fact that only non-maximal branches are eliminated from search.

3.3. Optimizing GenMax

3.3.1. Superset checking optimization. The main efficiency of GenMax stems from the fact that it eliminates branches that are subsumed by an already mined maximal pattern.

Were it not for this pruning, GenMax would essentially default to a depth-first exploration of the search tree. Before creating the combine set for the next pass, in line 4 in figure 4, GenMax check if $I_{l+1} \cup P_{l+1}$ is contained within a previously found maximal set. If yes, then the entire subtree rooted at I_{l+1} and including the elements of the possible set are pruned. If no, then a new extension is required. Another superset check is required at line 8, when I_{l+1} has no frequent extension, i.e., when the combine set C_{l+1} is empty. Even though I_{l+1} is a leaf node with no extensions it may be subsumed by some maximal set, and this case is not caught by the check in line 4 above.

The major challenge in the design of GenMax is how to perform this subset checking in the current set of maximal patterns in an efficient manner. If we were to naively implement and perform this search two times on an ever expanding set of maximal patterns **MFI**, and during each recursive call of backtracking, we would be spending a prohibitive amount of time just performing subset checks. Each search would take $O(|\mathbf{MFI}|)$ time in the worst case, where **MFI** is the current, growing set of maximal patterns. Note that some of the best algorithms for dynamic subset testing run in amortized time $O\left(\sqrt{s}\log s\right)$ per operation in a sequence of s operations (Yellin, 1994) (for us $s = O(\mathbf{MFI})$). In dense domain we have thousands to millions of maximal frequent itemsets, and the number of subset checking operations performed would be at least that much. Can we do better?

The answer is, yes! Firstly, we observe that the two subset checks (one on line 4 and the other on line 8) can be easily reduced to only one check. Since $I_{l+1} \cup P_{l+1}$ is a superset of I_{l+1}, in our implementation we do superset check only for $I_{l+1} \cup P_{l+1}$ While testing this set, we store the maximum position, say p, at which an item in $I_{l+1} \cup P_{l+1}$ is not found in a maximal set $M \in \mathbf{MFI}$. In other words, all items before p are subsumed by some maximal set. For the superset test for I_{l+1} we check if $|I_{l+1}| < p$. If yes, I_{l+1} is non-maximal. If no, we add it to **MFI**.

The second observation is that performing superset checking during each recursive call can be redundant. For example, suppose that the cardinality of the possible set P_{l+1} is m. Then potentially, MFI-backtrack makes m redundant subset checks, if the current **MFI** has not changed during these m consecutive calls. To avoid such redundancy, a simple check_status flag is used. If the flag is false, no superset check is performed. Before each recursive call the flag is false; it becomes true whenever C_{l+1} is empty, which indicates that we have reached a leaf, and have to backtrack.

The $O\left(\sqrt{s}\log s\right)$ time bounds reported in Yellin (1994) for dynamic subset testing do not assume anything about the sequence of operations performed. In contrast, we have full knowledge of how GenMax generates maximal sets; we use this observation to substantially speed up the subset checking process. The main idea is to progressively narrow down the maximal itemsets of interest as recursive calls are made. In other words, we construct for each invocation of **MFI**-backtrack a list of *local maximal frequent itemsets, LMFI*$_l$. This list contains the maximal sets that can potentially be supersets of candidates that are to be generated from the itemset I_l The only such maximal sets are those that contain all items in I_l. This way, instead of checking if $I_{l+1} \cup P_{l+1}$ is contained in the full current **MFI**, we check only in $LMFI_l$ – the local set of relevant maximal itemsets. This technique, that we *call progressive focusing,* is extremely powerful in narrowing the search to only the most relevant maximal itemsets, making superset checking practical on dense datasets.

```
// Invocation: LMFI-backtrack(∅, F_1, ∅, 0)
// LMFI_l is an output parameter
LMFI-backtrack(I_l, C_l, LMFI_l, l)
1.   for each x ∈ C_l
2.       I_{l+1} = I_l ∪ {x}
3.       P_{l+1} = {y : y ∈ C_l and y > x}
4.       if I_{l+1} ∪ P_{l+1} has a superset in LMFI_l
5.           return //subsequent branches pruned!
6. *     LMFI_{l+1} = ∅
7.       C_{l+1} = FI-combine (I_{l+1}, P_{l+1})
8.       if C_{l+1} is empty
9.           if I_{l+1} has no superset in LMFI_l
10.              LMFI_l = LMFI_l ∪ I_{l+1}
11.*     else LMFI_{l+1} = {M ∈ LMFI_l : x ∈ M}
12.          LMFI-backtrack(I_{l+1}, C_{l+1}, LMFI_{l+1}, l + 1)
13.*     LMFI_l = LMFI_l ∪ LMFI_{l+1}
```

Figure 6. Mining **MFI** with progressive focusing (*indicates a new line not in MFI-backtrack).

Figure 6 shows the pseudo-code for GenMax that incorporates this optimization (the code for the first two optimizations is not shown to avoid clutter). Before each invocation of LMFI-backtrack a new $LMFI_{l+1}$ is created, consisting of those maximal sets in the current $LMFI_l$ that contain the item x (see line 10). Any new maximal itemsets from a recursive call are incorporated in the current $LMFI_l$ at line 12. Note that the correctness of the algorithm is not affected by this optimization, since progressive focusing only removes those itemsets that cannot subsume the itemsets in line 4 and 9. Thus Theorem 1 still holds.

3.3.2. Frequency testing optimization. So far GenMax, as described, is independent of the data format used. The techniques can be integrated into any of the existing methods for mining maximal patterns. We now present some data format specific optimizations for fast frequency computations.

GenMax uses a vertical database format, where we have available for each item its tidset, the set of all transaction tids where it occurs. The vertical representation has the following major advantages over the horizontal layout: Firstly, computing the support of itemsets is simpler and faster with the vertical layout since it involves only the intersections of tidsets (or compressed bit-vectors if the vertical format is stored as bitmaps (Burdick et al., 2001)). Secondly, with the vertical layout, there is an automatic "reduction" of the database before each scan in that only those itemsets that are relevant to the following scan of the mining process are accessed from disk. Thirdly, the vertical format is more versatile in supporting various search strategies, including breadth-first, depth-first or some other hybrid search.

```
// Can I_{l+1} combine with other items in C_l?
FI-tidset-combine(I_{l+1}, P_{l+1})
1.   C = ∅
2.   for each y ∈ P_{l+1}
3.*      y' = y
4.*      t(y') = t(I_{l+1}) ∩ t(y)
5.*      if |t(y')| ≥ min_sup
6.           C = C ∪ {y'} //add y' in increasing order of support
7.   return C
```

Figure 7. Fl-combine using tidset intersections (*indicates a new line not in Fl-combine).

Let's consider how the Fl-combine (see figure 2) routine works, where the frequency of an extension is tested. Each item x in C_l actually represents the itemset $I_l \cup \{x\}$ and stores the associated tidset for the itemset $I_l \cup \{x\}$. For the initial invocation, since I_l is empty, the tidset for each item x in C_l is identical to the tidset, $t(x)$, of item x. Before line 3 is called in FI-combine, we intersect the tidset of the element I_{l+1} (i.e., $t(I_l \cup \{x\})$) with the tidset of element y (i.e., $t(I_l \cup \{y\})$). If the cardinality of the resulting intersection is above minimum support, the extension with y is frequent, and y' the new intersection result, is added to the combine set C for the next level. C is kept in increasing order of support of its elements. Figure 7 shows the pseudo-code for **Fl**-tidset-combine using this tidset intersection based support counting.

In Charm (Zaki and Hsiao, 2002) we first introduced two new properties of itemset-tidset pairs which can be used to further increase the performance. Consider the items x and y in C_l. If during intersection in line 4 in figure 7, we discover that $t(x)$ – or equivalently $t(I_{l+1})$—is a subset of or equal to $t(y)$, then we do not add y' to the combine set, since in this case, x always occurs along with y. Instead of adding y' to the combine set, we add it to I_{l+1}. This optimization was also used in Mafia (Burdick et al., 2001) under the name PEP.

3.3.3. Diffsets propagation. Despite the many advantages of the vertical format, when the tidset cardinality gets very large (e.g., for very frequent items) the intersection time starts to become inordinately large. Furthermore, the size of intermediate tidsets generated for frequent patterns can also become very large to fit into main memory. GenMax uses a new format called diffsets (Zaki and Gouda, 2003) for fast frequency testing.

The main idea of diffsets is to avoid storing the entire tidset of each element in the combine set. Instead we keep track of only the differences between the tidset of itemset I_l and the tidset of an element x in the combine set (which actually denotes $I_l \cup \{x\}$). These differences in tids are stored in what we call the *diffset,* which is a difference of two tidsets at the root level or a difference of two diffsets at later levels. Furthermore, these differences are propagated all the way from a node to its children starting from the root. In an extensive study (Zaki and Gouda, 2003), we showed that diffsets are very short compared to their

```
// Can I_{l+1} combine with other items in C_l?
FI-diffset-combine(I_{l+1}, P_{l+1})
1.  C = ∅
2.  for each y ∈ P_{l+1}
3.    y' = y
4.    if level == 0 then d(y') = t(I_{l+1}) − t(y)
5.    else d(y') = d(y) − d(I_{l+1})
6.      if σ(y') ≥ min_sup
7.          C = C ∪ {y'} //add y' in increasing order of support
8.  return C
```

Figure 8. Fl-combine: diffset propagation.

tidsets counterparts, and are highly effective in improving the running time of vertical methods.

We describe next how they are used in GenMax, with the help of an example. At level 0, we have available the tidsets for each item in F_1. When we invoke Fl-combine at this level, we compute the diffset of y', denoted as $d(y')$ instead of computing the tidset of y as shown in line 4 in figure 7. That is $d\left(y'\right) = t\left(x\right) - t\left(y\right)$. The support of y' is now given as $\sigma\left(y'\right) = \sigma\left(x\right) - \left|d\left(y'\right)\right|$. At subsequent levels, we have available the diffsets for each element in the combine list. In this case $d\left(y'\right) = d\left(y\right) - d\left(x\right)$, but the support is still given as $\sigma\left(y'\right) = \sigma\left(x\right) - \left|d\left(y'\right)\right|$. Figure 8 shows the pseudo-code for computing the combine sets using diffsets.

Example 4. Suppose, that we have the itemset *ADT*; we show how to get its support using the diffset propagation technique. In GenMax we start with item *A*, and extend it with item *D*. Now in order to find the support of *AD* we first find the diffset for *AD*, denoted *d*(*AD*) = *t*(*A*) − *t*(*D*) and then calculate its support as a $\sigma\left(AD\right) = \sigma\left(A\right) - \left|d\left(AD\right)\right|$. At the next level, we need to compute the diffset for *ADT* using the diffsets for *AD* and *AT*, where $I_l = \{A\}$ and $C_l, = \{D,T\}$. The diffset of itemset *ADT* is given as *d*(*ADT*) = *d*(*AT*) − *d*(*AD*), and its support is given as $\sigma\left(AD\right) - \left|d\left(ADT\right)\right|$ (Zaki and Gouda, 2003). Since longer patterns are always formed by combining its lexicographic first two subsets, which share the same prefix, the method is guaranteed to be correct.

3.4. Final GenMax algorithm

The complete GenMax algorithm is shown in figure 9, which ties in all the optimizations mentioned above. GenMax assumes that the input dataset is in the vertical tidset format. First GenMax computes the set of frequent items and the frequent 2-itemsets, using a vertical-to-horizontal recovery method (Zaki and Gouda, 2003). This information is used to reorder the items in the initial combine list to minimize the search tree size that is generated. GenMax uses the progressive focusing technique of LMFI-backtrack, combined

GenMax:

1. Compute F_1 and F_2
3. Compute $IF(x)$ for each item $x \in F_1$
4. Sort F_1 (decreasing in $IF(x)$, increasing in $\sigma(x)$)
5. $\mathbf{MFI} = \emptyset$
6. LMFI-backtrack($\emptyset, F_1, \mathbf{MFI}, 0$) //use diffsets
7. **return MFI**

Figure 9. The GenMax algorithm.

Database	I	AL	R	MPL
chess	76	37	3,196	23 (20%)
connect	130	43	67,557	31 (2.5%)
mushroom	120	23	8,124	22 (0.025%)
pumsb*	7117	50	49,046	43 (2.5%)
pumsb	7117	74	49,046	27 (40%)
T10I4D100K	1000	10	100,000	13 (0.01%)
T40I10D100K	1000	40	100,000	25 (0.1%)

Figure 10. Database characteristics: *I* denotes the number of items, *AL* the average length of a record, *R* the number of records, and *MPL* the maximum pattern length at the given *min_sup*.

with diffset propagation of Fl-diffset-combine to produce the exact set of all maximal frequent itemsets, **MFI**.

4. Experimental results

Past work has demonstrated that DepthProject (Agrawal et al., 2000) is faster than MaxMiner (Bayardo, 1998), and the latest paper shows that Mafia (Burdick et al., 2001) consistently beats DepthProject. In our experimental study below, we retain MaxMiner for baseline comparison. At the same time, MaxMiner shows good performance on some datasets, which were not used in previous studies. We use Mafia as the current state-of-the-art method and show how GenMax compares against it. However, *Mafia does not guarantee an exact answer; it returns only a superset of* **MFI**. We thus also compare GenMax with an augmented version of Mafia, called MafiaPP, that returns the exact **MFI**.

All our experiments were performed on a 400 MHz Pentium PC with 256 MB of memory, running RedHat Linux 6.0. For comparison we used the original source or object code for MaxMiner (Bayardo, 1998) and MAFIA (Burdick et al., 2001), provided to us by their authors. Timings in the figures are based on total wall-clock time, and include all preprocessing costs (such as horizontal-to-vertical conversion in GenMax and Mafia).

The times reported also include the program output. We believe our setup reflects realistic testing conditions (as opposed to some previous studies which report only the CPU time or may not include output cost).

4.1. Benchmark datasets

We chose several real and synthetic datasets for testing the performance of the the algorithms, shown in Table 10. The real datasets have been used previously in the evaluation of maximal patterns (Bayardo, 1998; Agrawal et al., 2000; Burdick et al., 2001). Typically, these real datasets are very dense, i.e., they produce many long frequent itemsets even for high values of support. The table shows the length of the longest maximal pattern (at the lowest minimum support used in our experiments) for the different datasets. For example on pumsb*, the longest pattern was of length 43 (any method that mines all frequent patterns will be impractical for such long patterns). We also chose two synthetic datasets, which have been used as benchmarks for testing methods that mine all frequent patterns. Previous maximal set mining algorithms have not been tested on these datasets, which are sparser compared to the real sets. All these datasets are publicly available from IBM Almaden (www.almaden.ibm.com/cs/quest/demos.html).

While conducting experiments comparing the 3 different algorithms, we observed that the performance can vary significantly depending on the dataset characteristics. We were able to classify our benchmark datasets into four classes based on the distribution of the maximal frequent patterns.

4.2. Type I datasets: Chess and pumsb

Figure 11 shows the performance of the three algorithms on chess and pumsb. These Type I datasets are characterized by a symmetric distribution of the maximal frequent patterns (leftmost graph). Looking at the mean of the curve, we can observe that for these datasets most of the maximal patterns are relatively short (average length 11 for chess and 10 for pumsb). The **MFI** cardinality figures on top center and right, show that for the support values shown, the **MFI** is 2 orders of magnitude smaller than all frequent itemsets.

Compare the total execution time for the different algorithms on these datasets (center and rightmost graphs). We use two different variants of Mafia. The first one, labeled Mafia, does not return the exact maximal frequent set, rather it returns a *superset* of all maximal patterns. The second variant, labeled MafiaPP, uses an option to eliminate non-maximal sets in a post-processing (PP) step. Both GenMax and MaxMiner return the exact **MFI**.

On chess we find that Mafia (without PP) is the fastest if one is willing to live with a superset of the **MFI**. Mafia is about 10 times faster than MaxMiner. However, notice how the running time of MafiaPP grows if one tries to find the exact **MFI** in a post-pruning step. GenMax, though slower than Mafia is significantly faster than MafiaPP and is about 5 times faster than MaxMiner. All methods, except MafiaPP, show an exponential growth in running time (since the y-axis is in log-scale, this appears linear) faithfully following the growth of

(over 100 times better in some cases, e.g., chess at 20%). On Type I data MaxMiner is noncompetitive.

*4.3. Type II datasets: Connect and pumsb**

Type II datasets, as shown in figure 12 are characterized by a left-skewed distribution of the maximal frequent patterns, i.e., there is a relatively gradual increase with a sharp drop in the number of maximal patterns. The mean pattern length is also longer than in Type I datasets; it is around 16 or 17. The **MFI** cardinality is also drastically smaller than **FI** cardinality; by a factor of 10^4 or more (in contrast, for Type I data, the reduction was only 10^2).

The main performance trend for both Type II datasets is that Mafia is the best till the support is very low, at which point there is a cross-over and GenMax outperforms Mafia. The reason is most likely due to the benefits of progressive focusing at low support thresholds. MafiaPP continues to be favorable for higher supports, but once again beyond a point post-pruning costs start to dominate. MafiaPP could not be run beyond the plotted points in any

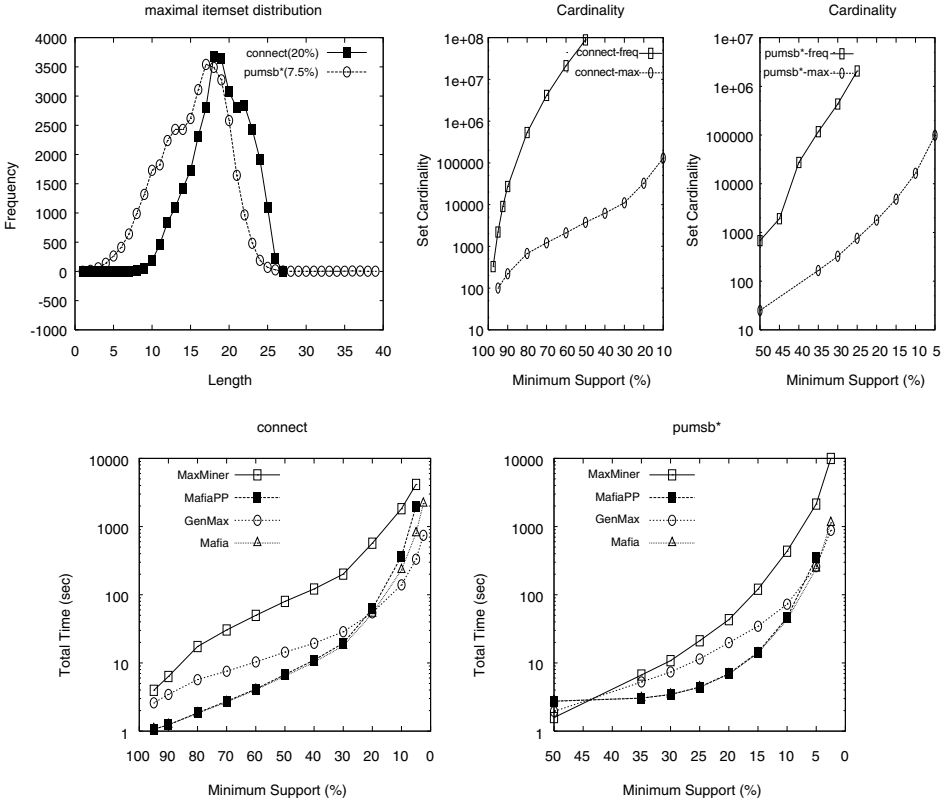

Figure 12. Type II datasets (connect and pumsb∗).

reasonable amount of time. MaxMiner remains noncompetitive (about 10 times slower). The initial start-up time for Mafia for creating the bit-vectors is responsible for the high offset at 50% support on pumsb*. GenMax appears to exhibit a more graceful increase in running time than Mafia.

4.4. Type III datasets: T10I4 and T40I10

As depicted in figure 13, Type III datasets—the two synthetic ones—are characterized by an exponentially decaying distribution of the maximal frequent patterns. Except for a few maximal sets of size one, the vast majority of maximal patterns are of length two! After that the number of longer patterns drops exponentially. The mean pattern length is very short compared to Type I or Type II datasets; it is around 4–6. **MFI** cardinality is not much smaller than the cardinality of all frequent patterns. The difference is only a factor of 10 compared to a factor of 100 for Type I and a factor of 10,000 for Type II.

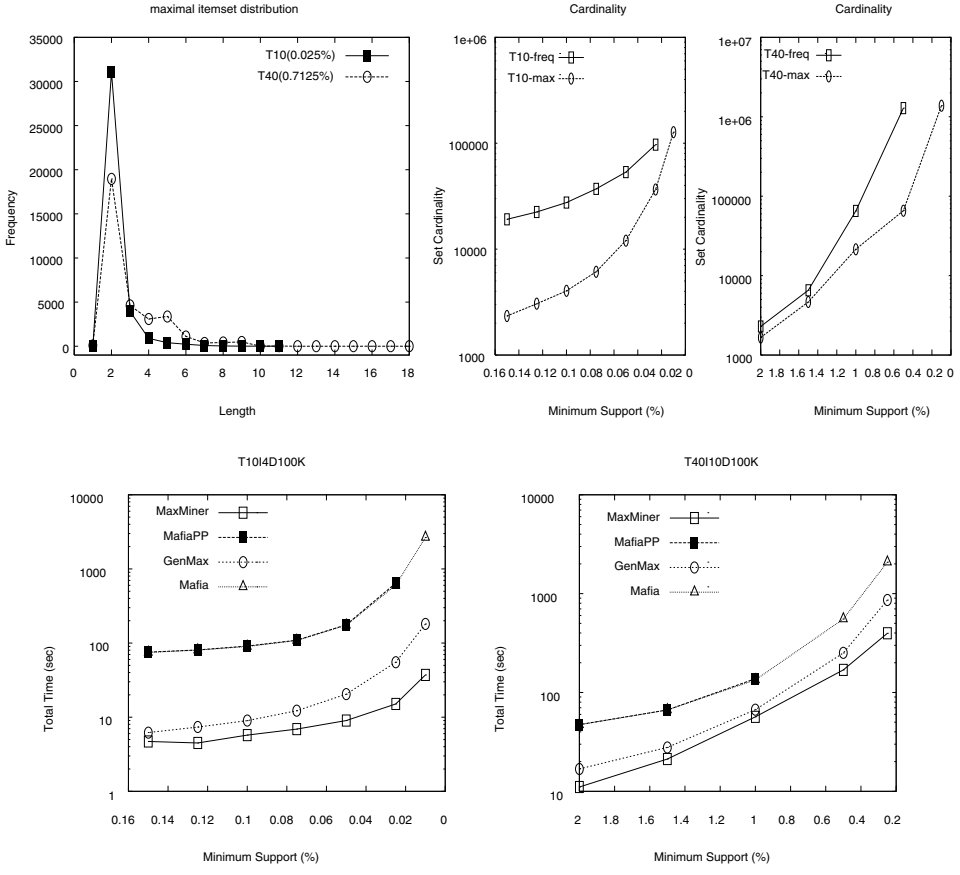

Figure 13. Type III datasets (T10 and T40).

Comparing the running times we observe that MaxMiner is the best method for this type of data. The breadth-first or level-wise search strategy used in MaxMiner is ideal for very bushy search trees, and when the average maximal pattern length is small. Horizontal methods are better equipped to cope with the quadratic blowup in the number of frequent 2-itemsets since one can use array based counting to get their frequency. On the other hand vertical methods spend much time in performing intersections on long item tidsets or bit-vectors. GenMax gets around this problem by using the horizontal format for computing frequent 2-itemsets (denoted F_2), but it still has to spend time performing $O(|F_2|)$ pairwise tidset intersections.

Mafia on the other hand performs $O(|F_1|^2)$ intersections, where F_1 is the set of frequent items. The overhead cost is enough to render Mafia noncompetitive on Type III data. On T10 Mafia can be 20 or more times slower than MaxMiner. GenMax exhibits relatively good performance, and it is about 10 times better than Mafia and 2 to 3 times worse than MaxMiner. On T40, the gap between GenMax/Mafia and MaxMiner is smaller since there are longer maximal patterns. MaxMiner is 2 times better than GenMax and 5 times better than Mafia. Since the **MFI** cardinality is not too large MafiaPP has almost the time as Mafia for high supports. Once again MafiaPP could not be run for lower support values. It is clear that, in general, post-pruning is not a good idea; the overhead is too much to cope with.

4.5. Type IV dataset: Mushroom

Mushroom exhibits a very unique **MFI** distribution. Plotting **MFI** cardinality by length, we observe in figure 14 that the number of maximal patterns remains small until length 19. Then there is a sudden explosion of maximal patterns at length 20, followed by another sharp drop at length 21. The vast majority of maximal itemsets are of length 20. The average transaction length for mushroom is 23 (see Figure 10), thus a maximal pattern spans almost a full transaction. The total **MFI** cardinality is about 1000 times smaller than all frequent itemsets.

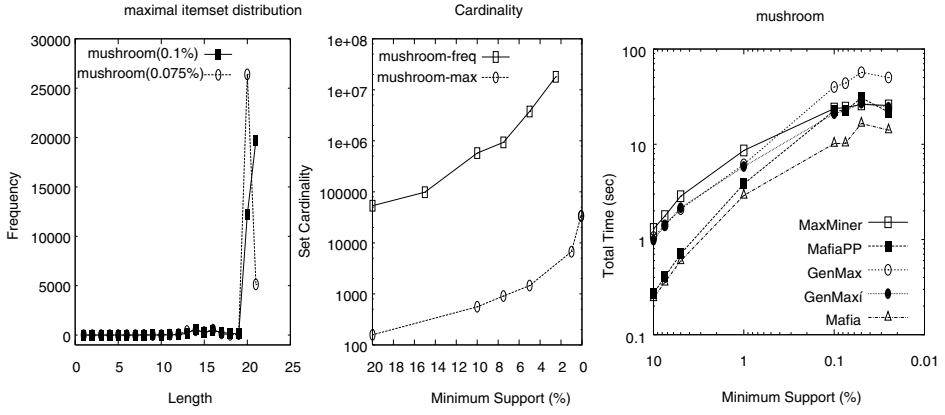

Figure 14. Type IV dataset (mushroom).

On Type IV data, Mafia performs the best. MafiaPP and MaxMiner are comparable at lower supports. This data is the worst for GenMax, which is 2 times slower than MaxMiner and 4 times slower than Mafia. In Type IV data, a smaller itemset is part of many maximal itemsets (of length 20 in case of mushroom); this renders our progressive focusing technique less effective. To perform maximality checking one has to test against a large set of maximal itemsets; we found that GenMax spends half its time in maximality checking. Recognizing this helped us improve the progressive focusing using an optimized intersection-based method (as opposed to the original list based approach). This variant, labeled GenMax', was able to cut down the execution time by half. GenMax' runs in the same time as MaxMiner and MafiaPP.

5. Conclusions

This is one of the first papers to comprehensively compare recent maximal pattern mining algorithms under realistic assumptions. Our timings are based on wall-clock time, we included all pre-processing costs, and also cost of outputting all the maximal itemsets (written to a file). We were able to distinguish four different types of **MFI** distributions in our benchmark testbed. We believe these distributions to be fairly representative of what one might see in practice, since they span both real and synthetic datasets. Type I is a normal **MFI** distribution with not too long maximal patterns, Type II is a left-skewed distributions, with longer maximal patterns, Type III is an exponential decay distribution, with extremely short maximal patterns, and finally Type IV is an extreme left-skewed distribution, with very large average maximal pattern length.

We noted that different algorithms perform well under different distributions. We conclude that among the current methods, MaxMiner is the best for mining Type III distributions. On the remaining types, Mafia is the best method if one is satisfied with a superset of the **MFI**. For very low supports on Type II data, Mafia loses its edge. Post-pruning non-maximal patterns typically has high overhead. It works only for high support values, and MafiaPP cannot be run beyond a certain minimum support value. GenMax integrates pruning of non-maximal itemsets in the process of mining using the novel progressive focusing technique, along with other optimizations for superset checking; among the methods tested GenMax is the best method for mining the exact **MFI**.

Our work opens up some important avenues of future work. The IBM synthetic dataset generator appears to be too restrictive. It produces Type III **MFI** distributions. We plan to develop a new generator that the users can use to produce various kinds of **MFI** distributions. This will help provide a common testbed against which new algorithms can be benchmarked. Knowing the conditions under which a method works well or does not work well is an important step in developing new solutions. In contrast to previous studies we were able to isolate these conditions for the different algorithms. For example, we were able to improve the performance of GenMax' to match MaxMiner on mushroom dataset. Another obvious avenue of improving GenMax and Mafia is to efficiently handle Type III data. It seems possible to combine the strengths of the three methods into a single hybrid algorithm that uses the horizontal format when required and uses bit-vectors/diffsets or perhaps bit-

vectors of diffsets in other cases or in combination. We plan to investigate this in the future.

Acknowledgments

We would like to thank Roberto Bayardo for providing us the MaxMiner algorithm and Johannes Gehrke for the MAFIA algorithm. This work was supported in part by NSF CAREER Award IIS-0092978, DOE Career Award DE-FG02-02ER25538, and NSF grants EIA-0103708 and EMT-0432098.

References

Agrawal, R., Aggarwal, C., and Prasad, V. 2000. Depth first generation of long patterns. In 7th Int'l Conference on Knowledge Discovery and Data Mining, pp. 108–118.

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A.I. 1996. Fast discovery of association rules. In Advances in Knowledge Discovery and Data Mining, Fayyad, U. et al. (Eds.), Menlo Park, CA: AAAI Press, pp. 307–328.

Bayardo, R.J. 1998. Efficiently mining long patterns from databases. In ACM SIGMOD Conf. on Management of Data, pp. 85–93.

Burdick, D., Calimlim, M., and Gehrke, J. 2001. MAFIA: A maximal frequent itemset algorithm for transactional databases. In IEEE Intl. Conf. on Data Engineering, pp. 443–452.

Goethals, B., and Zaki, M. 2003. Advances in frequent itemset mining implementations: Report on FIMI'03. SIGKDD Explorations, 6(1):109–117.

Gunopulos, D., Khardon, R., Mannila, H., Saluja, S., Toivonen, H., and Sharma, R. 2003. Discovering all most specific sentences. ACM Transactions on Database Systems, 28(2):140–174.

Han, J., Pei, J., and Yin, Y. 2000. Mining frequent patterns without candidate generation. In ACM SIGMOD Conf. on Management of Data, pp. 1–12.

Lin, D.-L., and Kedem, Z.M. 1998. Pincer-search: A new algorithm for discovering the maximum frequent set. In 6th Intl. Conf. on Extending Database Technology, pp. 105–119.

Yellin, D. 1994. An algorithm for dynamic subset and intersection testing. Theoretical Computer Science, 129:397–406.

Zaki, M.J. 2000. Generating non-redundant association rules. In 6th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining, pp. 34–43.

Zaki, M.J., and Gouda, K. 2003. Fast vertical mining using Diffsets. In 9th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining, pp. 326–335.

Zaki, M.J., and Hsiao, C.-J. 2002. CHARM: An efficient algorithm for closed itemset mining. In 2nd SIAM International Conference on Data Mining, pp. 457–473.