AIM: Another Itemset Miner

Amos Fiat, Sagi Shporer
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel
{fiat, shporer}@tau.ac.il

Abstract

We present a new algorithm for mining frequent itemsets. Past studies have proposed various algorithms and techniques for improving the efficiency of the mining task. We integrate a combination of these techniques into an algorithm which utilize those techniques dynamically according to the input dataset. The algorithm main features include depth first search with vertical compressed database, diffset, parent equivalence pruning, dynamic reordering and projection. Experimental testing suggests that our algorithm and implementation significantly outperform existing algorithms/implementations.

1. Introduction

Finding association rules is one of the driving applications in data mining, and much research has been done in this field [10, 7, 4, 6]. Using the support-confidence framework, proposed in the seminal paper of [1], the problem is split into two parts — (a) finding frequent itemsets, and (b) generating association rules.

Let I be a set of items. A subset $X \subseteq I$ is called an itemset. Let D be a transactional database, where each transaction $T \in D$ is a subset of $I : T \subseteq I$. For an itemset X, support(X) is defined to be the number of transactions T for which $X \subseteq T$. For a given parameter minsupport, an itemset X is call a *frequent itemset* if support$(X) \geq$ minsupport. The set of all frequent itemsets is denoted by $\mathcal{F}$.

The remainder of this paper is organized as follows. Section 2 contains a short of related work. In section 3 we describe the AIM-$\mathcal{F}$ algorithm. Section 4 contains experimental results. In Section 5 we conclude this short abstact with a discussion.

1.1. Contributions of this paper

We combine several pre-existing ideas in a fairly straightforward way and get a new frequent itemset mining algorithm. In particular, we combine the sparse vertical bit vector technique along with the difference sets technique of [14], thus reducing the computation time when compared with [14]. The various techniques were put in use dynamically according to the input dataset, thus utilizing the advantages and avoiding the drawbacks of each technique.

Experimental results suggest that for a given level of support, our algorithm/implementation is faster than the other algorithms with which we compare ourselves. This set includes the dEclat algorithm of [14] which seems to be the faster algorithm amongst all others.

2. Related Work

Since the introduction of the *Apriori* algorithm by [1, 2] many variants have been proposed to reduce time, I/O and memory.

Apriori uses breath-first search, bottom-up approach to generate frequent itemsets. (*I.e.*, constructs $i+1$ item frequent itemsets from i item frequent itemsets). The key observation behind Apriori is that all subsets of a frequent itemset must be frequent. This suggests a natural approach to generating frequent itemsets. The breakthrough with Apriori was that the number of itemsets explored was polynomial in the number of frequent itemsets. In fact, on a worst case basis, Apriori explores no more than n itemsets to output a frequent itemset, where n is the total number of items.

Subsequent to the publication of [1, 2], a great many variations and extensions were considered [3, 7, 13]. In [3] the number of passes over the database was reduced . [7] tried to reduce the search space by combining bottom-up and top-down search – if a set is infre-

quent than so are supersets, and one can prune away infrequent itemsets found during the top-down search. [13] uses equivalence classes to skip levels in the search space. A new mining technique, FP-Growth, proposed in [12], is based upon representing the dataset itself as a tree. [12] perform the mining from the tree representation.

We build upon several ideas appearing in previous work, a partial list of which is the following:

- Vertical Bit Vectors [10, 4] - The dataset is stored in vertical bit vectors. Experimentally, this has been shown to be very effective.
- Projection [4] - A technique to reduce the size of vertical bit vectors by trimming the bit vector to include only transaction relevant to the subtree currently being searched.
- Difference sets [14] - Instead of holding the entire tidset at any given time, Diffsets suggest that only changes in the tidsets are needed to compute the support.
- Dynamic Reordering [6] - A heuristic for reducing the search space - dynamically changing the order in which the search space is traversed. This attempts to rearrange the search space so that one can prune infrequent itemsets earlier rather than later.
- Parent Equivalence Pruning [4, 13] - Skipping levels in the search space, when a certain item added to the itemset contributes no new information.

To the best of our knowledge no previous implementation makes use of this combination of ideas, and some of these combinations are non-trivial to combine. For example, projection has never been previously used with difference sets and to do so requires some new observations as to how to combine these two elements.

We should add that there are a wide variety of other techniques introduced over time to find frequent itemsets, which we do not make use of. A very partial list of these other ideas is

- Sampling - [11] suggest searching over a sample of the dataset, and later validates the results using the entire dataset. This technique was shown to generate the vast majority of frequent itemsets.
- Adjusting support - [9] introduce SLPMiner, an algorithm which lowers the support as the itemsets grow larger during the search space. This attempts to avoid the problem of generating small itemsets which are unlikely to grow into large itemsets.

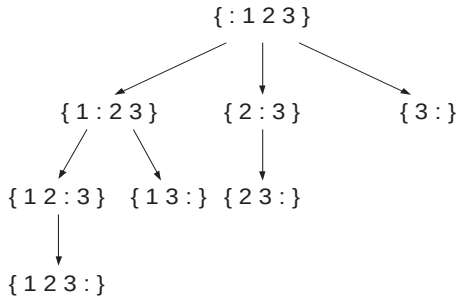

Figure 1. Full lexicographic tree of 3 items

3. The AIM-$\mathcal{F}$ algorithm

In this section we describe the building blocks that make up the AIM-$\mathcal{F}$ algorithm. High level pseudo code for the AIM-$\mathcal{F}$ algorithm appears in Figure 7.

3.1. Lexicographic Trees

Let $<$ be some lexicographic order of the items in I such that for every two items i and j, $i \neq j : i < j$ or $i > j$. Every node n of the lexicographic tree has two fields, n.head which is the itemset node n represent, and n.tail which is a list of items, possible extensions to n.head. A node of the lexicographic tree has a *l*evel. Itemsets for nodes at level k nodes contain k items. We will also say that such itemsets have length k. The root (level 0) node n.head is empty, and n.tail $= I$. Figure 1 is an example of lexicographic tree for 3 items.

The use of lexicographic trees for itemset generation was proposed by [8].

3.2. Depth First Search Traversal

In the course of the algorithm we traverse the lexicographic tree in a depth-first order. At node n, for every element α in the node's tail, a new node n' is generated such that n'.head $= n$.head $\bigcup \alpha$ and n'.tail $= n$.tail $- \alpha$. After the generation of n', α is removed from n.tail, as it will be no longer needed (see Figure 3).

Several pruning techniques, on which we elaborate later, are used in order to speed up this process.

3.3 Vertical Sparse Bit-Vectors

Comparison between horizontal and vertical database representations done in [10] shows that the representation of the database has high impact on the performance of the mining algorithm. In a vertical database the data is represented as a list of items,

```
Project(p : vector, v : vector )
/* p - vector to be projected upon
   v - vector being projected */
(1) t = Empty Vector
(2) i = 0
(3) for each nonzero bit in p, at offset j, in
    ascending order of offsets:
(4)     Set i'th bit of target vector t to be the
        j'th bit of v.
(5)     i = i + 1
(6) return t
```

Figure 2. Projection

```
DFS(n : node,)
(1) t = n.tail
(2) while t ≠ ∅
(3)     Let α be the first item in t
(4)     remove α from t
(5)     n'.head = n.head ⋃ α
(6)     n'.tail = t
(7)     DFS(n')
```

Figure 3. Simple DFS

where every item holds a list of transactions in which it appears.

The list of transactions held by every item can be represented in many ways. In [13] the list is a tid-list, while [10, 4] use vertical bit vectors. Because the data tends to be sparse, vertical bit vectors hold many "0" entries for every "1", thus wasting memory and CPU for processing the information. In [10] the vertical bit vector is compressed using an encoding called *skinning* which shrinks the size of the vector.

We choose to use a sparse vertical bit vector. Every such bit vector is built from two arrays - one for values, and one for indexes. The index array gives the position in the vertical bit vector, and the value array is the value of the position, see Figure 8. The index array is sorted to allow fast AND operations between two sparse bit vectors in a similar manner to the AND operation between the tid-lists. Empty values will be thrown away during the AND operation, save space and computation time.

3.3.1 Bit-vector projection

In [4], a technique called projection was introduced. Projection is a sparse bit vector compression technique specifically useful in the context of mining frequent

```
Apriori(n : node, minsupport : integer)
(1) t = n.tail
(2) while t ≠ ∅
(3)     Let α be the first item in t
(4)     remove α from t
(5)     n'.head = n.head ⋃ α
(6)     n'.tail = t
(7)     if (support(n'.head) ≥ minsupport)
(8)         Report n'.head as frequent itemset
(9)         Apriori(n')
```

Figure 4. Apriori

```
PEP(n : node, minsupport : integer)
(1) t = n.tail
(2) while t ≠ ∅
(3)     Let α be the first item in t
(4)     remove α from t
(5)     n'.head = n.head ⋃ α
(6)     n'.tail = t
(7)     if (support(n'.head) = support(n.head))
(8)         add α to the list of items removed by
            PEP
(9)     else if (support(n'.head) ≥ minsupport)
(10)        Report n'.head ⋃{All subsets of items
            removed by PEP} as frequent itemsets
(11)        PEP(n')
```

Figure 5. PEP

itemsets. The idea is to eliminate redundant zeros in the bit-vector - for itemset P, all the transactions which does not include P are removed, leaving a vertical bit vector containing only 1s. For every itemset generated from P (a superset of P), PX, all the transactions removed from P are also removed. This way all the extraneous zeros are eliminated.

The projection done directly from the vertical bit representation. At initialization a two dimensional matrix of 2^w by 2^w is created, where w is the word length or some smaller value that we choose to work with. Every entry (i,j) is calculated to be the projection of j on i (thus covering all possible projections of single word). For every row of the matrix, the number of bits being projected is constant (a row represents the word being projected upon).

Projection is done by traversing both the vector to project upon, p, and the vector to be projected, v. For every word index we compute the projection by table

DynamicReordering(n : node, minsupport : integer)
(1) $t = n$.tail
(2) for each α in t
(3) Compute $s_\alpha = \text{support}(n.\text{head} \bigcup \alpha)$
(4) Sort items α in t by s_α in ascending order.
(5) while $t \neq \emptyset$
(6) Let α be the first item in t
(7) remove α from t
(8) $n'.\text{head} = n.\text{head} \bigcup \alpha$
(9) $n'.tail = t$
(10) if (support($n'.head$) $\geq$ minsupport)
(11) Report n'.head as frequent itemset
(12) *DynamicReordering*(n')

Figure 6. Dynamic Reordering

lookup, the resulting bits are then concatenated together. Thus, computing the projection takes no longer than the AND operation between two compressed vertical bit lists.

In [4] projection is used whenever a *rebuilding threshold* was reached. Our tests show that because we're using sparse bit vectors anyway, the gain from projection is smaller, and the highest gains are when we use projection only when calculating the 2-itemsets from 1-itemsets. This is also because of the penalty of using projection with diffsets, as described later, for large k-itemsets. Even so, projection is used only if the sparse bit vector will shrunk significantly - as a threshold we set 10% - if the sparse bit vector contains less than 10% of '1's it will be projected.

3.3.2 Counting and support

To count the number of ones within a sparse bit vector, one can hold a translation table of 2^w values, where w is the word length. To count the number of ones in a word requires only one memory access to the translation table. This idea first appeared in the context of frequent itemsets in [4].

3.4 Diffsets

Difference sets (*Diffsets*), proposed in [14], are a technique to reduce the size of the intermediate information needed in the traversal using a vertical database. Using Diffsets, only the differences between the candidate and its generating itemsets is calculated and stored (if necessary). Using this method the intermediate vertical bit-vectors in every step of the DFS traversal are shorter, this results in faster intersections

AIM-$\mathcal{F}$(n : node, minsupport : integer)
/* Uses DFS traversal of lexicographic itemset tree
Fast computation of small frequent itemsets
for sparse datasets
Uses difference sets to compute support
Uses projection and bit vector compression
Makes use of parent equivalence pruning
Uses dynamic reordering */
(1) $t = n$.tail
(2) for each α in t
(3) Compute $s_\alpha = \text{support}(n.\text{head} \bigcup \alpha)$
(4) if ($s_\alpha = \text{support}(n.\text{head})$)
(5) add α to the list of items removed by PEP
(6) remove α from t
(7) else if ($s_\alpha <$ minsupport)
(8) remove α from t
(9) Sort items in t by s_α in ascending order.
(10) While $t \neq \emptyset$
(11) Let α be the first item in t
(12) remove α from t
(13) $n'.\text{head} = n.\text{head} \bigcup \alpha$
(14) $n'.\text{tail} = t$
(15) Report $n'.\text{head} \bigcup \{$All subsets of items removed by PEP$\}$ as frequent itemsets
(16) *AIM-$\mathcal{F}$*(n')

Figure 7. AIM-$\mathcal{F}$

between those vectors.

Let $t(P)$ be the tidset of P. The Diffset $d(PX)$ is the tidset of tids that are in $t(P)$ but not in $t(PX)$, formally : $d(PX) = t(P) - t(PX) = t(P) - t(X)$. By definition $\text{support}(PXY) = \text{support}(PX) - |d(PXY)|$, so only $d(PXY)$ should be calculated. However $d(PXY) = d(PY) - d(PX)$ so the Diffset for every candidate can be calculated from its generating itemsets.

Diffsets have one major drawback - in datasets, where the support drops rapidly between k-itemset to k+1-itemset then the size of $d(PX)$ can be larger than the size of $t(PX)$ (For example see figure 9). In such cases the usage of diffsets should be delayed (in the depth of the DFS traversal) to such k-itemset where the support stops the rapid drop. Theoretically the break even point is 50%: $\frac{t(PX)}{t(P)} = 0.5$, where the size of $d(PX)$ equals to $t(PX)$, however experiments shows small differences for any value between 10% to 50%. For this algorithm we used 50%.

Diffsets and Projection : As $d(PXY)$ in not a subset of $d(PX)$, Diffsets cannot be used directly for projection. Instead, we notice that $d(PXY) \subseteq$

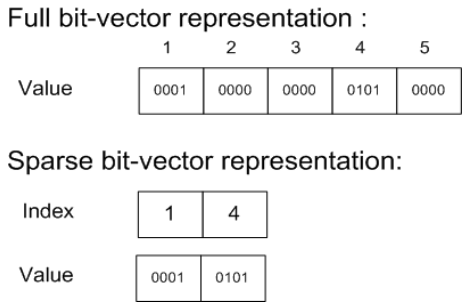

Figure 8. Sparse Bit-Vector data structure

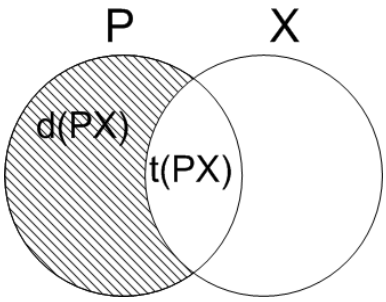

Figure 9. Diffset threshold

$t(PX)$ and $t(PX) = t(P) - d(PX)$. However $d(PX)$ is known, and $t(P)$ can be calculated in the same way. For example $t(ABCD) = t(ABC) - d(ABCD)$, $t(ABC) = t(AB) - d(ABC)$, $t(AB) = t(A) - d(AB)$ thus $t(ABCD) = t(A) - d(AB) - d(ABC) - d(ABCD)$. Using this formula the $t(PX)$ can be calculated using the intermediate data along the DFS trail. As the DFS goes deeper, the penalty of calculating the projection is higher.

3.5 Pruning Techniques

3.5.1 Apriori

Proposed by [2] the *Apriori* pruning technique is based on the monotonicity property of support: $\text{support}(P) \geq \text{support}(PX)$ as PX is contained in less transactions than P. Therefore if for an itemset P, $\text{support}(P) < \text{minsupport}$, the support of any extension of P will also be lower than minsupport, and the subtree rooted at P can be pruned from the lexicographic tree. See Figure 4 for pseudo code.

3.5.2 Parent Equivalence Pruning (PEP)

This is a pruning method based on the following property : If $\text{support}(n.\text{head}) = \textit{support}(n.\text{head} \bigcup \alpha)$ then all the transactions that contain $n.\text{head}$ also contain $n.\text{head} \bigcup \alpha$. Thus, X can be moved from the tail to the head, thus saving traversal of P and skipping to PX. This method was described by [4, 13]. Later when the frequent items are generated the items which were moved from head to tail should be taken into account when listing all frequent itemsets. For example, if k items were pruned using PEP during the DFS traversal of frequent itemset X then the all 2^k subsets of those k items can be added to X without reducing the support. This gives creating 2^k new frequent itemsets. See Figure 5 for pseudo code.

3.6 Dynamic Reordering

To increase the chance of early pruning, nodes are traversed, not in lexicographic order, but in order determined by support. This technique was introduced by [6].

Instead of lexicographic order we reorder the children of a node as follows. At node n, for all α in the tail, we compute $s_\alpha = \text{support}(t.\text{head} \bigcup \alpha)$, and the items are sorted in by s_α in increasing order. Items α in $n.\text{tail}$ for which $\text{support}(t.\text{head} \bigcup \alpha) < \text{minsupport}$ are trimmed away. This way, the rest of the sub-tree will benefit from a shortened tail. Items with smaller support, which are heuristically "likely" to be pruned earlier, will be traversed first. See Figure 6 for pseudo code.

3.7 Optimized Initialization

In sparse datasets computing frequent 2-itemsets can be done more efficiently than than by performing $\binom{n}{2}$ itemset intersections. We use a method similar to the one described in [13]: as a preprocessing step, for every transaction in the database, all 2-itemsets are counted and stored in an upper-matrix of dimensions $n \times n$. This step may take up to $O(n^2)$ operations per transaction. However, as this is done only for sparse datasets, experimentally one sees that the number of operations is small. After this initialization step, we are left with frequent 2 item itemsets from which we can start the DFS proceedure.

4. Experimental Results

The experiments were conducted on an Athlon 1.2Ghz with 256MB DDR RAM running Microsoft Windows XP Professional. All algorithms where compiled on VC 7. In the experiments described herein, we only count frequent itemsets, we don't create output.

We used five datasets to evaluate the algorithms performance. Those datasets where studied extensively in [13].

1. *connect* — A database of game states in the game connect 4.
2. *chess* — A database of game states in chess.
3. *mushroom* — A database with information about various mushroom species.
4. *pumsb** — This dataset was derived from the *pumsb* dataset and describes census data.
5. *T10I4D100K* - Synthetic dataset.

The first 3 datasets were taken from the UN Irvine ML Database Repository (http://www.ics.uci.edu/ mlearn/MLRepository). The synthetic dataset created by the IBM Almaden synthetic data generator (http://www.almaden.ibm.com/cs/quest/demos.html).

4.1 Comparing Data Representation

We compare the memory requirements of sparse vertical bit vector (with the projection described earlier) versus the standard tid-list. For every itemset length the total memory requirements of all tid-sets is given in figures 10, 11 and 12. We do not consider itemsets removed by PEP.

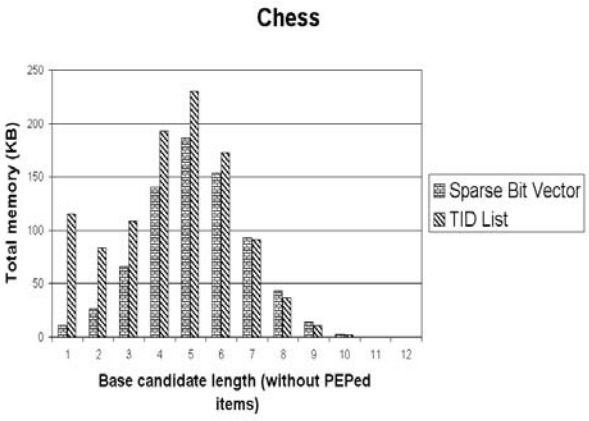

Figure 10. Chess - support 2000 (65%)

As follows from the figures, our sparse vertical bit vector representation requires less memory than tid-list for the dense datasets (chess, connect). However for the sparse dataset (T10I4D100K) the sparse vertical bit vector representation requires up to twice as much memory as tid-list. Tests to dynamically move from sparse vertical bit vector representation to tid-lists showed no significant improvement in performance, however, this should be carefully verified in further experiments.

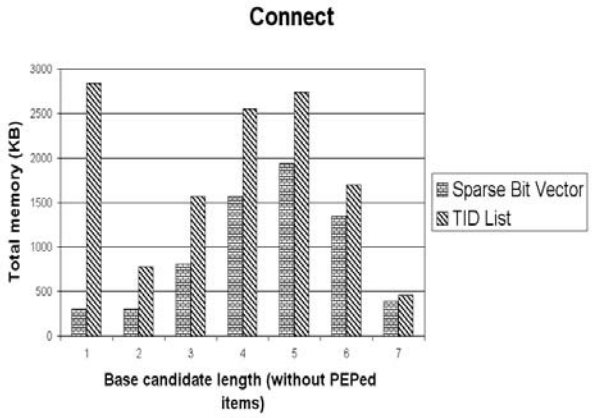

Figure 11. Connect - support 50000 (75%)

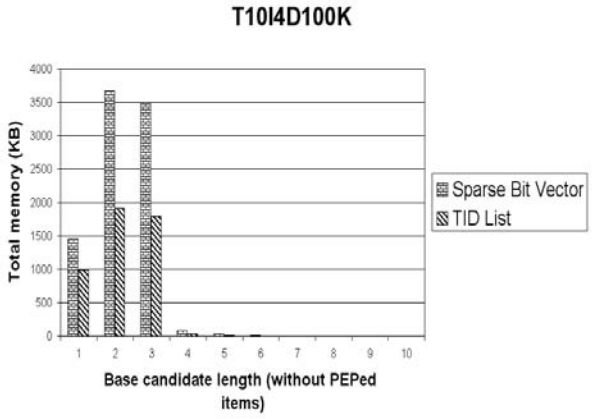

Figure 12. T10I4D100K - support 100 (0.1%)

4.2 Comparing The Various Optimizations

We analyze the influence of the various optimization techniques on the performance of the algorithm. First run is the final algorithm on a given dataset, then returning on the task, with a single change in the algorithm. Thus trying to isolate the influence of every optimization technique, as shown in figures 13 and 14.

As follows from the graphs, there is much difference in the behavior between the datasets. In the dense dataset, Connect, the various techniques had tremendous effect on the performance. PEP, dynamic reorder-

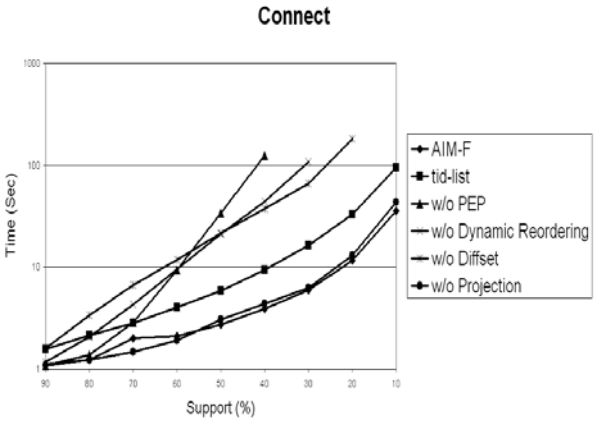

Figure 13. In¤uence of the various optimization on the Connect dataset mining

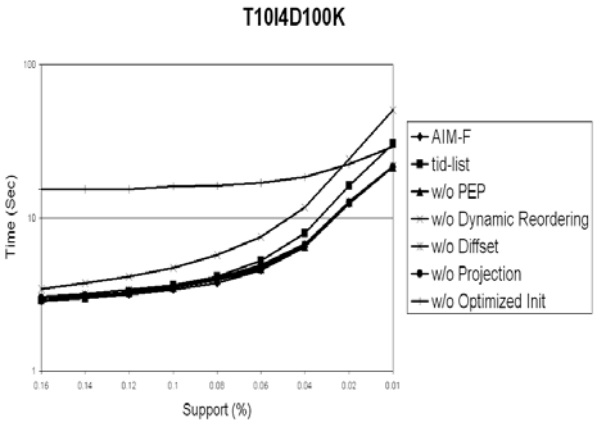

Figure 14. In¤uence of the various optimization on the T10I4D100K dataset mining

ing and diffsets behaved in a similar manner, and the performance improvement factor gained by of them increased as the support dropped. From the other hand the sparse bit vector gives a constant improvement factor over the tid-list for all the tested support values, and projection gives only a minor improvement.

In the second figure, for the sparse dataset T10I4D100K, the behavior is different. PEP gives no improvement, as can expected in sparse dataset, as every single item has a low support, and does not contain existing itemsets. There is drop in the support from k-itemset to k+1-itemset due to the low support therefore diffset also gives no impact, and the same goes for projection. A large gain in performance is made by optimized initialization, however the performance gain is constant, and not by a factor. Last is the dynamic reordering which contributes to early pruning much like in the dense dataset.

4.3 Comparing Mining Algorithms

For comparison, we used implementations of

1. Apriori [2] - horizontal database, BFS traversal of the candidates tree.
2. FPgrowth [5] - tree projected database, searching for frequent itemsets directly without candidate generation, and
3. dEclat [13] - vertical database, DFS traversal using diffsets.

All of the above algorithm implementations were provided by Bart Goethals (http://www.cs.helsinki/u/goethals/) and used for comparison with the AIM-$\mathcal{F}$ implementation.

Figures 15 to 19 gives experimental results on the various algorithms and datasets. Not surprising, Apriori [2] generally has the lowest performance amongst the algorithms compared, and in some cases the running time could not be computed as it did not finish even at the highest level of support checked. For these datasets and compared with the specific algorithms and implementations described above, our algorithm/implementation, AIM-$\mathcal{F}$, seemingly outperforms all others.

In general, for the dense datasets (Chess, Connect, Pumsb* and Mushroom, figures 15,16,17 and 18 respectively), the sparse bit vector gives AIM-$\mathcal{F}$ an order of magnitude improvement over dEclat. The diffsets gives dEclat and AIM-$\mathcal{F}$ another order of magnitude improvement over the rest of the algorithms.

For the sparse dataset T10I4D100K (Figure 19), the optimized initialization gives AIM-$\mathcal{F}$ head start, which

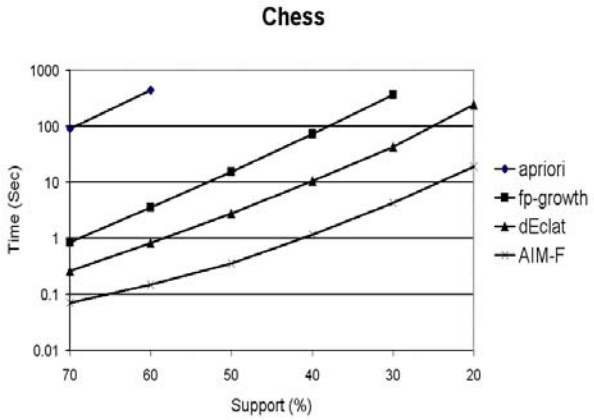

Figure 15. Chess dataset

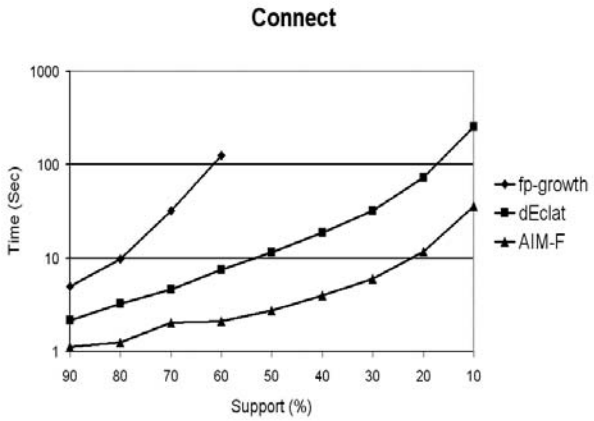

Figure 16. Connect dataset

is combined in the lower supports with the advantage of the sparse vertical bit vector (See details in figure 14)

5. Afterword

This paper presents a new frequent itemset mining algorithm, AIM-$\mathcal{F}$. This algorithm is based upon a mixture of previously used techniques combined dynamically. It seems to behave quite well experimentally.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

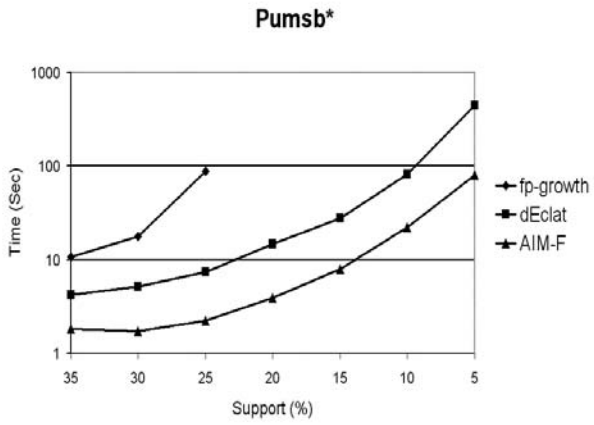

Figure 17. Pumsb* dataset

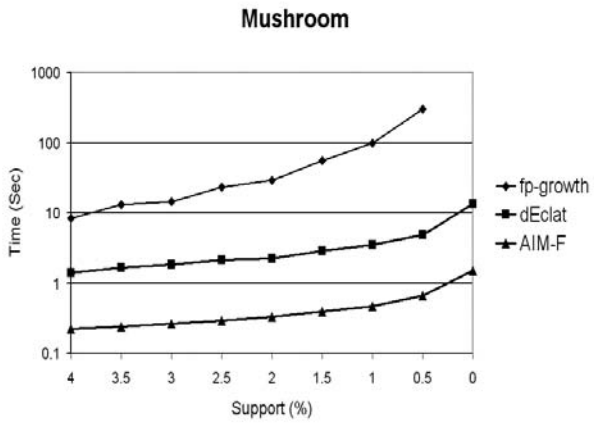

Figure 18. Mushroom dataset

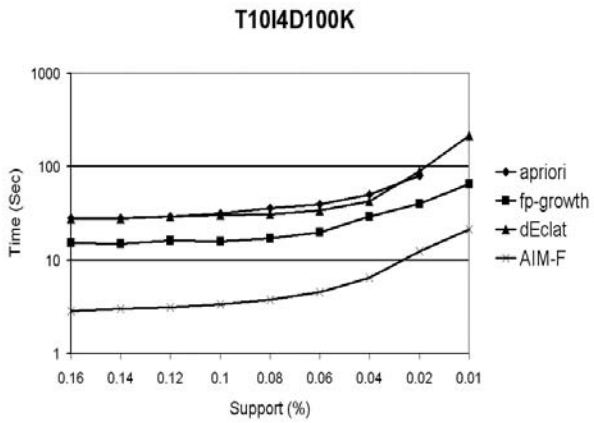

Figure 19. T10I4D100K dataset

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, pages 255–264, 1997.

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

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

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

[7] D.-I. Lin and Z. M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *EDBT'98*, volume 1377 of *Lecture Notes in Computer Science*, pages 105–119, 1998.

[8] R. Rymon. Search through systematic set enumeration. In *KR-92*, pages 539–550, 1992.

[9] M. Seno and G. Karypis. Slpminer: An algorithm for finding frequent sequential patterns using length decreasing support constraint. In *ICDE*, 2002.

[10] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *SIGMOD*, 2000.

[11] H. Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.

[12] S. Yen and A. Chen. An efficient approach to discovering knowledge from large databases. In *4th International Conference on Parallel and Distributed Information Systems*.

[13] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[14] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, RPI, 2001.