A Compression Algorithm for Mining Frequent Itemsets

Raudel Hernández León[1,2], Airel Pérez Suárez[1,2], and Claudia Feregrino-Uribe[1]

[1] National Institute for Astrophysics, Optics and Electronics, INAOE, MEXICO
{raudel,airel,cferegrino}@ccc.inaoep.mx
[2] Advanced Technologies Application Center, CENATAV, CUBA
{rhernandez,asuarez}@cenatav.co.cu

Abstract In this chapter, we propose a new algorithm for mining frequent itemsets. This algorithm is named *AMFI* (*Algorithm for Mining Frequent Itemsets*), it compresses the data while maintains the necessary semantics for the frequent itemsets mining problem and, for this task, it is more efficient than other algorithms that use traditional compression algorithms. The *AMFI* efficiency is based on a compressed vertical binary representation of the data and on a very fast support count. *AMFI* introduces a novel way to use equivalence classes of itemsets by performing a breadth first search through them and by storing the class prefix support in compressed arrays. We compared our proposal with an implementation that uses the *PackBits* algorithm to compress the data.

Keywords: data mining, frequent patterns, compression algorithms

1 Introduction

Mining association rules in transaction datasets has been demonstrated to be useful and technically feasible in several application areas, particularly in retail sales [1, 2, 3, 4], document datasets applications [5], and also in intrusion detection [6]. Association rule mining is usually divided in two steps. The first one consists of finding all itemsets appearing on the dataset (or having a support) above a certain threshold, these itemsets are called *frequent itemsets* (FI). The second one consists on extracting association rules from the FI found in the first step. The FI mining task is very difficult because of its exponential complexity, for that reason the work developed in this chapter will focus on the first step.

The management and storage of large datasets have always been a problem to solve in data mining, particularly in FI mining where the representation of data is decisive to compute the itemset supports.

Conceptually, a dataset is a two-dimensional matrix where the rows represent the transactions and the columns represent the items. This matrix can be implemented in the following four different formats [7]:

– Horizontal item-list (HIL): The dataset is represented as a set of rows (transactions) where each row stores an ordered list of items.

– Horizontal item-vector (HIV): This is similar to HIL, except that each row stores a bit-vector of 1's and 0's to represent the presence or absence of each item.
– Vertical Tid-list (VTL): The dataset is represented as a set of columns (items) where each column stores an ordered list of transactions *ids* in which the item is contained. Note that the VTL format needs exactly the same space as the HIL format.
– Vertical tid-vector (VTV): This is similar to VTL, except that each column stores a bit-vector of 1's and 0's to represent the presence or absence of the item in each transaction. Note that the VTV format needs exactly the same space as the HIV format.

Many algorithms have been proposed using vertical binary representations (VTV) in order to improve the process of obtaining FI [7, 8, 9, 10]. The fact that the presence or absence of an item in a transaction can be stored in a bit and that thousands of transactions can be present in a single matrix suggests the use of compression algorithms on the data. To find an efficient algorithm that compresses the data while maintaining the necessary semantic for the FI mining problem is the goal of this work.

We propose an algorithm based on a breadth first search [1] (BFS) through equivalence classes [11] combined with a compressed vertical binary representation of the dataset. This compressed representation, in conjunction with the equivalence class processing, produces a very fast support count and it produces a less expensive representation, specially in large sparse datasets.

In this chapter formal definitions are given and some compression algorithms including *PackBits* method are described following with the explanation of $AMFI$ and the pseudo code of the algorithm. Experimental results are discussed and finally the conclusions drawn from this work are presented.

2 Preliminaries

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items. Let D be a set of transactions, where each transaction T is a set of items, so that $T \subseteq I$. An itemset X is a subset of I. The support of an itemset X is the number of transactions in D containing to X. If the support of an itemset is greater than or equal to a given support threshold (*minSup*), the itemset is called a *frequent itemset* (FI). The size of an itemset is defined as its cardinality; an itemset containing k items is called a k-itemset.

For example, in Table 1, if we have a support threshold equal to three, the FI obtained are: {coke}, {diaper}, {beer} and {diaper, beer}.

In [11], the authors proposed partitioning the itemset space into equivalence classes. These equivalence classes are defined by the equivalence relation "k-

[1] In graph theory, BFS is a search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

Table 1: Transactional datasets

id	**ítems**
1	coke, milk
2	bread, diaper, beer
3	coke, diaper, beer
4	pan, diaper, beer
5	coke, milk, diaper

itemsets sharing its first $k-1$ items belong to the same equivalence class EC_k", therefore all elements of EC_k have size k (see Fig.1).

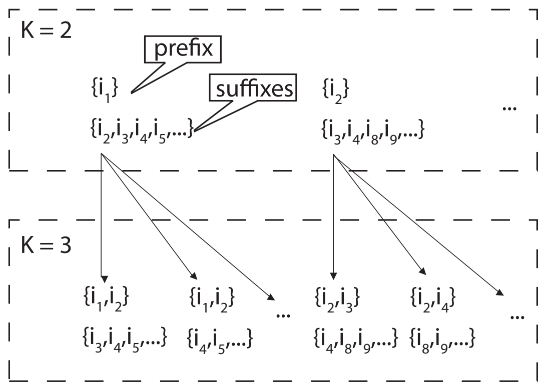

Figure 1: Equivalence classes

Each equivalence class of level $k-1$ generates several equivalence classes at level k.

Most of the algorithms for finding FI are based on the *Apriori* algorithm [2]. To achieve an efficient frequent patterns mining, an anti-monotonic property of frequent itemsets, called the *Apriori* heuristic, was formulated [2]. The basic intuition of this property is that any subset of a frequent itemset must be frequent. *Apriori* is a BFS algorithm, with an *HIL* organization, that iteratively generates two kinds of sets: C_k and L_k. The set L_k contains the frequent k-itemsets. Meanwhile, C_k is the set of candidate k-itemsets, representing a superset of L_k. This process continues until a null set L_k is generated.

The set L_k is obtained by scanning the dataset and determining the support for each candidate k-itemset in C_k. The set C_k is generated from L_{k-1} following the next procedure.

$$C_k = \{c \mid \text{Join}(c, L_{k-1}) \wedge \text{Prune}(c, L_{k-1})\} \tag{1}$$

where:

$$\begin{aligned} &\text{Join}(\{i_1, i_2, \ldots, i_{k-1}, i_k\}, L_{k-1}) \equiv \\ &\langle \{i_1, \ldots, i_{k-1}\} \in L_{k-1} \wedge \{i_1, \ldots, i_k\} \in L_{k-1} \rangle, \end{aligned} \tag{2}$$

$$\begin{aligned} &\text{Prune}(c, L_{k-1}) \equiv \\ &\langle \forall s[(s \subset c \wedge |s| = k-1) \rightarrow s \in L_{k-1}] \rangle, \end{aligned} \tag{3}$$

The main problem about the computation of FI is the support counting, that is, computing the number of times that an itemset appears in the dataset. To choose a compression algorithm suitable for data compacting and later on to compute the FI is not an easy task.

3 Compression Algorithms

The compression of a transactional dataset can be performed horizontally or vertically. Taking into account the characteristics of the problem, horizontal compaction can be a throwaway due to transactions being defined as sets of items: the sets do not have repeated elements, for which no redundancy is present for a compression algorithm be able to work properly. When data are vertically represented, a transactional identifier list or vector can be obtained per every item. Then, to compute the support for an itemset under this representation, it is required to intersect a transactional identifier list from transactions associated to each item.

Recently there have been in the literature some works about using compression for frequent itemsets. One of them is [12] that proposes to use a dynamic clustering method to compress the frequent itemsets approximately. *Expression similarity* and *support similarity* are defined according to the requirements of the frequent itemset compression. Authors mention that with their method user's do not need to specify the number of frequent itemsets clusters explicitly and user's expectation of compression ratio is incorporated. They claim that their method is feasible and the compression quality is good. Another work is [13] that proposes an algorithm for vertical association rule mining that compresses a vertical dataset using bit vectors. Authors claim that their algorithm needs only a small amount of memory compared to other compression techniques that had been used by many association rule mining algorithms. Yet another scheme that uses compression to find the most interesting frequent itemsets is [14] that uses the principle of *the best set of frequent itemsets is that set that compresses the database best*. Rather than compressing the set of frequent items, they compress the database.

Since the last mid-century, many compression algorithms have been developed [15, 16, 17, 18, 19]. In all lossless compression implementations, there is a trade-off between computational resources and the compression ratio. Often, in both statistical and dictionary-based methods, the best compression ratios are obtained at expenses of long execution times and high memory requirements.

Statistical compressors are characterized by consuming higher resources than dictionary based when they are implemented in both software and hardware, however they can achieve compression ratios near to the source entropy. The most demanding task in this kind of algorithms is the implementation of the model to get the statistics of the symbols and to assign the bit string. Perhaps, the most representative statistical method is the proposed by Huffman [15] in 1952. In this algorithm a tree is built according to the frequency of the symbols. All symbols are placed at the leaves of the tree. The Huffman method achieves compression by replacing every symbol by a variable bit string. The bit string assigned to every symbol is determined by visiting every internal node from the root up to the leaf corresponding to the symbol. Initially the bit string is the null string. For every internal node visited, one bit is concatenated to the bit string, 1 or 0, depending on the current visited node whether it is a right or left child of its father. Symbols at longer branches will be assigned larger bit strings.

In the dictionary-based methods, the most time-consuming task is searching for strings in a dictionary, which usually has hundreds of locations. Dictionary-based algorithms are considered simpler to implement than statistical ones but the compression ratio that can be achieved is lower. Another kind of compression algorithms, *ad-hoc*, that were developed in early days of data compression are Run Length Encoding-like (RLE) algorithms [20]. RLE takes advantage of the presence of consecutive identical single symbols often found in data streams. It replaces long runs of repeated symbols with a special token and the length of the run. This method is particularly useful for small alphabets and provides better compression ratios when symbols are correlated with their predecessors.

Selecting a compression method among the existent ones is non-trivial. While one method may be faster, other may achieve better compression ratio and yet another may require less computational resources. Furthermore, due to the nature of mining frequent itemsets, using these algorithms for the transactional identifier list compression, the semantics required for the intersection are lost, bringing as a consequence the necessity of decompressing before intersecting.

After a careful analysis of existing compression algorithms, we concluded that RLE type of algorithms are more suitable for compressing our data. In [21] several variants of RLE algorithm are described, however, [22] describes a variant that in our opinion, can adjust better to the type of data managed here and it may compress with higher compression rates besides allowing intersecting without requiring decompression.

3.1 PackBits Algorithm

PackBits algorithm is a fast and simple compression scheme for run-length encoding of data. A *PackBits* data stream consists of packets of one byte of header followed by data. The header is a signed byte; the data can be signed or unsigned.

In the following table, let n be the value of the header byte as a signed integer.

Note that interpreting 0 as positive or negative makes no difference in the output. Runs of two bytes adjacent to non-runs are typically written as literal

Table 2: Data stream of PackBits

Header byte	Data following the header byte
0 to 127	$(1+n)$ literal bytes of data
-1 to -127	One byte of data, repeated $(1-n)$ times in the decompressed output
-128	No operation (skip and treat next byte as a header byte)

data. It should also be noticed that it is impossible, from the *PackBits* data, to determine the end of the data stream; i.e., one must know a priori the size of the uncompressed data before reading a *PackBits* data stream to know where it ends.

4 Characteristic of *AMFI* Algorithm

A new algorithm for FI mining is proposed in this section. The efficiency of this algorithm is based on a compressed vertical binary representation of the data and on a very fast support count.

4.1 Storing the Transactions

Let us call *filtered transaction* to the itemset that is obtained by removing no-frequent items from a transaction. The size of the filtered transactions is obviously smaller than the size of the dataset. Based on the anti-monotonic property of FI [2], all FI of a dataset can be computed even if only filtered transactions are available.

The set of filtered transactions can be represented as an m x n matrix, where m is the number of transactions and n is the number of frequent items. We can denote the presence or absence of an item in each transaction by a binary value (1 if it is present, 0 otherwise).

If the maximum number of transactions is not greater than the *CPU* word size w (32 or 64 bits), the dataset can be stored as a simple set of integers. However, a dataset is normally much greater than the *CPU* word size. For this reason, we propose to use an array of integers to store the presence or not of each frequent item along the transactions. It will be explained later on how to extend these integer arrays to a frequent itemset.

Let M be the binary representation of a dataset, with n items and m transactions. Retrieving from M the columns associated to frequent items, we can represent each item j as an integer array I_j where each integer has size w, as follows:

$$I_j = \{W_{1,j}, W_{2,j}, \ldots, W_{q,j}\},\ q = \lceil m/w \rceil \tag{4}$$

where each integer of the array can be defined as:

$$W_{k,j} = \sum_{r=1}^{w} 2^{w-r} * M_{((k-1)*w+r),j} \tag{5}$$

being $M_{i,j}$ the bit value of item j in transaction i, in case of $i > m$ then $M_{i,j} = 0$. This representation of FI allows a fast counting of the itemset support in large datasets.

4.2 Reordering of Frequent 1-itemsets

As other authors, in *AMFI*, we have used the heuristic of reordering the frequent 1-itemsets in increasing support order. This will cause a reduction of candidate sets in the next level. This heuristic was first used in MaxMiner [23], and has been used in other methods since then [8, 24, 25, 26, 27, 28, 29]. In the case of our algorithm, reordering frequent 1-itemsets contributes to a faster convergence, as well as saving memory.

4.3 AMFI Algorithm

AMFI is a BFS algorithm through equivalence classes with a compressed vertical binary representation. This algorithm iteratively generates a list EC_k. The elements of this list represent the equivalence classes of size k and have the format:

$$\langle \text{Prefix}_{k-1}, \text{IA}_{\text{Prefix}_{k-1}}, \text{Suffixes}_{\text{Prefix}_{k-1}} \rangle, \tag{6}$$

where Prefix_{k-1} is the $(k-1)$-itemset that is common to all the itemsets of the equivalence class, $\text{Suffixes}_{\text{Prefix}_{k-1}}$ is the set of all items j which extend to Prefix_{k-1}, where j is lexicographically greater than any item in the prefix, and $\text{IA}_{\text{Prefix}_{k-1}}$ is an array of non null integers that is built with the intersection (using AND operation) of the arrays I_j, where j belongs to Prefix_{k-1}. The IA arrays store the accumulated supports of the prefix of each equivalence class EC_k. If k is larger then the number of elements of IA is lesser because the AND operation generates null integers, and null integers are not stored because they do not have influence on the support. The procedure for obtaining IA is as follows: Let i and j be two frequent items,

$$\begin{aligned} IA_{\{i\}\cup\{j\}} = & \\ & \{(W_{k,i} \ \& \ W_{k,j}, k) \ |(W_{k,i} \ \& \ W_{k,j}) \neq 0, k \in [1,q]\}, \end{aligned} \tag{7}$$

similarly, let the frequent itemset X and the frequent item j

$$\begin{aligned} IA_{X\cup\{j\}} = \{(b \ \& \ W_{k,j}, k) \ | & \\ & (b,k) \in IA_X, (b \ \& \ W_{k,j}) \neq 0, k \in [1,q]\}, \end{aligned} \tag{8}$$

This representation not only reduces the required memory space to store the integer arrays but also eliminates the *Join* step described in (2).

In order to compute the support of an itemset X with an integer-array IA_X, the following expression is used:

$$\mathrm{Support}(IA_X) = \sum_{(b,k)\in IA_X} \mathrm{BitCount}(b) \tag{9}$$

where BitCount(b) is a function that calculates the Hamming Weight of b. The IA cardinality is reduced with the increment of the itemsets size due to the downward closure property. It allows for improvement of the processes (8) and (9).The *AMFI* algorithm pseudo code is shown in Algorithm 1.

```
   **Input**: Dataset in binary representation
   **Output**: Frequent itemsets
 1 Answer = $\emptyset$
 2 $L$ = {frequent 1-itemsets}
 3 forall $i \in L$ do
 4     ECGenAndCount($\langle\{i\}, I_i, Suffixes_{\{i\}}\rangle, EC_2$)
 5     $k = 3$
 6     while $EC_{k-1} \neq \emptyset$ do
 7         forall $ec \in EC_{k-1}$ do
 8             ECGenAndCount($ec, EC_k$)
 9         end
10         Answer = Answer $\cup$ $EC_k$
11         $k = k + 1$
12     end
13 end
14 return $Answer$
```

Algorithm 1: AMFI

In line 2 of algorithm 1, the frequent 1-itemsets are calculated and ordered in increasing support order. In line 4, the equivalence classes of each frequent 1-itemset are built. From line 6 to 12, each equivalence class of size 2 is processed using ECGenAndCount function. The ECGenAndCount function takes an equivalence class of length $k-1$ as argument and generates a set of equivalence classes of length k (see Algorithm 2).

In line 2 of algorithm 2, all the items i that form the suffix of the input equivalence class (EC_{k-1}) are crossed. In line 3 the prefixes Prefix$'$ of the equivalence classes of level k are built by adding each suffix i to the prefix of EC_{k-1}. In line 4, the IA array associated to each Prefix$'$ is calculated by means of *AND* operation between the IA of EC_{k-1} and the I_i associated to the item i (8). From lines 6 to 13, the suffix items j of EC_{k-1}, lexicographically greater than i, are crossed and the support of the sets Prefix$'$ $\cup$ j is calculated.

Input: An equivalence class in $\langle \text{Prefix}, \text{IA}_{\text{Prefix}}, \text{Suffixes}_{\text{Prefix}} \rangle$ format
Output: The equivalence classes set generated

```
Answer = $\emptyset$
forall $i \in Suffixes_{Prefix}$ do
    $\text{Prefix}' = \text{Prefix} \cup \{i\}$
    $\text{IA}_{\text{Prefix}'} = \text{IA}_{\text{Prefix} \cup \{i\}}$
    $\text{Suffixes}'_{\text{Prefix}'} = \emptyset$
    forall $(i' \in Suffixes_{Prefix})$ and $(i' > i)$ do
        if Support($IA_{Prefix' \cup \{i'\}}$) then
            $\text{Suffixes}'_{\text{Prefix}'} = \text{Suffixes}'_{\text{Prefix}' \cup \{i'\}}$
        end
    end
    if $Suffixes'_{Prefix'} \neq \emptyset$ then
        $\text{Answer} = \text{Answer} \cup \{\langle \text{Prefix}', \text{IA}_{\text{Prefix}'}, \text{Suffixes}'_{\text{Prefix}'} \rangle\}$
    end
end
return Answer
```

Algorithm 2: ECGenAndCount

4.4 Memory Considerations

As mentioned, a data set can be represented in four ways, horizontally (HIL and HIV) and vertically (VTL and VTV). Making a decision between VTL and VTV representations is a non trivial task. Burdick, Calimlim and Gehrke in [8] analyzed these two vertical formats when they proposed the MAFIA algorithm. They showed, experimentally, that the memory efficacy of these representations depends on the density of the dataset. Particularly, on 32 bit machines, the VTL format is guaranteed to be a more expensive representation in terms of space if the support of an item (or itemset) is greater than 1/32 or about 3% of the number of transactions. In the VTL representation, we need an entire word to represent the presence of an item versus the single bit of the VTV approach.

In the compressed IA array of the CA algorithm, a pair of integers for each one of the simple (uncompressed) VTV format is required. As the CA algorithm representation includes pairs of words only for non null integers, the memory overhead of this representation is higher than the simple VTV if the support of an item (or itemset) is greater than 1/2. Furthermore, considering a dataset of m transactions on 32 bit machines and an item (or itemset) with a support sup, a simple VTV requires $m/8$ bytes of memory while the CA algorithm, in its worst case, when the item (or itemset) transactions are sparsely distributed, requires $8 * \min(m * sup, m/32)$ bytes.

So, we can infer that if the support of an item (or itemset) is less than 1/64 or about 1.5% the CA algorithm has a representation less expensive. Therefore, we can conclude that the compressed integer array representation is better than the simple VTV representations in sparse datasets with *minSup* less than 1.5% or, in the dense datasets, less than 50%.

5 Experimental Results

Several experiments were carried out where our proposed algorithm, *AMFI*, was compared against a version that compresses the data using PackBits. Time consumption and memory requirements were considered as measurements of efficiency.

Experiments were developed with two newsitem datasets and two synthetic datasets (Table 3).

Table 3: Summary of the main datasets characteristics

	Transactions	Items Count	Avg. Length
El Pais	550	14489	173.1
TDT	8169	55532	133.5
Kosarak	990002	41935	8.1
Webdocs	1704140	5266562	175.98

Some of these datasets are sparse, such as *El Pais*, and some, very sparse, such as *Webdocs*. Newsitems datasets were lemmatized using the *Treetagger* program [30], and the *stopwords* were eliminated.

The *Kosarak* dataset was provided by Ferenc Bodon to *FIMI* repository [31] and contains (anonymized) click-stream data of a Hungarian on-line news portal. The *Webdocs* dataset was built from a spidered collection of web html documents and was donated to *FIMI* repository by Claudio Lucchese *et al.* *TDT* dataset contains news (newsitems) data collected daily from six news sources in American English, over a period of six months (January - June, 1998). The *El Pais* dataset contains 550 news, published at El Pais (Spain) newspaper in June in 1999.

Our tests were performed on a PC with an Intel Core 2 Duo at 1.86 GHz CPU and 1 GB DDR2 RAM. The operating system was Windows XP SP2. We considered CPU+IO time (in seconds) at execution time for all algorithms included in this paper.

In figures 2, 3, 4 and 5, a comparison of memory consumption by level is shown, meaning that the comparison is done for frequent 1-itemsets, frequent 2-itemsets and so on until frequent 6-itemset. We plot the values until level 6 in order to not overload the graphics, but the performance is the same.

As it can be seen from the figures, the *AMFI* algorithm requires less memory than the variant of *PackBits* as the size of the FI increases. In level 1, *AMFI* consumes more memory since it stores all the bytes while *PackBits* compresses the bytes with equal value (as the datasets are very sparse, see Table 3, many bytes are equal to 0).

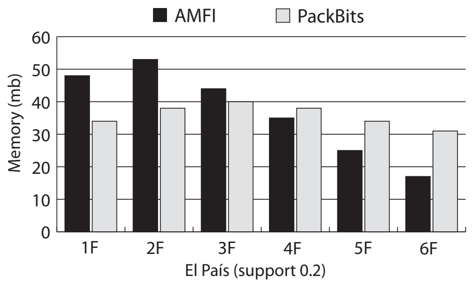

Figure 2: Memory consumption (El Pais dataset)

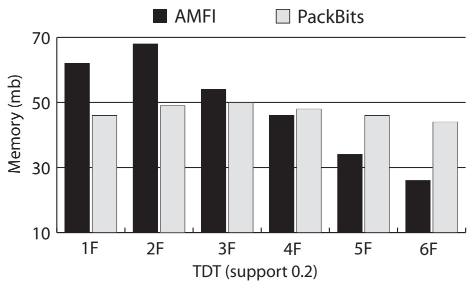

Figure 3: Memory consumption (TDT dataset)

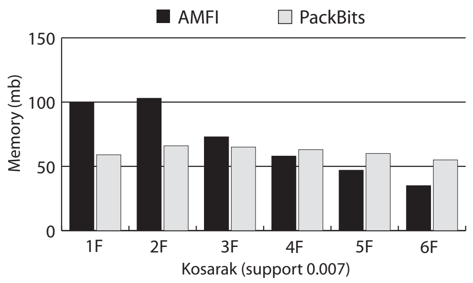

Figure 4: Memory consumption (Kosarak dataset)

As the levels increase, *PackBits* requires always to compress a constant amount of bytes, while *AMFI* will store only the bytes that are different from 0, which diminish fast due to the intersection operations.

In figures 6, 7, 8 and 9 a comparison of execution time with different supports is shown. As it can be seen, *AMFI* algorithm not only requires less memory but also is more efficient. This is mainly due to the number of blocks different from 0 decreases with increasing levels (the size of the itemsets) and intersecting only these blocks is faster than iterating two compressed byte flows that intersect all the blocks.

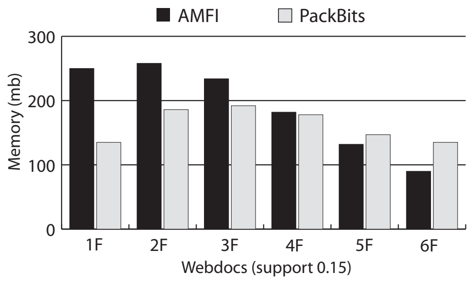

Figure 5: Memory consumption (Webdocs dataset)

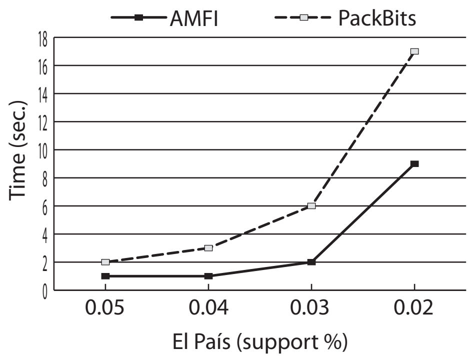

Figure 6: Time consumption (El Pais dataset)

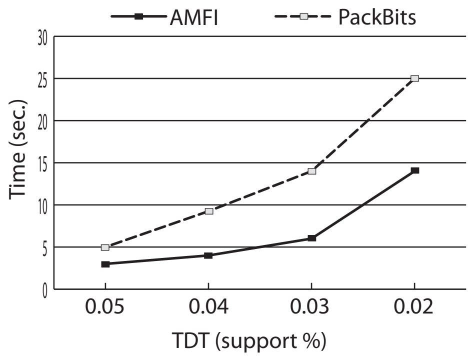

Figure 7: Time consumption (TDT dataset)

To show how decreasing the number of nonzero blocks in *AMFI* algorithm, we took the two largest datasets and we computed the average value of the number of elements of the IA array in the equivalence classes from EC_2 to EC_9 (Table 4). This average value indicates the average number of intersections needed to compute the support of an FI in each equivalence class. It is important

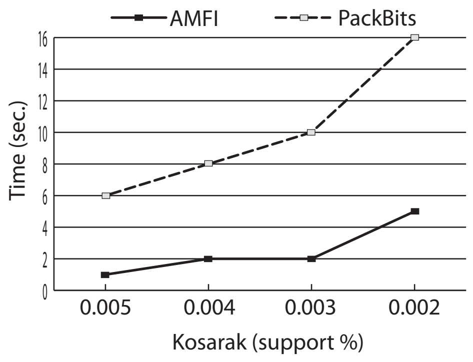

Figure 8: Time consumption (Kosarak dataset)

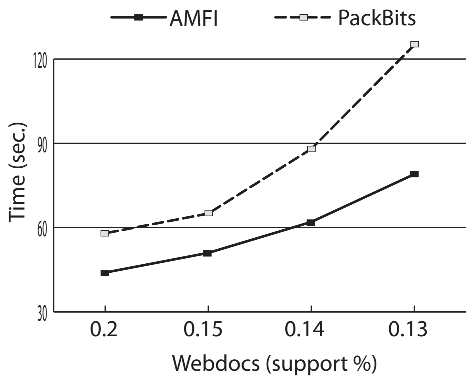

Figure 9: Time consumption (Webdocs dataset)

to highlight that the number of intersections is large only in the processing of EC_2 and, the average of zero blocks generated in EC_k is equal to the difference between the average of intersections needed for processing EC_{k+1} and EC_k respectively.

The values shown in table 4 corroborate the results obtained in the conducted experiments.

Table 4: Average value of the number of elements of IA from EC_2 to EC_9

Datasets	EC_2	EC_3	EC_4	EC_5	EC_6	EC_7	EC_8	EC_9
Kosarak	30938	3703	2981	2420	2114	2029	2000	1980
Webdocs	53255	4208	3927	3221	2415	2212	2183	1813

6 Conclusions

In this chapter we have presented a compressed vertical binary approach for mining FI. In order to reach a fast support computing, our algorithm uses a BFS through equivalence classes storing the class prefix support in compressed arrays. The experimental results showed that *AMFI* algorithm achieves better performance than *PackBits* as much in consumption of memory as in run time. It can be concluded that although existing compression methods are good, they are not always suitable for certain problems due to when compressing the required semantic is lost.

References

[1] Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C. (1993) 207–216

[2] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In Proceedings of the 20*th* International Conference on Very Large Data Bases, VLDB'94, Santiago de Chile, Chile (1994) 487–499

[3] Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In Technical Report GIT-CC-95-04, Institute of Technology, Atlanta, USA (1995)

[4] Brin, S., Motwani, R., Ullman, J.D., Tsur, S.: Dynamic itemset counting and implication rules for market basket data. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, USA (1997)

[5] Feldman, R., Dagan, I.: Kdt-knowledge discovery in texts. In Proceedings of the First International Conference on Knowledge Discovery (KDD) (1995) 112–117

[6] Silvestri, C., Orlando, S.: Approximate mining of frequent patterns on streams. Intell. Data Anal., Vol. 11, No. 1 (2007) 49–73

[7] Shenoy, P., Haritsa, J., Sudarshan, S., Bhalotia, G., Bawa, M., Shah, D.: Turbo-charging vertical mining of large databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, USA (2000)

[8] Burdick, D., Calimlim, M., Gehrke, J.: Mafia: A maximal frequent itemset algorithm for transactional databases. In Proceedings of the International Conference on Data Engineering (ICDE), Heidelberg, Germany (2001)

[9] Grahne, G., Zhu, J.: Fast algorithms for frequent itemset mining using fp-trees. IEEE Transactions on Knowledge and Data Engineering, Vol. 17, No. 10 (2005) 1347–1362

[10] Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In Proceedings ACM-SIGMOD International Conference on Management of Data, New York, NY, USA (2000)

[11] Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In Proceedings of the 3rd International Conference on KDD and Data Mining, EU (1997)

[12] Yan, H., Sang, Y.: Approximate frequent itemsets compression using dynamic clustering method. IEEE Conference on Cybernetics and Intelligent Systems (2008) 1061–1066

[13] Mafruz Zaman Ashrafi, D.T., Smith, K.: An efficient compression technique for frequent itemset generation in association rule mining. Advances in Knowledge Discovery and Data Mining, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, Vol. 3518 (2005) 125–135

[14] Arno Siebes, Jilles Vreeken, M.v.l.: Item sets that compress. Proceedings of the SDM'06 (2006) 393–404

[15] Huffman, D.: A method for the construction of minimum redundancy codes. In Proceedings of the IRE 40(9) (1952) 1098–1101

[16] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3) (1977) 337–343

[17] Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory IT-24(5) (1978) 530–536

[18] Fiala, E.R., Greene, D.H.: Data compression with finite windows. Communications of the ACM 32(4) (1989) 490–505

[19] Phillips, D.: Lzw data compression. The Computer Application Journal Circuit Cellar Inc., 27 (1992) 36–48

[20] W., G.S.: Run-length encodings. IEEE Transactions on Information Theory, 12 (1966) 399–401

[21] Salomon, D.: Data compression: The complete reference. 3rd Edition, Published by Springer. ISBN 0-387-40697-2. LCCN QA76.9 D33S25, 899 pages (2004)

[22] http://www.fileformat.info/format/tiff/corion-packbits.htm

[23] Bayardo, R.J.: Efficiently mining long patterns from databases. In ACM SIGMOD Conf. on Management of Data (1998) 85–93

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

[25] Gouda, K., Zaki, M.J.: Genmax: An efficient algorithm for mining maximal frequent itemsets. Data Mining and Knowledge Discovery, 11 (2005) 1–20

[26] Zaki, M.J., Hsiao, C.J.: Charm: An efficient algorithm for closed itemset mining. In 2nd SIAM International Conference on Data Mining (2002) 457–473

[27] Calders, T., Dexters, N., Goethals, B.: Mining frequent itemsets in a stream. Proceedings of the IEEE International Conference on Data Mining (2007) 83–92

[28] Calders, T., Dexters, N., Goethals, B.: Mining frequent items in a stream using flexible windows. Intelligent Data Analysis, Vol. 12, No. 3 (2008)

[29] Kalpana, B., Nadarajan, R.: Incorporating heuristics for efficient search space pruning in frequent itemset mining strategies. CURRENT SCIENCE 94 (2008) 97–101

[30] Schmid, H.: Probabilistic part-of-speech tagging using decision trees. In: International Conference on New Methods in Language Processing, (Software in: www.ims.uni-stuttgart.de/ftp/pub/corpora/tree-tagger1.ps.gz), Manchester, UK (1994)

[31] FIMI-Frequent Itemset Mining Implementations Repository, (Software developed by Ferec Bodon, URL: http://fimi.cs.helsinki.fi/src)