MAFIA: A Performance Study of Mining Maximal Frequent Itemsets

Doug Burdick [*]
University of Wisconsin-Madison
who0ps99@cs.wisc.edu

Manuel Calimlim
Cornell University
calimlim@cs.cornell.edu

Jason Flannick [†]
Stanford University
flannick@cs.stanford.edu

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Tomi Yiu
Cornell University
ty42@cornell.edu

Abstract

We present a performance study of the MAFIA algorithm for mining maximal frequent itemsets from a transactional database. In a thorough experimental analysis, we isolate the effects of individual components of MAFIA, including search space pruning techniques and adaptive compression. We also compare our performance with previous work by running tests on very different types of datasets. Our experiments show that MAFIA performs best when mining long itemsets and outperforms other algorithms on dense data by a factor of three to thirty.

1 Introduction

MAFIA uses a vertical bitmap representation for support counting and effective pruning mechanisms for searching the itemset lattice [6]. The algorithm is designed to mine maximal frequent itemsets (*MFI*), but by changing some of the pruning tools, MAFIA can also generate all frequent itemsets (*FI*) and closed frequent itemsets (*FCI*).

MAFIA assumes that the entire database (and all data structures used for the algorithm) completely fit into main memory. 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 some important performance bottlenecks.

In a thorough experimental evaluation, we first quantify the effect of each individual pruning component on the performance of MAFIA. Because of our strong pruning mechanisms, MAFIA performs best on dense datasets where large subtrees can be removed from the search space. On shallow datasets, MAFIA is competitive though not always the fastest algorithm. On dense datasets, our results indicate that MAFIA outperforms other algorithms by a factor of three to thirty.

[*]Research for this paper done while at Cornell University
[†]Research for this paper done while at Cornell University

2 Search Space Pruning

MAFIA uses the lexicographic subset tree originally presented by Rymon [9] and adopted by both Agarwal [3] and Bayardo [4]. 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.

In the simplest itemset traversal, 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 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.

For each candidate itemset, 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, itemsets already inserted into the **MFI** will be lexicographically ordered earlier.

2.1 Parent Equivalence Pruning (PEP)

One method of pruning involves comparing the transaction sets of each parent/child pair. Let x be a node n's head and y be an element in n's tail. If $t(x) \subseteq t(y)$, then any transaction containing x also contains 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 node's tail to the node's head.

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 [4]. 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* (Frequent Head Union Tail) pruning.

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*.

2.4 Dynamic Reordering

The benefit of dynamically reordering the children of each node based on support instead of following the lexicographic order is significant. An algorithm that trims the tail to only frequent extensions at a higher level will save a lot of computation. The order of the tail elements is also an important consideration. Ordering the tail elements by increasing support will keep the search space as small as possible. This heuristic was first used by Bayardo [4].

In Section 5.3.1, we quantify the effects of the algorithmic components by analyzing different combinations of pruning mechanisms.

3 MAFIA Extensions

MAFIA is designed and optimized for mining maximal frequent itemsets, but the general framework can be used to mine all frequent itemsets and closed frequent itemsets.

The algorithm can easily be extended to mine all frequent itemsets. The main changes required are suppressing any pruning tools (PEP, FHUT, HUTMFI) and adding all frequent nodes in the itemset lattice to the set *FI* without any superset checking. Itemsets are counted using the same techniques as for the regular MAFIA algorithm.

MAFIA can also be used to mine closed frequent itemsets. An itemset is *closed* if there are no supersets with the same support. PEP is the only type of pruning used when mining for frequent closed itemsets (*FCI*). Recall from Section 2.1 that PEP moves all extensions with the same support from the tail to the head of each node. Any items remaining in the tail must have a lower support and thus are different closed itemsets. Note that we must still check for supersets in the previously discovered **FCI**.

4 Optimizations

4.1 Effective MFI Superset Checking

In order to enumerate the exact set of maximally frequent itemsets, before adding any itemset to the *MFI* we must check the entire *MFI* to ensure that no superset of the itemset has already been found. This check is done often, and significant performance improvements can be realized if it is done efficiently. To ensure this, we adopt the *progressive focusing* technique introduced by Gouda and Zaki [7].

The basic idea is that while the entire *MFI* may be large, at any given node only a fraction of the *MFI* are possible supersets of the itemset at the node. We therefore maintain for each node a *LMFI* (Local MFI), which is the subset of the *MFI* that contains supersets of the current node's itemset. For more details on the *LMFI* concept, please see the paper by Gouda and Zaki [7].

4.2 Support Counting and Bitmap Compression

MAFIA uses a vertical bitmap representation for the database [6]. 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. Generation of new itemset bitmaps involves bitwise-ANDing *bitmap(X)* with a bitmap for 1-itemset Y and storing the result in *bitmap ($X \cup Y$)*. For each byte in *bitmap($X \cup Y$)*, the number of 1's in the byte is determined using a pre-computed table. Summing these lookups gives the support of $(X \cup Y)$.

4.3 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 because both the absence and presence of the itemset in a transaction need to be represented. However, note that we only need information about transactions containing the itemset X to count the support of the subtree rooted at node N. So, conceptually we can remove the bit for transaction T from X if T does not contain X. This is

Dataset	T	I	ATL
T10I4D100K	100,000	1,000	10
T40I10D100K	100,000	1,000	40
BMS-POS	515,597	1,657	6.53
BMS-WebView-1	59,602	497	2.51
BMS-WebView-2	3,340	161	4.62
chess	3196	76	37
connect4	67,557	130	43
pumsb	49,046	7,117	74
pumsb-star	49,046	7,117	50

T = Numbers of transactions
I = Numbers of items
ATL = Average transaction length

Figure 1. Dataset Statistics

a form of lossless compression on the vertical bitmaps to speed up calculations.

4.3.1 Adaptive Compression

Determining when to compress the bitmaps is not as simple as it first appears. Each 1-extension bitmap in the tail of the node N must be projected relative to the itemset X, and the cost for projection may outweigh the benefits of using the compressed bitmaps. The best approach is to compress only when we know that the savings from using the compressed bitmaps outweigh the cost of projection.

We use an adaptive approach to determine when to apply compression. At each node, we estimate both the cost of compression and the benefits of using the compressed bitmaps instead of the full bitmaps. When the benefits outweight the costs, compression is chosen for that node and the subtree rooted at that node.

5 Experimental Results

5.1 Datasets

To test MAFIA, we used three different types of data. The first group of datasets is sparse; the frequent itemset patterns are short and thus nodes in the itemset tree will have small tails and few branches. We first used artificial datasets that were created using the data generator from IBM Almaden [1]. Stats for these datasets can be found in Figure 1 under *T10I4D100K* and *T40I10D100K*. The distribution of maximal frequent itemsets is displayed in Figure 2. For all datasets, the minimum support was chosen to yield around 100,000 elements in the MFI. Note that both T10I4 and T40I10 have very high concentrations of itemsets around two and three items long with T40I10 having another smaller peak around eight to nine items.

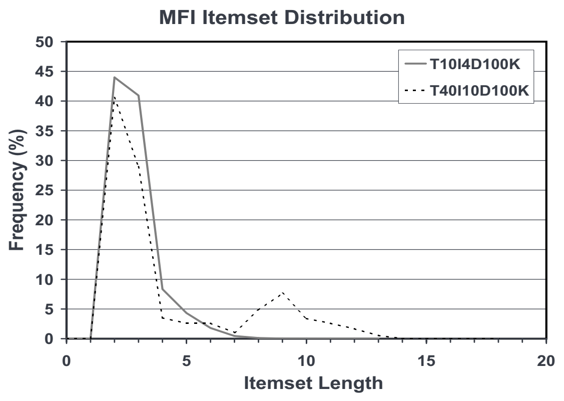


Figure 2. Itemset Lengths for shallow, artificial datasets

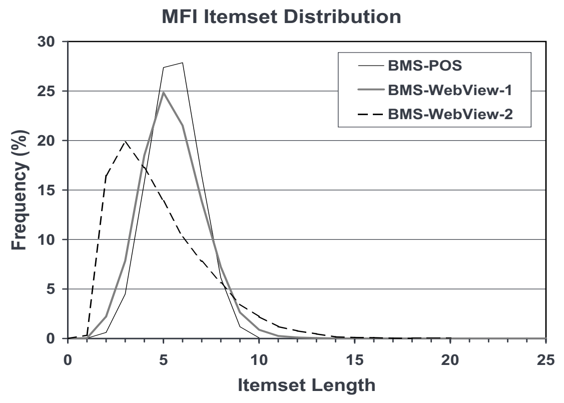


Figure 3. Itemset Lengths for shallow, real datasets

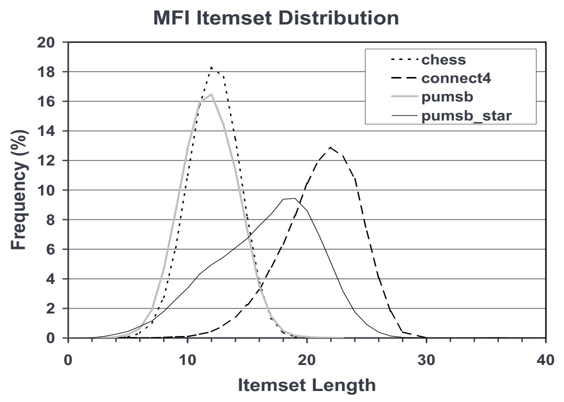


Figure 4. Itemset Lengths for dense, real datasets

The second dataset type is click stream data from two different e-commerce websites (*BMS-WebView-1* and *BMS-WebView-2*) where each transaction is a web session and each item is a product page view; this data was provided by Blue Martini [8]. *BMS-POS* contains point-of-sale data from an electronics retailer with the item-ids corresponding to product categories. Figure 3 shows that BMS-POS and BMS-WebView-1 have very similar normal curve itemset distributions with the average length of a maximal frequent itemset around five to six items long. On the other hand, BMS-WebView-2 has a right skewed distribution; there's a sharp incline until three items and then a more gradual decline on the right tail.

Finally, the last datasets used for analysis are the dense datasets. They are characterized by very long itemset patterns that peak around 10-25 items (see Figure 4). *Chess* and *Connect4* are gathered from game state information and are available from the UCI Machine Learning Repository [5]. The *Pumsb* dataset is census data from PUMS (Public Use Microdata Sample). *Pumsb-star* is the same dataset as *Pumsb* except all items of 80% support or more have been removed, making it less dense and easier to mine. Figure 4 shows that Chess and Pumsb have nearly identical itemset distributions that are normal around 10-12 items long. Connect4 and Pumsb-star are somewhat left-skewed with a slower incline that peaks around 20-23 items and then a sharp decline in the length of the frequent itemsets.

5.2 Other Algorithms

5.2.1 DepthProject

DepthProject demonstrated an order of magnitude improvement over previous algorithms for mining maximal frequent itemsets [2]. MAFIA was originally designed with DepthProject as the primary benchmark for comparison and we have implemented our own version of the DepthProject algorithm for testing.

The primary differences between MAFIA and DepthProject are the database representation (and consequently the support counting) and the application of pruning tools. DepthProject uses a horizontal database layout while MAFIA uses a vertical bitmap format, and supports of itemsets are counted very differently. Both algorithms use some form of compression when the bitmaps become sparse. However, DepthProject also utilizes a specialized counting technique called bucketing for the lower levels of the itemset lattice. When the tail of a node is small enough, bucketing will count the entire subtree with one pass over the data. Since bucketing counts all of the nodes in a subtree, many itemsets that MAFIA will prune out will be counted with DepthProject. For more details on the DepthProject algorithm, please refer to the paper by Agarwal and Aggarwal [2].

5.2.2 GenMax

GenMax is a new algorithm by Gouda and Zaki for finding maximal itemset patterns [7]. GenMax introduced a novel concept for finding supersets in the *MFI* called *progessive focusing*. The newest version of MAFIA has incorporated this technique with the *LMFI* update. GenMax also uses diffset propagation for fast support counting. Both algorithms use similar methods for itemset lattice exploration and pruning of the search space.

5.3 Experimental Analysis

We performed three types of experiments to analyze the performance of MAFIA. First, we analyze the effect of each pruning component of the MAFIA algorithm to demonstrate how the algorithm works to trim the search space of the itemset lattice. The second set of experiments examines the savings generated by using compression to speed support counting. Finally, we compare the performance of MAFIA against other current algorithms on all three types of data (see Section 5.1). In general, MAFIA works best on dense data with long itemsets, though the algorithm is still competitive on even very shallow data.

These experiments were conducted on a 1500 Mhz Pentium with 1GB of memory running Redhat Linux 9.0. All code was written in C++ and compiled using gcc version 3.2 with all optimizations enabled.

5.3.1 Algorithmic Component Analysis

First, we present a full analysis of each pruning component of the MAFIA algorithm (see Section 2 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 can significantly reduce the size of the search space by removing infrequent items from each node's tail.

Figures 5 and 6 show the effects of each component of the MAFIA algorithm on the Connect4 dataset at 40% minimum support. The components of the algorithm are represented in a *cube* format with the running times (in seconds) and the number of itemsets counted during the MAFIA search. 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 of the pruning components yield great savings in running time compared to using no pruning. Applying a single pruning mechanism runs two to three orders of

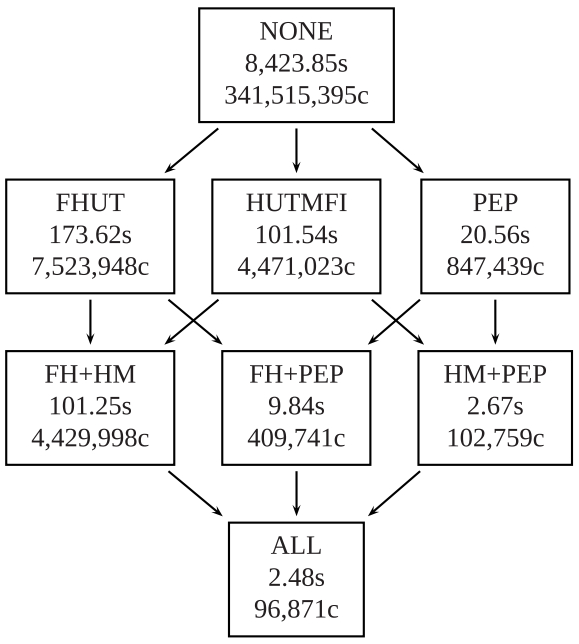


Figure 5. Pruning Components for Connect4 at 40% support *without reordering*

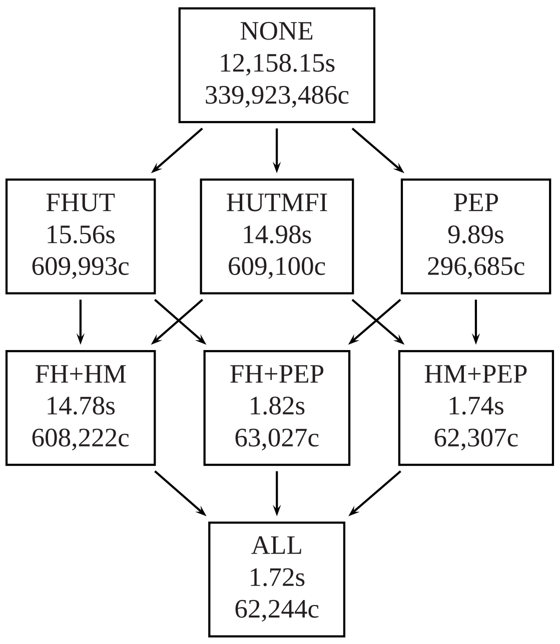


Figure 6. Pruning Components for Connect4 at 40% support *with reordering*

magnitude faster while using all of the pruning tools is four orders of magnitude faster than no pruning.

Several of the pruning components seem to overlap in trimming the search space. In particular, HUTMFI and FHUT yield very similar results, since they use the same type of superset pruning but with different methods of implementation. It is interesting to see that adding FHUT when HUTMFI is already performed yields very little savings, i.e. from HUTMFI to FH+HM or from HM+PEP to ALL, the running times do not significantly change. HUTMFI first 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 largest impact of the three pruning methods. Most of the running time of the algorithm occurs at the lower levels of the tree where the border between frequent and infrequent itemsets exists. Near this border, many of the itemsets have the same exact support right above the minimum support and thus, PEP is more likely to trim out large sections of the tree at the lower levels.

Dynamically reordering the tail also has dramatic savings (cf. Figure 5 with Figure 6). At the top of each cube, it is interesting to note that without any pruning mechanisms, dynamic reordering will actually run slower than static ordering. Fewer itemsets get counted, but the cost of reordering so many nodes outweighs the savings of counting fewer nodes.

However, once pruning is applied, dynamic reordering runs nearly an order of magnitude faster than the static ordering. PEP is more effective since the tail is trimmed as early in the tree as possible; all of the extensions with the same support are moved from the tail to the head in one step at the start of the subtree. Also, FHUT and HUTMFI have much more impact. With dynamic reordering, subtrees generated from the end of tail have the itemsets with the highest supports and thus the HUT is more likely to be frequent.

5.3.2 Effects of Compression in MAFIA

Adaptive compression uses cost estimation to determine when it is appropriate to compress the bitmaps. Since the cost estimate adapts to each dataset, adaptive compression is always better than using no compression. Results on different types of data show that adaptive compression is at least 25% faster as higher supports and at lower supports up to an order of magnitude faster.

Figures 7 and 8 display the effect of compression on sparse data. First, we analyze the sparse, artificial datasets T10I4 and T40I10 that are characterized by very short itemsets, where the average length of maximally frequent itemsets is only 2-6 items. Because these datasets are so sparse with small subtrees, at higher supports compression is not

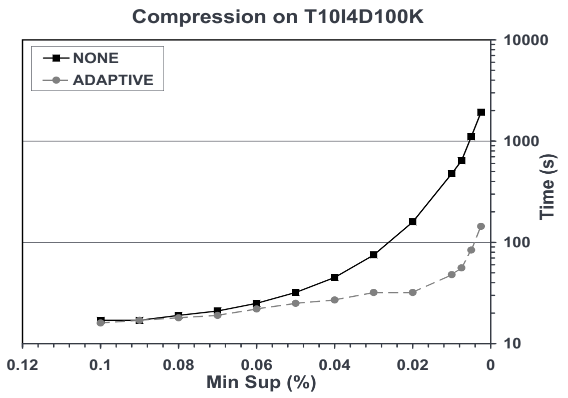


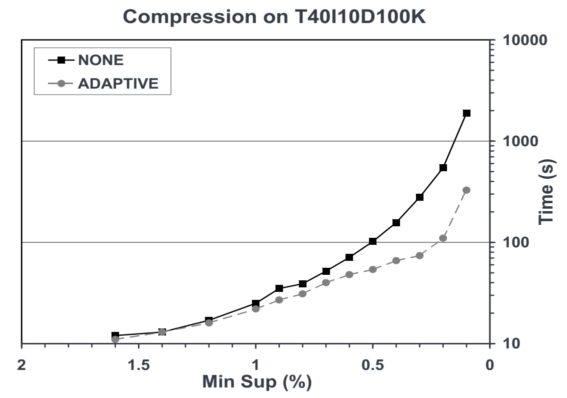
Figure 7. Compression on sparse datasets

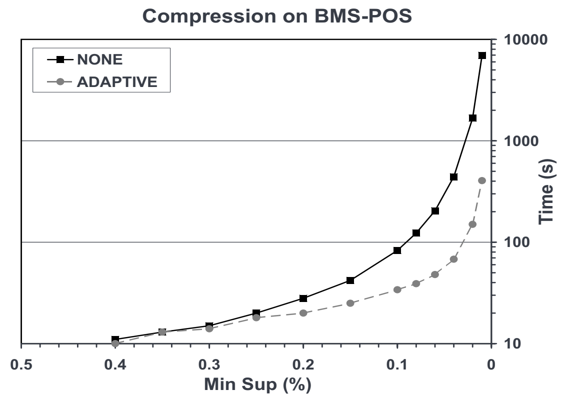
often used and thus has a negligible effect. But as the support drops and the subtrees grow larger, the effect of compression is enhanced and the running times for adaptive compression increase to nearly 3-10 times faster.

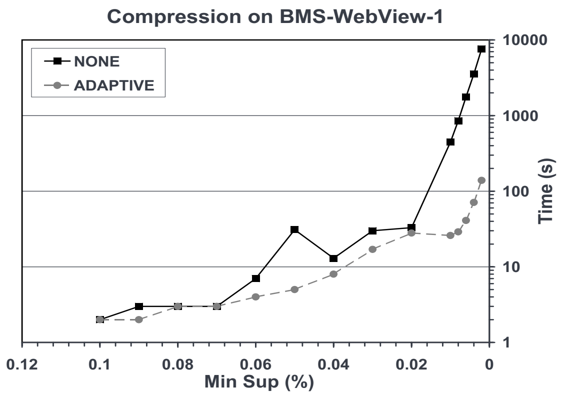
Next are the results on the sparse, real datasets: BMS-POS, BMS-WebView-1, and BMS-WebView-2 in Figure 8. Note that for BMS-POS, adaptive compression follows the exact same pattern as the synthetic datasets with the difference growing from negligible to over 10 times better. BMS-WebView-1 follows the same general pattern except for an anomalous spike in the running times without compression around .05%. However, for BMS-WebView-2 compression has a very small impact and is only really effective at the lowest supports. Recall from Figure 3 that BMS-WebView-2 has a right-skewed distribution of frequent itemsets, which may help explain the different compression effect.

The final group of datasets is found in Figure 9 and shows the results of compression on dense, real data. The results on Chess and Pumsb indicate that very few compressed bitmaps were used; apparently, the adaptive compression algorithm determined compression to be too expensive. As a result, adaptive compression is only around 15-30% better than using no compression at all. On the other hand, the Connect4 and Pumsb-star datasets use a much higher ratio of compressed bitmaps and adaptive compression is more than three times faster than no compression.

It is interesting to note that Chess and Pumsb both have left-skewed distributions (see Figure 4) while Connect4 and Pumsb-star follow a more normal distribution of itemsets. The results indicate that when the data is skewed (left or right), adaptive compression is not as effective. Still, even in the worst case adaptive compression will use the cost estimate to determine that compression should not be chosen and thus is at least as fast as never compressing at all. In the best case, compression can significantly speed up support







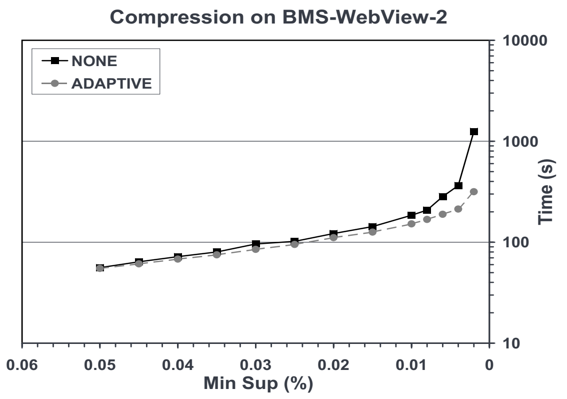
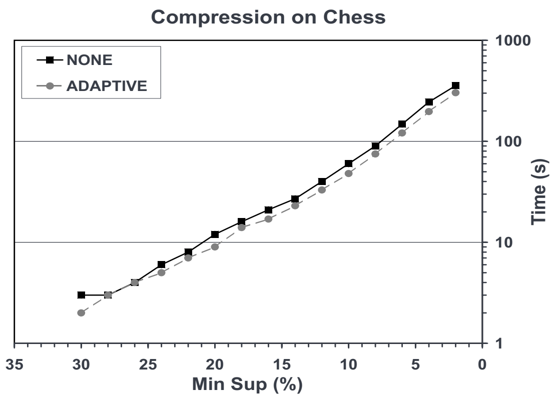
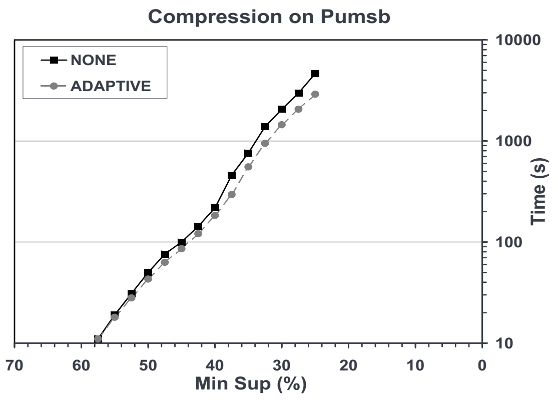
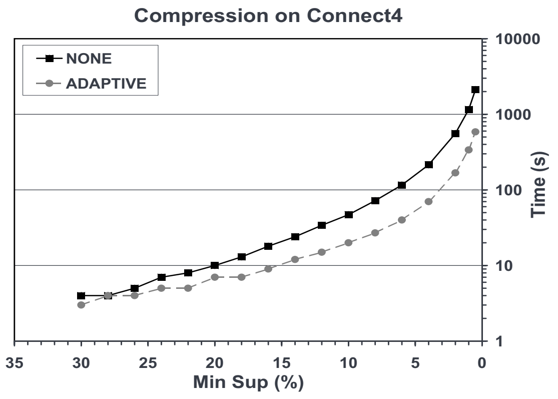


Figure 8. Compression on more sparse datasets







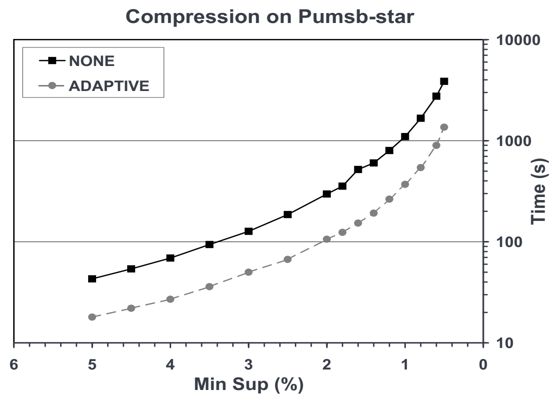


Figure 9. Compression on dense datasets

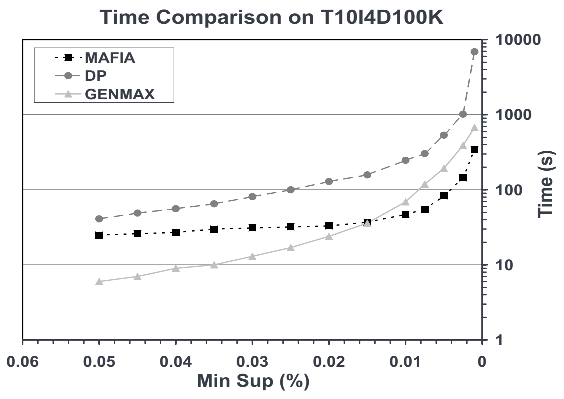


Figure 10. Performance on sparse datasets

counting by over an order of magnitude.

5.3.3 Performance Comparisons

Figures 10 and 11 show the results of comparing MAFIA with DepthProject and GenMax on sparse data. MAFIA is always faster than DepthProject and grows from twice as fast at the higher supports to more than 20 times faster at the lowest supports tested. GenMax demonstrates the best performance of the three algorithms for higher supports and is around two to three times faster than MAFIA. However, note that as the support drops and the itemsets become longer, MAFIA passes Genmax in performance to become the fastest algorithm.

The performances for sparse, real datasets are found in Figure 11. MAFIA has the worst performance on BMS-WebView-2 for higher supports, though it eventually passes DepthProject as the support lowers. BMS-POS and BMS-WebView-1 follow a similar pattern to the synthetic datasets where MAFIA is always better than DepthProject, and GenMax is better than MAFIA until the lower supports where they cross over. In fact, at the lowest supports for BMS-WebView-1, MAFIA is an order of magnitude better than GenMax and over 50 times faster than DepthProject. It is clear that MAFIA performs best when the itemsets are longer, though even for sparse data MAFIA is within two to three times the running times of DepthProject and GenMax.

The dense datasets in Figure 12 support the idea that MAFIA runs the fastest on longer itemsets. For all supports on the dense datasets, MAFIA has the best performance. MAFIA runs around two to five times faster than GenMax on Connect4, Pumsb, and Pumsb-star and over five to ten times faster on Chess. DepthProject is by far the slowest algorithm on all of the dense datasets and runs between ten to thirty times worse than MAFIA on all of the datasets across all supports.

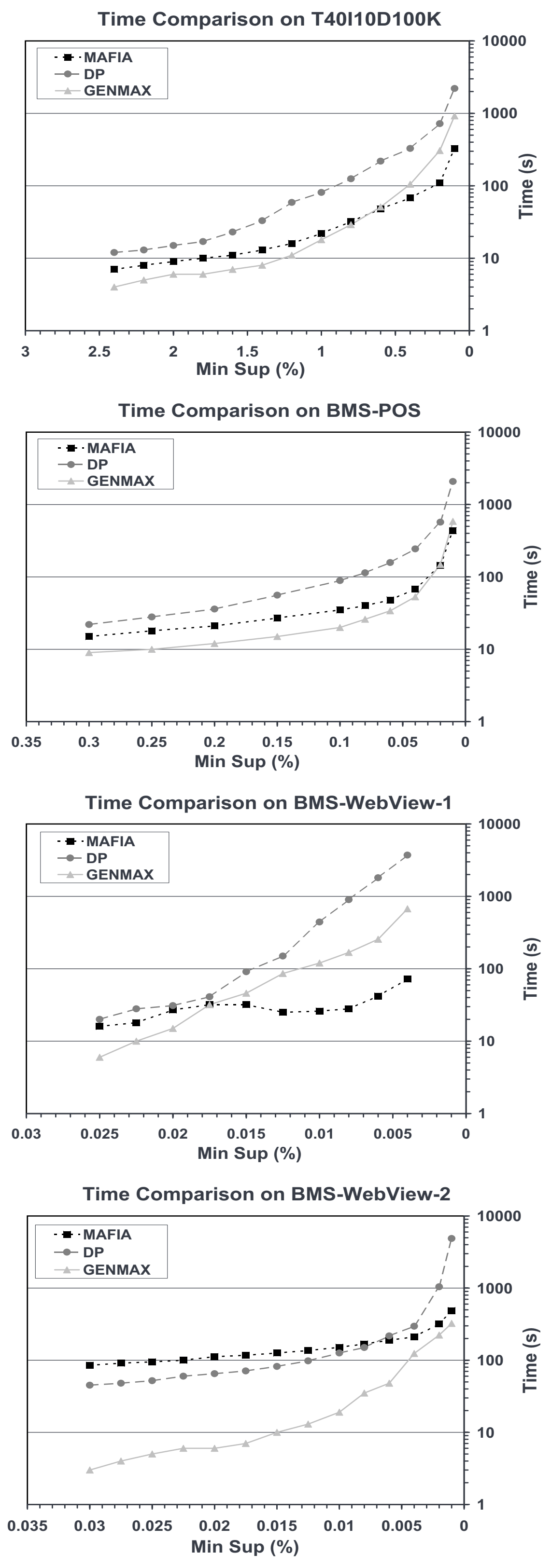


Figure 11. Performance on more sparse datasets

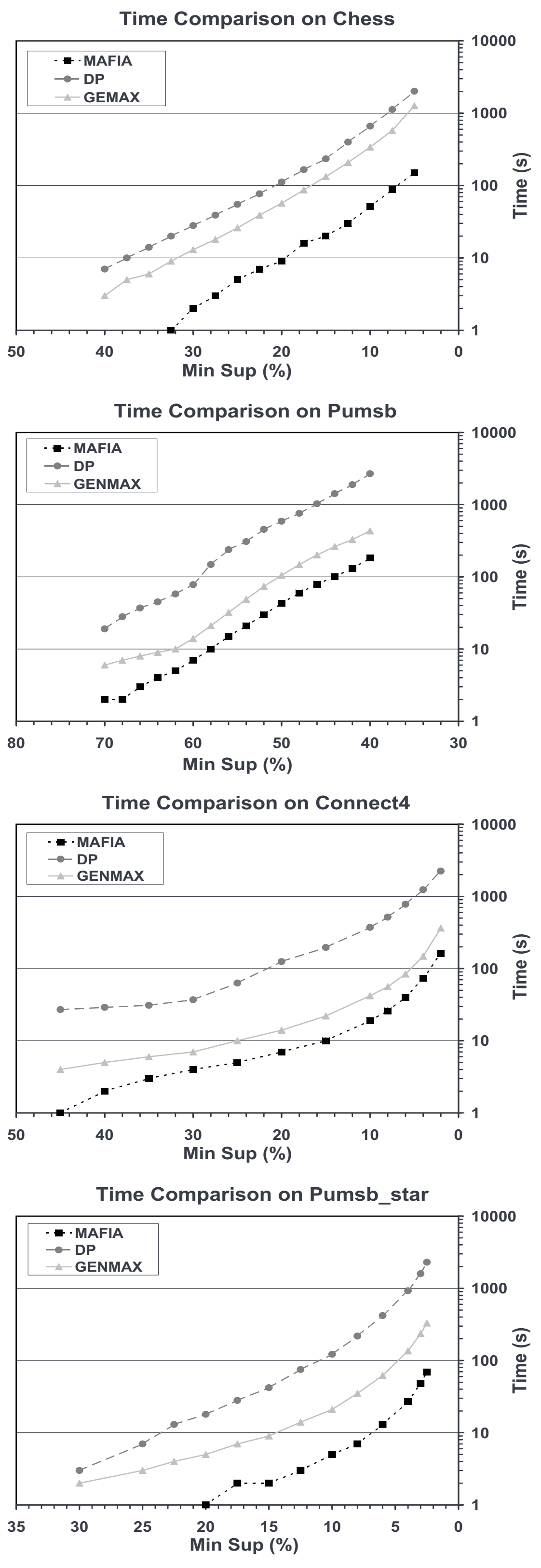


Figure 12. Performance on dense datasets

6 Conclusion

In this paper we present a detailed performance analysis of MAFIA. The breakdown of the algorithmic components show that powerful pruning techniques such as parent-equivalence pruning and superset checking are very beneficial in reducing the search space. We also show that adaptive compression/projection of the vertical bitmaps dramatically cuts the cost of counting supports of itemsets. Our experimental results demonstrate that MAFIA is highly optimized for mining long itemsets and on dense data consistently outperforms GenMax by two to ten and DepthProject by ten to thirty.

Acknowledgements: We would like to thank Ramesh Agarwal and Charu Aggarwal for discussing DepthProject and giving us advice on its implementation. We also thank Jayant Haritsa for his insightful comments on the MAFIA algorithm, Jiawei Han for helping in our understanding of CLOSET and providing us the executable of the FP-Tree algorithm, and Mohammed Zaki for making the source code of GenMax available.

References

[1] Data generator available at http://www.almaden.ibm.com/software/quest/Resources/.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[3] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.

[5] C. Blake and C. Merz. UCI repository of machine learning databases, 1998.

[6] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE 2001*, Heidelberg, Germany, 2001.

[7] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[8] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. http://www.ecn.purdue.edu/KDDCUP.

[9] R.Rymon. Search through systematic set enumeration. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.